Vers la compilation vérifiée de Sea of Nodes : propriétés et raisonnement sémantiques

Yon Fernández de Retana

▶ **To cite this version:**

Yon Fernández de Retana. Vers la compilation vérifiée de Sea of Nodes : propriétés et raisonnement sémantiques. Génie logiciel [cs.SE]. Université de Rennes, 2018. Français. NNT : 2018REN1S020 . tel-01865395

HAL Id: tel-01865395

https://theses.hal.science/tel-01865395v1

Submitted on 31 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2018

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale (MathSTIC)

présentée par

Yon Fernández de Retana

préparée à l'unité de recherche IRISA
Institut de recherche en informatique et systèmes aléatoires

UFR Informatique Électronique (ISTIC)

Vers la compilation vérifiée de *Sea of Nodes* : propriétés et raisonnement sémantiques

Thèse soutenue à Rennes
le 5 Juillet 2018

devant le jury composé de :

David MONNIAUX
Directeur de recherches – Laboratoire Verimag / rapporteur

Alain DARTE
Directeur de recherches – Laboratoire de l'Informatique du Parallélisme / rapporteur

Thibaut BALABONSKI
Maître de conférences – Université Paris-Sud, LRI / examinateur

Thomas JENSEN
Directeur de recherches – INRIA Rennes / examinateur

David PICHARDIE
Professeur des universités – ENS Rennes, IRISA / directeur de thèse

Delphine DEMANGE
Maître de conférences – Univ Rennes 1, IRISA / co-encadrant de thèse

Table des matières

Chapitre 1

Introduction

Ce manuscrit est consacré à l'étude formelle de représentations intermédiaires optimisantes en compilation sous l'angle de la compilation vérifiée.

Dans ce chapitre, nous présentons les problématiques des compilations optimisante et *vérifiée*. Nous introduisons ensuite la problématique motrice de cette thèse, ainsi que les contributions principales. Le chapitre est clos par une présentation rapide du contenu des autres chapitres.

1.1 De l'évolution des représentations intermédiaires

Au long des dernières décennies, les langages de programmation ont évolué et se sont éloignés de la représentation binaire de la machine. Ceci rend à la fois les programmes plus portables entre les architectures, plus faciles à écrire, plus sûrs et moins propices aux erreurs ; on fait un meilleur usage du temps du programmeur. On trouve ainsi dans les langages de programmation une multitude d'éléments haut-niveau qui font abstraction des détails de la machine, allant de l'allocation de registres, à des systèmes de types variés, en passant par la gestion automatique de la mémoire ou les fonctions de première classe.

Cependant, ces avantages ont un coût : la tâche du compilateur consistant à transformer le langage source en du code assembleur s'est complexifiée. Le compilateur est devenu une pierre fondamentale et délicate de la programmation d'aujourd'hui. Plusieurs problématiques émergent.

L'absence de bugs dans le compilateur, c'est-à-dire sa correction, devient subtile, demandant des spécifications claires sur les comportements du programme à préserver et beaucoup de minutie à l'heure de l'écriture des transformations. Depuis des décennies, dans les compilateurs industriels, ces spécifications et les invariants à préserver ont reçu un traitement relativement informel ; le lien entre l'implémentation et la spécification est resté

dans la tête du programmeur. Malheureusement, cette approche ne donne pas de garanties suffisantes dans le contexte des systèmes critiques, comme l'avionique ou la médecine. C'est pourquoi récemment, avec l'avènement de la compilation vérifiée, en particulier avec CompCert [46], des méthodes alternatives et plus sûres pour assurer la correction du compilateur et la correspondance entre les spécifications et l'implémentation commencent à faire leur apparition.

Par ailleurs, les langages de haut-niveau laissent de plus en plus la tâche d'optimiser le code généré au compilateur. Tout un panorama de techniques et algorithmes d'optimisations s'est développé. On trouvera donc, par exemple, des algorithmes d'allocation de registres de plus en plus sophistiqués, mais aussi de la propagation de constantes, de l'élimination de sous-expressions communes ou des règles de réécritures pour remplacer certaines séquences d'instructions par d'autres équivalentes mais plus efficaces. Notons que toutes ces optimisations ne font qu'ajouter à la complexité du compilateur et rendent donc la question de la correction d'autant plus importante. Il faut, de plus, prouver non seulement la correction des simples optimisations (transformation d'une représentation intermédiaire vers elle-même), mais également celle des transformations entre différentes représentations intermédiaires. En effet, le compilateur optimisant utilise normalement plusieurs représentations intermédiaires du programme : en général, on retrouve au moins une représentation intermédiaire reflétant de près la structure du programme source et une représentation non structurée plus simple et plus adaptée aux optimisations.

Le choix d'une représentation intermédiaire adaptée pour représenter un programme est déterminant dans le type d'analyses et d'optimisations que l'on peut réaliser, ainsi que leur simplicité et leur efficacité. Ainsi, là où l'arbre de syntaxe abstraite, proche de la représentation haut-niveau du programme et se reposant sur des abstractions éloignées de la machine, est adapté à des analyses haut-niveau sur les types et sur les erreurs de syntaxe ou sémantique, d'autres représentations sont plus adaptées aux optimisations du code généré. Notamment, le graphe de flot de contrôle (CFG) est une représentation qui se prête bien aux analyses de flot de données [2, 42]. Des représentations plus évoluées, souvent dérivées du CFG, sont actuellement utilisées dans les compilateurs industriels. En particulier, des représentations SSA [25] (*Static Single Assignment*) sont utilisées, sous diverses formes, dans la majorité des compilateurs industriels optimisants actuels. On retrouve par exemple des représentations SSA dans GCC [64], LLVM [66], HotSpot [37, 53] ou Go [65]. Ces représentations s'intéressent notamment à simplifier et mettre en évidence les liens de dépendance entre instructions, afin de donner plus de flexibilité aux optimisations lorsqu'elles modifient le code. Ceci est particulièrement flagrant dans la représentation utilisée dans HotSpot et due à Cliff Click [21], qui essaie d'éliminer le plus possible de dépendances superflues entre instructions calculant des

données.

1.2 Vérification et compilation

Durant les deux dernières décennies, la vérification formelle de techniques de compilation réalistes est devenue réalité, inspirée par les travaux effectués sur le compilateur CompCert [46], un compilateur formellement vérifié pour le langage C et qui implémente l'essentiel du standard C90. Un compilateur vérifié se caractérise par la présence d'une preuve formelle, souvent réalisée à l'aide d'un assistant de preuve comme Coq [24], que les comportements du programme sont préservés par la compilation ; en d'autres termes, on a une preuve que la compilation n'introduit pas de bugs. La représentation SSA est arrivée plus tard dans ce panorama avec des efforts indépendants autour de CompCertSSA [4], Vellvm [75, 74], ainsi que les travaux de Blech et al. [8, 7]. Le projet CompCertSSA utilise un validateur complet et vérifié pour la génération du *pruned-SSA*, fondé sur les frontières de dominance [25]. Le projet Vellvm démontre une version simplifiée de la génération SSA de LLVM utilisant de la promotion de variables allouées sur la pile. Blech et al. étudient l'utilisation de graphes de termes, ainsi que d'ensembles relationnels pour représenter la sémantique d'évaluation d'un bloc de base (*basic-block* en anglais) tenant compte des dépendances de données et font le lien avec une représentation séquentielle des instructions du bloc ; ils prouvent formellement également une optimisation d'élimination de code mort. Plus récemment, l'algorithme de génération proposé par Braun et al. [12] a été formalisé en Isabelle/HOL [16].

Beaucoup d'efforts ont donc été menés sur la vérification de la génération de SSA en soi, mais du progrès a aussi été réalisé sur la formalisation d'invariants et propriétés utiles de SSA qui facilitent le raisonnement au moment de prouver des optimisations. En particulier, [74, 4, 28] formalisent l'invariant sémantique de *strictness*, ainsi que du raisonnement équationnel basique et des raisonnements fondés sur la dominance. Ces outils sémantiques permettent par exemple de prouver formellement la correction de la propagation conditionnelle de constantes (SCCP), l'élimination de sous-expressions communes (CSE) à l'aide du *Global Value Numbering* (GVN) [28], ou la propagation de copies et des micro-optimisations relatives à la mémoire [74].

1.3 Contributions

La propriété SSA a donc déjà été étudiée d'un point de vue sémantique sur des représentations en CFG simples, mais le sujet des dépendances entre instructions a été seulement effleuré d'un point de vue formel précédemment. Cette thèse apporte une étude sémantique de transformations

de programmes sous forme SSA en utilisant des représentations qui, non seulement intègrent la propriété fondamentale SSA, mais aussi la flexibilité en termes de dépendances de données entre instructions.

Les contributions de cette thèse sont les suivantes.

- Nous donnons une sémantique formelle pour une représentation *Sea of Nodes*, utilisant une sémantique dénotationnelle pour l'évaluation des instructions correspondant à des calculs de données, mais une sémantique petit-pas pour le flot de contrôle entre régions (blocs) du programme qui, elle, est une extension naturelle de ce que l'on peut retrouver dans CompCertSSA [4] ou Vellvm [74].

- Nous mettons en évidence et prouvons une propriété fondamentale sur les graphes *Sea of Nodes* : pendant l'exécution d'un programme, la valeur calculée par une instruction reste valide dans toutes les régions dominées par le point de définition.

- Nous illustrons cette propriété en prouvant la correction sémantique d'un algorithme efficace de détection et élimination de *zero-checks* redondants qui opère directement sur l'arbre de dominance du graphe *Sea of Nodes*.

- Nous prouvons un algorithme de propagation de constantes sur *Sea of Nodes* qui utilise une analyse creuse de flot de données standard.

- Nous donnons une nouvelle sémantique pour notre forme SSA qui épouse le concept du bloc de base contenant un ensemble non déterministiquement ordonné d'instructions. Nous utilisons cette sémantique pour prouver la correction d'un retour à une représentation plus séquentielle utilisant des blocs de base sous réserve que certaines propriétés de *strictness* soient vérifiées ; c'est-à-dire que chaque instruction est placée dans un bloc particulier en respectant certaines contraintes de dominance.

- Nous détaillons un travail de destruction vérifiée réaliste d'une représentation SSA séquentialisée, intégré dans le compilateur vérifié CompCertSSA.

Les propriétés sémantiques sur la représentation *Sea of Nodes* nécessaires à la preuve de l'optimisation d'élimination de *zero-checks* redondants, ainsi qu'à la preuve de la propagation simple de constantes, ont fait l'objet d'un développement Coq [61, 24], de même que les preuves de préservation sémantique des optimisations elles-mêmes. Les résultats principaux ont fait l'objet d'une publication [27].

La contribution relative à la destruction de SSA a aussi fait l'objet d'un développement Coq, intégré dans le compilateur CompCertSSA développé et prouvé en Coq, ainsi que d'une publication [26].

1.4 Structure du manuscrit

Le chapitre 2 introduit des représentations intermédiaires utilisées en compilation optimisante, en particulier celles essentielles à la compréhension de ce manuscrit, dont *Sea of Nodes*. Le chapitre 3 fournit des éléments essentiels sur la compilation vérifiée, introduisant notamment le concept de simulation. Le chapitre 4 donne une syntaxe et sémantique formelles pour une représentation *Sea of Nodes*. Le chapitre 5 présente la propriété fondamentale sur la préservation des valeurs que l'on utilise dans le chapitre 6. Le chapitre 6 présente et prouve formellement deux optimisations : une élimination de *zero-checks* redondants et un algorithme de propagation de constantes (version simple et conditionnelle). Le chapitre 7 traite du retour à une représentation SSA utilisant des blocs de base, l'étape finale avant la destruction de la forme SSA vers une représentation plus proche de la machine. Le chapitre 8 traite d'une destruction de SSA réaliste, formellement prouvée et intégrée dans CompCertSSA. Le chapitre 9 fait l'état des perspectives pour des travaux futurs et conclut le manuscrit.

Chapitre 2

Contexte : représentations intermédiaires

Dans ce chapitre, nous donnons une introduction à différentes représentations intermédiaires utilisées pour de la compilation optimisante et nous nous focalisons, en particulier, sur celles qui sont essentielles à la compréhension de ce manuscrit. La plupart des résultats présentés dans ce chapitre se retrouvent dans les livres classiques de compilation [1, 3], sauf référence plus explicite.

2.1 Introduction

La première représentation intermédiaire que l'on retrouve dans un compilateur est l'arbre de syntaxe abstraite (AST) obtenu juste après la phase d'analyse syntaxique. Cette représentation est proche de la représentation haut-niveau du programme. Elle se prête donc bien à certaines analyses, comme l'analyse de types, ainsi qu'à la production de messages d'erreur de compilation utiles à l'utilisateur. L'AST n'est, par contre, pas adapté à l'optimisation du code généré par le compilateur.

En effet, un premier exemple qui peut venir à l'esprit est celui d'optimisations comme l'élimination de sous-expressions communes (CSE) : la nature de la représentation sous forme d'arbre empêche intrinsèquement le partage d'expressions communes. Une première solution est d'utiliser plutôt un graphe acyclique (DAG), ce qui permet de partager des sous-arbres.

En fait, un certain nombre d'optimisations couramment utilisées dans les compilateurs modernes sont fondées sur le concept d'analyse de flot de données [42]. On trouve dans cette catégorie par exemple des optimisations comme la propagation de constantes, ainsi que des variantes avancées de l'élimination de sous-expressions communes, ou l'élimination de code redondant. Ces analyses reposent sur une représentation du programme sous forme de graphe de flot de contrôle (CFG) qui, comme son nom l'indique, représente le programme sous forme d'un graphe dans lequel le flot de

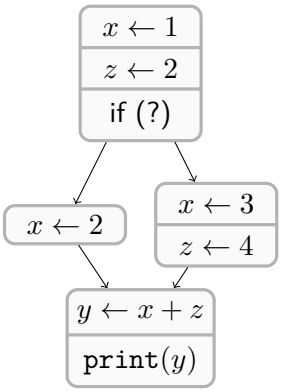


FIG. 2.1 : Exemple de CFG

contrôle est explicite : dans le cas le plus simple, on a un graphe orienté dont les nœuds sont des instructions, et où chaque instruction a connaissance de ses successeurs immédiats dans le CFG via des arcs. Dans un CFG il y a en général un nœud qui est le point d'entrée (première instruction exécutée) et potentiellement plusieurs nœuds de sortie (fin d'exécution).

La figure 2.1 montre un exemple de CFG. L'instruction $x \leftarrow 1$ est suivie d'une instruction $z \leftarrow 2$, puis d'une instruction de branchement (if), qui a deux successeurs possibles : lequel sera pris dépend du résultat de la condition du if.

Les compilateurs transforment donc l'AST en une représentation intermédiaire non structurée à base de flot de contrôle afin de pouvoir effectuer des optimisations plus poussées. Au passage, les compilateurs simplifient aussi la sémantique du langage en simplifiant les informations haut-niveau qui ne sont plus nécessaires à ce stade de la compilation (par exemple des informations concernant les types).

Dans la suite du chapitre, nous décrivons plusieurs représentations intermédiaires dérivées du CFG, ainsi que des propriétés complémentaires, utilisées en compilation optimisante.

2.2 Bloc de base et dépendances de données

Une remarque qu'il est naturel de se faire par rapport au CFG, c'est que la plupart des instructions ne contribuent pas vraiment au flot de contrôle : uniquement les instructions correspondant à des sauts (conditionnels ou non) sont essentielles pour le décrire. Dans l'exemple de la figure 2.1, on voit que certaines instructions, comme les trois premières, ont été regroupées, collées à la suite, laissant implicites les flèches. En fait, ceci correspond à la notion basique de bloc de base (*basic-block* en anglais) comme séquence $i_1, \ldots, i_m$ finie d'instructions qui vérifient les propriétés suivantes :

- uniquement i_1 peut avoir plus d'un prédécesseur ;

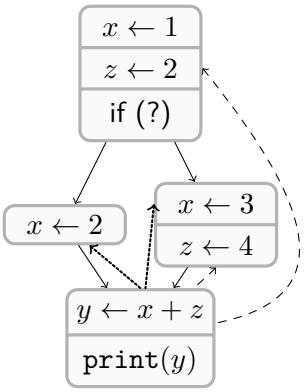


FIG. 2.2 : Dépendances de données

- uniquement i_m peut avoir plus d'un successeur ;
- pour $k = 1, \ldots, m-1$, chaque instruction i_k a comme successeur i_{k+1}.

En allant un peu plus loin, la notion de bloc de base permet aussi de mettre en évidence que, non seulement les instructions du bloc de base ne contribuent que trivialement au flot de contrôle, mais en fait la rigidité du flot de contrôle entre elles (une séquence) peut en fait être rendue plus flexible, car des instructions indépendantes entre elles pourraient être réordonnées sans changer la sémantique du programme. C'est par exemple le cas, dans l'exemple de la figure 2.1, des instructions $x \leftarrow 1$ et $z \leftarrow 2$, qui peuvent être exécutées dans n'importe quel ordre. Par contre, dans le bloc du bas, le `print` doit être exécuté après la définition de y. L'instruction `print`(y) *dépend* de l'instruction $y \leftarrow x + z$.

Plus formellement, Ferrante et al. [32] disent qu'une instruction i' a une *dépendance de données* sur i s'il existe une variable x utilisée en i' et définie en i, et qu'il existe un chemin dans le CFG de i vers i' sans redéfinitions de x.

La figure 2.2 montre les dépendances de données dans l'exemple précédent pour l'instruction $y \leftarrow x + z$. Les flèches en pointillés correspondent à des dépendances induites par l'utilisation de x, les flèches avec des tirets à des dépendances induites par l'utilisation de z. Par exemple, pour x, on voit que, suivant la branche prise après le `if( ?)`, une définition différente de x atteint l'utilisation, donc deux dépendances sont générées pour $y \leftarrow x + z$ par l'utilisation de x. De manière analogue, z induit deux dépendances aussi.

Cette notion de dépendances de données permet de représenter les instructions d'un bloc en termes de dépendances, plutôt que séquentiellement, ce qui offre plus de flexibilité aux optimisations pour réordonnancer. Par exemple, le premier bloc, au lieu d'être représenté comme la séquence

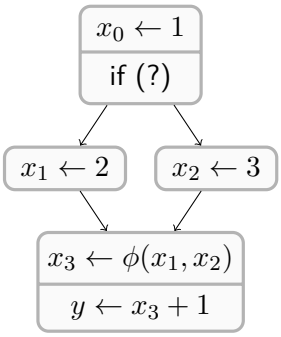


FIG. 2.3 : Exemple de programme SSA

$x \leftarrow 1$, $z \leftarrow 2$, `if( ?)`, pourra être représenté comme un ensemble de trois instructions, dont la dernière est fixée, le `if( ?)`, l'instruction qui contrôle la sortie du bloc, et qui dépend potentiellement des deux premières, mais les deux premières n'ont pas besoin d'arc entre elles (la seule contrainte est qu'elles appartiennent au bloc).

Ce genre de représentation tenant compte des dépendances de données à l'intérieur d'un bloc est par exemple utile pour réordonner des instructions mémoire comme des *loads* de sorte à optimiser l'utilisation du *pipeline* dans le processeur [35]. De manière plus générale, cela simplifie le raisonnement pour prouver la correction d'optimisations qui peuvent modifier l'ordre des instructions dans le bloc. En effet, certains programmes sémantiquement équivalents ayant la même représentation syntaxique en termes de dépendances de données peuvent avoir plusieurs représentations séquentielles dont l'équivalence sémantique n'est pas évidente.

2.3 SSA

La forme SSA [25] (*Static Single Assignment*) est une propriété d'une représentation de programme dans laquelle chaque variable est définie statiquement en un point unique du programme. Afin de pouvoir obtenir cette propriété, des instructions spéciales situées aux points de jonction du programme, appelées les ϕ-instructions, de la forme $x \leftarrow \phi(x_1, \ldots, x_n)$, sélectionnent, à l'exécution et en fonction du chemin d'exécution suivi, une variable parmi les x_i, dont la valeur est ensuite affectée à x. Les ϕ-instructions du bloc sont exécutées en parallèle et avant les autres instructions du bloc.

Par exemple, dans la figure 2.3, chaque variable x_i est définie en un seul point. Au point de jonction, une ϕ-instruction choisit, suivant que l'exécution provient du premier ou du deuxième prédécesseur, la valeur de x_1 ou x_2 et l'affecte à x_3.

On remarque que, sous cette forme, les dépendances de données entre

instructions deviennent des dépendances entre variables : à chaque instruction correspond une unique variable. La définition de dépendance de données devient plus simple, car il n'y a pas de redéfinition de variable : l'utilisation d'une variable dans une instruction donnée ne génère qu'une seule dépendance de donnée, contrairement à l'exemple de la figure 2.2 vu à la section précédente.

2.3.1 Insensibilité au flot

Sous forme SSA, grâce à la propriété de définition unique, certaines analyses de flot de données utiles dans des optimisations, comme pour la propagation de constantes [71], deviennent insensibles au flot de contrôle. Ceci les rend plus simples à implémenter et plus efficaces. En effet, au lieu d'attacher en tout point de programme une information pour chaque variable, on n'a besoin de stocker qu'une seule information par variable pour tout le graphe. Par exemple, dans le cas de la propagation de constantes, comme il n'y a pas de redéfinition d'une variable du fait du point de définition unique, si une variable représente une constante au moment de sa définition, elle représente toujours la même constante dans le reste du programme : il suffit donc de s'intéresser à la valeur d'une variable au point de définition.

D'autres analyses, comme l'analyse de *liveness*, restent cependant sensibles au flot de contrôle.

2.3.2 Chaînes *use-def*

Une autre conséquence de la propriété SSA est la simplification de ce que l'on appelle les chaînes *use-def* qui font la correspondance entre les points de définition d'une variable et ses utilisations. Elles sont dans le cas général définies comme suit :

Définition 2.3.1. En chaque point d'utilisation d'une variable x, on définit une chaîne *use-def* correspondante par la liste des nœuds des définitions de x atteignantes.

En effet, pour un CFG qui n'est pas sous forme SSA, un point donné pour une variable donnée, plusieurs définitions doivent potentiellement être prises en compte. La figure 2.4 illustre en plusieurs points d'utilisation d'une variable x les différentes définitions qui peuvent être en vigueur au moment de l'utilisation (flèches avec des tirets ou en pointillés de l'utilisation vers les définitions). Comme on le voit, dans le cas de l'instruction $z \leftarrow 1 + x$, on a deux définitions atteignantes $x \leftarrow 42$ et $x \leftarrow 1 + x$. Ceci signifie qu'une chaîne *use-def* est nécessaire par utilisation de variable, mais aussi qu'une chaîne peut avoir plus d'un élément.

Sous forme SSA, on a uniquement une chaîne *use-def* par variable, plutôt que par point du programme, et la chaîne est réduite à un seul élément :

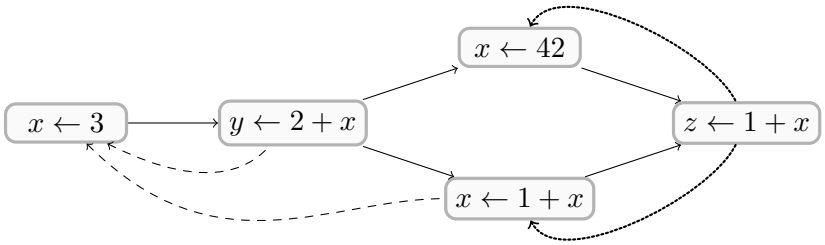


FIG. 2.4 : Exemple de chaînes *use-def*

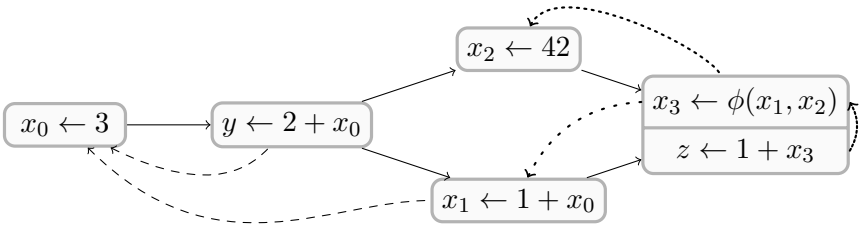


FIG. 2.5 : Exemple d'arcs SSA

l'unique nœud de définition de la variable. En particulier, cela signifie qu'il est facile d'intégrer ces chaînes dans la représentation intermédiaire elle-même sous la forme de ce que l'on appelle les *arcs SSA*. La figure 2.5 illustre les arcs SSA sur une version du programme précédent mise sous forme SSA. Pour plus de clarté, des styles différents (plus ou moins de pointillés ou tirets) sont, comme précédemment, utilisés en fonction de la variable créant l'arc.

Ce sont les arcs SSA qui rendent des optimisations comme la propagation de constantes insensibles au flot de contrôle. On trouvera une analyse détaillée pour la propagation de constantes à la section 6.2.

Dominance et lemme équationnel. La notion de dominance est particulièrement utile, sous forme SSA, pour décrire des zones du programme où certaines informations sur les valeurs sont vraies.

Définition 2.3.2. Un nœud n du CFG *domine* un autre nœud du graphe n' si tout chemin du nœud point d'entrée jusqu'à n' passe par n. La dominance est dite *stricte* si $n \neq n'$.

En particulier, une propriété utile pour raisonner sur la valeur des variables, formulée en termes de dominance, est le dit lemme équationnel [4].

Lemme 2.3.3. *L'équation définissant une variable x au point de programme n est valide en tout point strictement dominé par n.*

La notion d'équation définissant une variable correspond à l'égalité de l'évaluation des membres gauche et droit d'une affectation avec les valeurs des variables dans l'environnement au point courant. Par exemple, si x est défini par $x \leftarrow 3$ en un nœud n, alors en tout point strictement dominé par n, on sait que $x = 3$. Dans le cas d'une affectation $x \leftarrow y + 1$ en un nœud n, l'environnement donne des valeurs à x et y telles que $x = y + 1$ en tout point domminé par n ; par contre, par exemple dans le cas d'une boucle, l'exécution peut repasser par la définition de y et se retrouver dans une zone non dominée par n, auquel cas l'équation n'est plus valable, puisque l'environnement donne encore à x la valeur obtenue à partir de la valeur précédente de y.

SSA et blocs de base. SSA est en soi une propriété d'une représentation et celle-ci dispose de flexibilité pour représenter un programme qui la respecte. Par exemple, dans CompCertSSA [22], la représentation utilisée est en fait un simple CFG avec des instructions et des variables, tel que mentionné en fin de section 2.1. Une telle approche dans CompCertSSA a rendu son intégration dans la chaîne de CompCert plus aisée, puisque plus proche du langage RTL de CompCert (un simple CFG d'instructions avec des variables). Il est cependant possible de représenter la forme SSA à l'aide d'une représentation utilisant des blocs de base et éliminant la notion de variable, confondant un nœud ou instruction avec la variable qu'il définit. C'est l'approche que l'on retrouve par exemple dans LLVM ou Go.

2.4 *Sea of Nodes*

La représentation qui va nous occuper le plus dans ce manuscrit, surnommée *Sea of Nodes* et introduite par Cliff Click dans sa thèse [20], est une continuation des idées décrites dans les deux sections précédentes. Elle est utilisée dans le compilateur HotSpot [37, 53] pour Java, ainsi que dans le projet LibFirm [13]. Le projet Graal [29] utilise une variante de la même représentation qui met l'accent sur les optimisations spéculatives dans un compilateur dynamique pour Java.

Sea of Nodes est une représentation SSA qui pousse le concept des dépendances de données plus loin en se libérant de la contrainte qui veut que toute instruction appartienne à un bloc de base ou, plus généralement, à un point du CFG. Chaque instruction calculant une valeur est un nœud du graphe dont les entrées sont d'autres nœuds correspondant à des instructions calculant des données. Par exemple, dans le pseudo code

```
1 if (b) {
2     x ← - a
```

```
3       y ← c + 1
4   }
```

l'instruction $y \leftarrow c + 1$ va être représentée par un nœud flottant correspondant à une instruction d'addition, avec seulement deux entrées venant des nœuds correspondant à la définition d'une variable c et de la constante 1 (des dépendances de données). On confond donc le nœud représentant une instruction et la variable que l'instruction définit, mais, de plus, on oublie (sauf pour les ϕs) l'appartenance à un bloc particulier. Le branchement conditionnel est représenté par un nœud conditionnel avec deux successeurs correspondant au cas `true` et `false`, appelés projections, du fait que l'on projette sur une branche ou l'autre. On remarque que l'ordre dans lequel on définit x et y reste libre. C'est seulement plus tard dans la chaîne de compilation qu'un ordre complet va être décidé sur les instructions, d'abord en les affectant à un bloc particulier, puis en les ordonnant à l'intérieur d'un bloc.

En fait, la représentation *Sea of Nodes* représente les blocs eux-mêmes par des nœuds, appelés nœuds de région, qui seront connectés par des dépendances aux nœuds traitant du flot de contrôle, comme par exemple des sauts, conditionnels ou non, ainsi que des retours de fonction. Ceci simplifie l'écriture de certaines analyses de flot de données, comme la propagation de constantes conditionnelle. Cette dernière profite aussi des nœuds de *projection* pour la conditionnelle, qui permettent d'éviter d'avoir à attacher de façon ad hoc l'information sur la branche prise au niveau des arcs. Nous verrons aussi que la nature flottante des nœuds de données permet d'écrire une sémantique d'évaluation des nœuds de données avec laquelle il est commode de raisonner. De plus, en introduisant moins de dépendances superflues, certains programmes sémantiquement équivalents, mais syntaxiquement différents avec une représentation plus séquentielle, deviennent syntaxiquement équivalents : la syntaxe épouse mieux la sémantique. Il s'agit là d'une propriété intéressante pour une approche formelle. Ces sujets sont traités dans les chapitres 4, 5 et leur application à la preuve d'optimisations est abordée dans le chapitre 6.

La représentation *Sea of Nodes*, adaptée pour raisonner sur les optimisations, s'éloigne d'une représentation séquentielle des programmes : à un moment, il faut revenir sur une représentation séquentielle et, donc, comme mentionné plus haut, affecter une région à tous les nœuds flottants, puis reconstruire des blocs de base avec des instructions ordonnées. La première étape, donner à chaque nœud de données une région, par exemple à l'aide de l'algorithme de *Global Code Motion* dû à Click [19], est le sujet principal du chapitre 7. Une des forces de la représentation *Sea of Nodes* est de séparer proprement les phases d'optimisation du *Global Code Motion* : d'une part, cela simplifie l'implémentation et, d'autre part, cela facilite le raisonnement et d'autant plus d'un point de vue formel. La séquentialisation ultérieure des instructions à l'intérieur d'un bloc est mentionée brièvement en fin du

chapitre 7.

2.5 Variantes de SSA

Il existe des variantes et extensions de SSA dont l'objectif est en général de faciliter certaines analyses plus spécifiques. On passe en revue quelques représentations classiques dans cette section.

En particulier, nous donnons des exemples de représentations comme la forme GSA ou le VSDG, qui font des choix spéciaux au niveau des nœuds et dépendances de contrôle (boucles et conditionnelles), tout en ayant une approche analogue à *Sea of Nodes* pour ce qui est des dépendances de données.

Nous donnons aussi deux exemples d'extensions (SSI, *Array* SSA) qui sont essentiellement indépendantes de la méthode choisie pour représenter SSA et donc pourraient a priori être intégrées naturellement dans une représentation inspirée de *Sea of Nodes*.

2.5.1 *Program Dependence Graph*

Le *Program Dependence Graph* [32] (PDG) est une représentation, pas nécessairement SSA, sous forme de graphe qui utilise deux types de dépendances. Les dépendances de données correspondent essentiellement à celles que l'on trouve dans *Sea of Nodes*. Le graphe fait également usage de dépendances de contrôle au sens suivant : un nœud n' a une dépendance de contrôle sur un nœud n si l'exécution ou non de n' se décide en n (une formulation plus formelle s'exprime en termes de post-dominance). Par exemple, si n a deux branches successeurs (dans le CFG), et que n' se trouve sur l'une des branches mais pas sur l'autre, on a une telle dépendance. Remarquons que ces dépendances de contrôle ne correspondent pas à celles du CFG ; en particulier, un nœud avec un seul successeur n'induit pas de dépendance de contrôle sur son successeur. Notons que le PDG à lui seul ne contient pas l'information suffisante au niveau des points de jonction pour obtenir un modèle d'exécution. Le PDG facilite certaines optimisations comme la détection de parallélisme, ainsi que la modification incrémentale des dépendances de données suite à un déroulage de boucle ou la suppression d'une branche.

2.5.2 *Gated Single Assignment*

La forme *Gated Single Assignment* [34] (GSA) est une extension de SSA qui permet d'interpréter les ϕ-instructions indépendamment du flot de contrôle. L'information spécifiant quelle variable choisir parmi les variables d'une ϕ-instruction est ajoutée à la ϕ-instruction en entrée. Cette information est un prédicat, correspondant à une condition calculée lors

d'un branchement conditionnel, et permet de choisir entre deux variables. La représentation utilise des ϕ-instructions spéciales pour les entrées, avec une entrée pour l'initialisation et une autre pour les itérations, des ϕ-instructions en sortie de boucle avec un prédicat permettant de déterminer la valeur de sortie d'une variable, et des ϕ-instructions avec prédicat qui permettent de simuler la logique d'une expression `if-then-else`. Notons que seuls les programmes dont le graphe de flot de contrôle est réductible peuvent être représentés sous forme GSA. La forme est intéressante, puisqu'elle donne plus d'information au compilateur sur la variable qui sera choisie par les ϕ-instructions. Elle peut également permettre de faciliter la génération de code, du fait de la sémantique exécutable de ses ϕ-instructions étendues.

2.5.3 *Value State Dependence Graph*

Le *Value State Dependence Graph* [31, 39] (VSDG), reprend des idées du GSA. Il s'agit d'un graphe de dépendances utilisant des nœuds spéciaux pour les boucles (nœuds θ) et les conditionnelles (nœuds γ). Tout comme pour le GSA, seulement des programmes dont le flot de contrôle est réductible peuvent être représentés. Deux types de dépendances sont utilisées, les dépendances de valeurs (correspondant aux dépendances de données) et des dépendances d'état — ces dernières correspondent essentiellement au flot de contrôle, avec la particularité que la forme prévoit l'ajout de dépendances de contrôle additionnelles à des fins de séquentialisation.

Une des particularités de cette forme est que deux nœuds γ peuvent être combinés lorsque la condition est la même, en utilisant un couple de valeurs pour chaque arc, afin de regrouper les deux conditionnelles `if-then-else` en une seule. Les nœuds $\theta(C, I, R, L, X)$ combinent les nœuds d'entrée et de sortie de boucle dans GSA, avec une condition C, une valeur initiale I, des valeurs courantes L et d'itération R, et une valeur de retour X.

Cette représentation rend certaines optimisations intrinsèques, comme l'élimination de code mort, qui correspond aux nœuds pour lesquels il n'y a pas de chemin de dépendances depuis le nœud de retour de la fonction.

2.5.4 Thorin

La représentation intermédiaire Thorin [44] est une représentation intermédiaire fonctionnelle d'ordre supérieur, en particulier intéressante pour des transformations mettant en jeu des fonctions d'ordre supérieur et autres concepts fonctionnels comme la récursivité terminale. La représentation reprend les idées de la représentation fonctionnelle CPS [41] (*Continuation Passing Style*) qui est en étroite correspondance avec SSA. Un programme Thorin est donc un ensemble de fonctions, qui introduisent des paramètres et consistent en un seul appel de fonction, sans valeur de retour. Thorin

se distingue de la forme CPS par l'utilisation d'une représentation sous forme de graphe de nœuds et faisant l'usage de dépendances, à la *Sea of Nodes*, plutôt que d'imbrication de blocs lexicaux, pour associer l'utilisation d'une valeur à sa définition. Ceci rend, en particulier, la représentation intéressante non seulement pour les langages fonctionnels, mais également pour les langages offrant la possibilité d'écrire des programmes impératifs et fonctionnels (comme Java, Go, etc.).

2.5.5 *Static Single Information*

La forme *Static Single Information* [60] (SSI) utilise, en plus des ϕ-instructions, des fonctions spéciales σ au niveau des branchements, correspondant à des renommages additionnels sur des variables dont on veut conserver de l'information. En particulier, ces fonctions permettent de conserver de l'information additionnelle fournie par un branchement conditionnel. Par exemple, si un branchement conditionnel se fait sur la condition $x = 0$, un renommage en deux variables x_1 et x_2 sur chacune des branches permet d'accéder facilement à l'information que la condition donne sur la valeur de x sur chacune de celles-ci.

2.5.6 *Array* SSA

La forme *Array* SSA [43] est une extension de SSA qui permet de gérer les alias entre éléments d'un tableau. La technique utilisée est de renommer les variables de tableau de sorte à ce que la propriété de définition unique SSA soit vérifiée pour les éléments des tableaux également, en introduisant des ϕ-instructions étendues pour les variables de tableau afin de tenir compte des versions successives issues des modifications des éléments. Notons qu'une différence par rapport aux ϕ-instructions classiques, qui fusionnent des variables, est que d'autres types de ϕ-instructions, dites de définition, sont introduites également après chaque modification d'un élément, afin de représenter la fusion élément par élément des tableaux. Par exemple, si X_0 est un tableau, on introduit une variable X_1 lors d'une modification d'un élément du tableau $X_1[i] \leftarrow \cdots$, puis une variable X_2 définie par $X_2 = \phi(X_0, X_1)$, représentant la fusion des deux tableaux (élément modifié et éléments non modifiés), de sorte qu'à chaque définition une variable différente est utilisée.

Un des intérêts de cette forme réside dans les possibilités de parallélisation automatique.

2.6 Conclusion

Ce chapitre nous a donc permis de faire un panorama sur des représentations intermédiaires utilisées en optimisations. En particulier, nous

avons vu comment la notion de dépendance de données permet de faire une transition entre le graphe de flot de contrôle et des représentations plus flexibles comme *Sea of Nodes*, qui est celle que nous étudions formellement dans ce manuscrit.

Le chapitre suivant fournit les notions fondamentales en compilation vérifiée permettant d'appréhender le travail mené dans le reste du manuscrit.

Chapitre 3

Contexte : compilation vérifiée

Ce chapitre fournit quelques bases de compilation vérifiée utiles dans le reste du manuscrit. Nous passons également en revue des travaux existants en compilation vérifiée.

3.1 Introduction

Comme nous l'avons mentionné en introduction, les compilateurs pour les langages d'aujourd'hui sont devenus des programmes conséquents, utilisant des analyses statiques complexes, en particulier pour effectuer des optimisations afin de générer du code plus efficace. Il est donc difficile d'écrire un compilateur correct, sans bugs et qui n'introduit donc pas de comportements non voulus dans le code généré. Yang et al. [72] ont en particulier montré que ces phases d'optimisations sont source de bugs dans les compilateurs industriels comme GCC ou LLVM. Ceci est en particulier problématique dans le contexte des systèmes critiques, comme dans l'avionique ou les logiciels utilisés pour des opérations médicales. En effet, dans ces systèmes-là, il est primordial que le code généré soit fidèle au programme source initial et sa spécification. En particulier, les propriétés de sûreté prouvées sur le code source, par exemple à l'aide d'analyses statiques, doivent être préservées par le compilateur. Il semble donc indispensable d'essayer d'obtenir des garanties fortes sur la correction du compilateur. Une façon d'obtenir de telles garanties est d'avoir une preuve formelle et mécanique de cette correction.

3.2 Compilateurs vérifiés

Prouver la correction d'un compilateur, c'est prouver qu'il préserve le comportement des programmes dont la sémantique est bien définie, c'est-à-dire sans comportement non définis, comme une division par zéro, un déréférence de pointeur nul ou un dépassement d'entier signé en C.

Plus précisément, il s'agit donc de définir formellement la sémantique des programmes, ainsi que leurs comportements, puis de prouver que les transformations préservent ces comportements vis-à-vis des sémantiques des langages source et cible.

Après plusieurs décennies d'histoire, le domaine de la compilation voit surgir un nouveau type de compilateur : des compilateurs réalistes vérifiés mécaniquement. Le premier exemple d'un tel compilateur, et le plus complet, est CompCert [46], développé et vérifié à l'aide de l'assistant de preuve Coq. Il s'agit d'un compilateur vérifié qui génère du PowerPC, ARM ou x86 (32-bits et 64-bits) pour des programmes source écrits dans un large sous-ensemble du langage C (contenant l'essentiel du standard C90). CompCert formalise la sémantique opérationnelle d'une douzaine de langages intermédiaires et prouve un théorème de préservation sémantique pour chaque phase. Chaque langage intermédiaire est doté d'une sémantique formelle qui décrit l'exécution du programme. Ces sémantiques sont décrites par des systèmes de transitions entre états d'exécution que l'on représente naturellement en Coq par un type inductif.

La présence de tant de langages intermédiaires est due au fait qu'en compilation vérifiée, il est plus pratique de raisonner sur un petit nombre de changements à la fois. Par exemple, dans CompCert, une des premières transformations rend le langage déterministe, c'est-à-dire, en gros, que le langage n'autorise qu'un seul comportement observable possible pour l'exécution d'un programme. Ensuite, d'autres transformations simplifient petit à petit le langage. Initialement, on part d'une forme AST très proche de la structure et sémantique du langage C lui-même. Cette représentation AST est progressivement simplifiée, par exemple de sorte à simplifier le flot de contrôle (un seul type de boucles), ou afin de se libérer d'une partie des informations de typage du langage C. Ensuite, on passe à une représentation non structurée du code sous forme de CFG d'instructions avec des registres, appelée RTL, où ont lieu la plupart des optimisations de code. Plus loin dans la chaîne de compilation, des transformations vers des langages intermédiaires plus proches de l'assembleur et plus dépendants de l'architecture ont lieu, de l'allocation de registres jusqu'à la génération de code elle-même.

3.3 Préservation sémantique

Les théorèmes de préservation sont exprimés en termes de comportement de programmes, c'est-à-dire des traces finies ou infinies d'évènements observables (essentiellement des appels systèmes ou autres fonctions considérées externes) qui sont réalisés pendant l'exécution du programme. Les théorèmes établissent que chaque phase de compilation préserve ces comportements. Une conséquence des théorèmes est que pour tout programme P (dans un langage L) dont la sémantique est bien définie, si le programme

est compilé vers un programme P' (dans un langage L'), alors tous les comportements de P' sont des comportements possibles de P.

En pratique, afin de prouver un telle propriété de préservation sémantique (appelée une *simulation arrière*), on prouve souvent plutôt une *simulation avant* qui, sous certaines hypothèses standard sur les langages (essentiellement le déterminisme du langage cible), est équivalente mais plus facile à prouver. Une simulation avant prouve que si P a une sémantique bien définie, alors tous les comportements de P sont des comportements possibles de P'.

Nous rappelons que les compilateurs vérifiés assurent cette préservation de comportements seulement lorsque la sémantique du programme initial est bien définie. Ce dernier point doit être prouvé en amont par des analyses statiques sur le code source du logiciel critique que l'on veut compiler.

3.3.1 Techniques de compilation vérifiée

Pour prouver qu'une transformation préserve la sémantique, plusieurs approches sont possibles.

La première, c'est d'écrire l'algorithme dans le langage d'un assistant de preuve comme Coq, puis de prouver directement sur cet algorithme les propriétés nécessaires à la préservation sémantique : si l'algorithme produit un résultat, alors il est sémantiquement équivalent.

Une deuxième approche, assez populaire et que nous utilisons parfois dans ce manuscrit, est celle de la validation a posteriori par validateur formellement prouvé [54, 50, 70, 69]. Dans ce cas là, l'algorithme de la transformation n'est pas prouvé, mais le résultat est validé par un validateur prouvé : si le validateur accepte le résultat, c'est que la sémantique a été préservée.

Une troisième approche parfois utilisée dans le domaine de la compilation vérifiée est celle du *proof-carrying code* [49]. Cette approche n'établit pas un théorème de préservation sémantique entre programme source et transformé, mais se contente de vérifier, a posteriori, certaines propriétés de sûreté spécifiques — par exemple de typage — sur le code transformé, relatives à une certaine spécification.

3.3.2 Simulation *lock-step.*

Comme mentionné précédemment, sous certaines hypothèses, montrer une simulation avant revient à prouver la préservation sémantique. Établir une telle simulation peut être fait en exhibant une relation $\sim$ entre états d'exécution des programmes source et cible qui porte tous les invariants nécessaires pour prouver la préservation de comportement. Dans ce manuscrit, dans les simulations que nous considérons, les programmes source et cible ont des états d'exécution qui correspondent après seulement un

pas dans la sémantique, alors que d'autres phases de compilation, non traitées dans ce manuscrit, remplacent une instruction haut-niveau par une séquence d'instructions. On note $\sigma \xrightarrow{t}_P \sigma'$ un pas entre deux états σ et σ' avec un comportement observable (trace) t pour un programme P. Pour rappel, le système de transitions qui décrit les pas valides entre deux états d'exécution, c'est la sémantique du langage. Il sera fait usage du schéma suivant de simulation, dit *lock-step* :

Lemme 3.3.1. *Soit $\sim$ une relation entre les états d'exécution des programmes source et cible satisfaisant ces propriétés :*

- *pour tout état initial σ_1 de P, il existe un état initial σ_2 de P' tel que $\sigma_1 \sim \sigma_2$;*
- *si $\sigma_1 \xrightarrow{t}_P \sigma_2$ et $\sigma_1 \sim \sigma_1'$, alors il existe un état σ_2' tel que $\sigma_1' \xrightarrow{t}_{P'} \sigma_2'$, et $\sigma_2 \sim \sigma_2'$;*
- *si $\sigma_1 \sim \sigma_2$ et σ_1 est un état final de P, alors σ_2 est un état final de P'.*

Alors tous les comportements de P sont aussi des comportements de P'.

En particulier dans les sections 8.3 et 8.4, nous utilisons ce lemme pour prouver la correction sémantique de la destruction SSA. Pour des raisons d'économie de présentation, nous traitons en détail uniquement la deuxième propriété qui est la plus intéressante. Notons qu'il s'agit de montrer une simple implication.

3.4 Travaux en compilation optimisante vérifiée

Dans cette section, nous faisons un tour un peu plus détaillé des travaux connus autour de la compilation vérifiée optimisante.

3.4.1 CompCert

CompCert dispose au niveau du langage intermédiaire RTL – un simple CFG d'instructions — de plusieurs optimisations, dont de la propagation de constantes, élimination d'appels terminaux récursifs, *inlining*, élimination simple de sous-expressions communes et élimination de code mort. En particulier, des techniques de preuve générales pour les analyses de flot de données dans RTL ont été étudiées [5]. Cependant, CompCert n'utilise pas de raisonnements liés à la dominance et, notamment, n'utilise pas la forme SSA.

Par ailleurs, CompCert intègre un algorithme efficace d'allocation de registres vérifié a posteriori [56].

Des travaux sur des optimisations de type *peephole* [47] ont également été effectués dans le projet Peek [48]. Il s'agit d'optimisations à un niveau assembleur pour lesquelles un ensemble de propriétés locales sur la transformation permet d'assurer la correction globale de la transformation. En pratique, l'idée est de remplacer certains motifs récurrents de séquences d'instructions par d'autres séquences d'instructions équivalentes et plus efficaces. Bien que locales, ces optimisations requièrent néanmoins des analyses de *liveness* sur les registres, afin de savoir quand est-ce que la valeur d'un registre doit être préservée ou pas.

3.4.2 CompCertSSA

CompCertSSA intègre un langage intermédiaire ressemblant à RTL dans CompCert, mais vérifiant la propriété SSA. Ceci permet de réaliser des optimisations plus avancées [28], dont de la propagation conditionnelle de constantes creuse (SCCP) et de l'élimination de sous-expressions communes fondée sur une analyse de *Global Value Numbering* (GVN). Ces optimisations utilisent un même cadriciel adapté à la preuve d'optimisations creuses (*sparse*). Les résultats expérimentaux donnent une meilleure précision et de meilleurs temps de compilation que les techniques vérifiées précédentes au niveau RTL dans CompCert pour les optimisations analogues pour la propagation de constantes et l'élimination de sous-expressions communes. La propagation de constantes, en particulier, a été mesurée comme étant deux à trois fois plus rapide.

3.4.3 Vellvm

Le projet Vellvm [75, 74] est un cadriciel, développé et prouvé en Coq, qui permet de raisonner sur des programmes exprimés dans la représentation intermédiaire de LLVM [66]. Les auteurs du projet donnent plusieurs sémantiques opérationnelles pour la représentation de LLVM. On trouve en particulier un sémantique petit-pas similaire à celle de CompCertSSA et une sémantique grand-pas évaluant en un pas une fonction qu'ils affirment être plus pratique pour prouver certaines optimisations par validation a posteriori. De plus, de façon analogue à CompCertSSA, ils spécifient également la propriété de *strictness* selon laquelle la définition d'une variable domine ses utilisations, ainsi que certaines propriétés de base sur la dominance.

Ils prouvent également une version simplifiée de la transformation `mem2reg` de LLVM qui consiste à promouvoir des variables allouées dans la pile en des variables SSA. C'est l'analogue de la génération SSA, mais adaptée au cas de la représentation intermédiaire de LLVM. En effet, les compilateurs utilisant LLVM peuvent se contenter de générer du code pour la représentation intermédiaire de LLVM sans avoir besoin d'introduire de ϕ-instructions, en faisant plutôt usage de la pile pour les variables locales.

Cependant, afin que les optimisations SSA soient efficaces et capables de raisonner convenablement sur les variables locales, il est nécessaire de transformer ensuite ces variables mémoire en variables SSA.

3.4.4 Bloc de base et dépendances de données

Blech et al. [8, 7] proposent deux approches pour représenter l'évaluation des instructions d'un bloc de base en tenant compte des dépendances de données. Leur travail a fait l'objet d'un développement à l'aide de l'assistant de preuve Isabelle/HOL [38].

La première approche représente les instructions d'un bloc à l'aide d'un ensemble d'arbres obtenus en dupliquant les nœuds partagés dans le graphe de termes acyclique formé par les dépendances entre instructions du bloc. Ils utilisent cette première approche pour prouver une optimisation d'élimination de code mort.

Leur deuxième approche utilise les dépendances de données pour voir les nœuds de données d'un bloc comme un ensemble partiellement ordonné. La relation d'ordre partiel détermine si une instruction doit être évaluée avant une autre, ou si l'ordre est libre.

Dans les deux cas, leur sémantique procède bloc par bloc, en évaluant d'abord les ϕ-instructions, puis ensuite les autres instructions d'un bloc et, enfin, l'instruction qui contrôle la sortie du bloc vers le bloc suivant. Ceci est à mettre en contraste avec l'approche de CompCertSSA qui exécute les ϕ-instructions d'un point de jonction lors du traitement du prédécesseur afin de simplifier la sémantique (voir section 8.2.2).

Ils montrent également, pour les deux approches, une équivalence sémantique entre leur sémantique tenant compte de façon flexible des dépendances de données à l'intérieur d'un bloc et une sémantique séquentielle pour les instructions du bloc, obtenue graduellement à partir de la précédente en ajoutant des contraintes additionnelles. Blech et al. concluent que la deuxième approche sémantique avec les ensembles relationnels est plus adaptée pour la preuve de séquentialisation des instructions.

3.4.5 CakeML

Un projet de compilation vérifiée plus récent est CakeML [63, 17]. Il s'agit d'un compilateur vérifié pour un sous-ensemble du langage ML et ciblant diverses architectures, dont les très courantes x86-64 et ARMv8. Le compilateur est lui-même écrit dans ce sous-ensemble de ML et vérifié à l'aide du prouveur HOL4 [36]. ML étant un langage fonctionnel, certains choix différents de ceux de CompCert sont faits.

La partie du compilateur dédiée aux optimisations réalise quelques optimisations simples, dont de la propagation de constantes. Un renommage de variables dans l'esprit de SSA, suivi d'une passe d'élimination de code mort,

est réalisé avant l'allocation de registres au niveau du langage intermédiaire WordLang, afin d'en améliorer les performances. Cependant, il ne s'agit pas vraiment d'une forme SSA, car elle ne fait pas l'usage de ϕ-instructions, se contentant plutôt de générer des déplacements des valeurs des variables au cours du renommage afin d'éviter d'introduire des ϕ-instructions. De plus, le graphe de flot de contrôle dans WordLang a une particularité importante : il est acyclique. Ceci est dû à l'utilisation de récursivité pour décrire les boucles en CakeML, produisant donc naturellement un flot de contrôle uniquement vers l'avant.

3.4.6 LVC

Le projet LVC est un compilateur vérifié en Coq pour un langage du premier ordre appelé IL. Schneider et al. [59] donnent une sémantique fonctionnelle et une sémantique impérative pour IL. La sémantique impérative interprète les liaisons comme des affectations à une variable et les appels de fonction comme des `goto`. Ils proposent un transformation d'allocation de registres en forme SSA pour IL, inspirée de travaux précédents de Hack et al. [33], dont un des avantages est de permettre de considérer l'allocation de registres séparément du *spilling*. En particulier, cela leur permet de raisonner séparément, d'un point de vue compilation vérifiée, sur cette phase de *spilling* [57].

3.5 Conclusion

Ce chapitre nous a permis d'introduire les notions fondamentales de la compilation vérifiée. Nous avons, de plus, présenté un panorama des divers efforts menés jusqu'à présent en compilation vérifiée optimisante. Nous remarquons que même si certains de ces travaux ont étudié des représentations SSA, aucun n'a étudié une représentation avec des dépendances de données relâchées à la *Sea of Nodes* : les travaux s'en approchant le plus sont ceux de Blech et al. [8], mais ils limitent le cadre à un bloc de base.

Le chapitre suivant étudie donc, d'un point de vue formel, la syntaxe et la sémantique d'une forme *Sea of Nodes*.

Chapitre 4

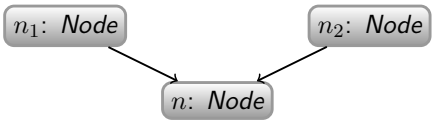
Sea of Nodes : syntaxe et sémantique

Dans ce chapitre, nous allons donner une syntaxe et sémantique formelles pour une représentation intermédiaire *Sea of Nodes*. La syntaxe et la sémantique présentées ont fait l'objet d'un développement Coq [24, 61].

4.1 Syntaxe

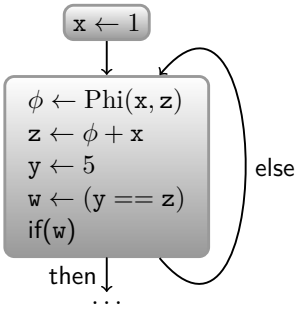
Nous représentons une fonction *Sea of Nodes* à l'aide d'un graphe g. Dans ce graphe, les nœuds représentent des instructions ou des auxiliaires pour le flot de contrôle. Les arcs représentent des dépendances entre nœuds : un arc d'un nœud n vers un nœud n' signifie que n' dépend de n. Intuitivement, l'évaluation de n est nécessaire à l'évaluation de n'. Le nœud n est appelé *entrée* ou *prédécesseur* de n', et n' est appelé une *sortie* ou *successeur* de n

Chaque nœud a un unique identifiant dans le graphe. Il est de plus défini par sa nature (définie plus loin), ainsi que par la liste de ses nœuds d'entrée. Pour un nœud d'identifiant n de nature *Nœud*, et d'entrées n_1 et n_2, nous utilisons la représentation graphique suivante :

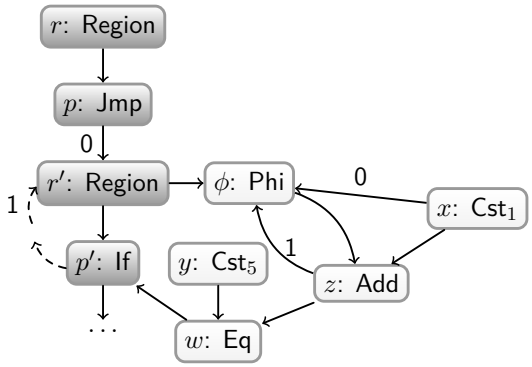


De plus, pour faciliter la lecture, nous utilisons différents tons de gris pour distinguer les nœuds de natures différentes.

Un exemple illustratif. Avant d'entrer plus en détail dans la définition technique de notre représentation *Sea of Nodes*, nous allons l'illustrer sur un exemple simple. La figure 4.1 donne un exemple d'un programme SSA simple représenté à l'aide d'un graphe de flot de contrôle entre blocs de



(a) Programme sous forme de graphe de flot de contrôle



(b) Graphe *Sea of Nodes* correspondant. Les index des prédécesseurs des nœuds Phi et des nœuds de région sont indiqués par des étiquettes sur les arcs.

FIG. 4.1 : Exemple de programme SSA et sa forme *Sea of Nodes* correspondante.

base (voir figure 4.1a), ainsi que son graphe *Sea of Nodes* correspondant (figure 4.1b).

Portons notre attention sur la figure 4.1a. Le programme représente une boucle incrémentant un compteur. Le compteur est initialisé à 1, puis incrémenté de 1 à chaque itération, jusqu'à ce qu'il atteigne la valeur 5. Notons que le programme est sous forme SSA. En effet, chaque variable est affectée au plus une fois. Pour assurer la propriété de définition unique aux points de jonction, nous rappelons que, dans SSA, pour un point de jonction avec m prédécesseurs, une phi-instruction $\mathtt{x} \leftarrow \mathtt{Phi}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_m)$ sélectionne à l'exécution, en fonction du chemin du flot de contrôle qui est exécuté, la bonne définition à utiliser parmi les définitions $\mathtt{x}_i$ atteignant le point de jonction. Le $\mathtt{x}_k$ correspondant est affecté à $\mathtt{x}$, où k correspond à l'index du prédécesseur d'où provient l'exécution. Dans l'exemple de programme, le point de jonction du flot de contrôle est au point d'entrée de la boucle, et la phi-instruction $\phi \leftarrow \mathtt{Phi}(\mathtt{x}, \mathtt{y})$ est utilisée pour sélectionner la bonne définition du compteur (parmi x et z). Du coup, lorsque l'exécution arrive dans le corps de la boucle pour la première fois, x sera sélectionné. À partir de la deuxième itération de la boucle, z sera sélectionné. Afin de définir la sémantique des phi-instructions, il faut donc considérer que les prédécesseurs d'un point de jonction sont ordonnés, de sorte à être capable de choisir le bon argument de la phi-instruction.

Considérons maintenant la figure 4.1b, pour mettre en avant les aspects importants de notre forme *Sea of Nodes*. Certains nœuds servent à représenter les calculs de données (nœuds en gris clair dans la figure 4.1b) et d'autres nœuds qui servent pour l'exécution du flot de contrôle (nœuds en

gris foncé dans la figure 4.1b).

Pour prendre un exemple de nœud de données, z correspond au pseudo-code $\mathtt{z} \leftarrow \phi + \mathtt{x}$, ses entrées sont les nœuds ϕ et x. De façon analogue, w est un nœud de données correspondant au pseudo-code $\mathtt{w} \leftarrow (\mathtt{y} == \mathtt{z})$. Les nœuds relatifs au flot de contrôle ont des fonctions plus variées. Les nœuds Region sont les analogues des blocs de base. Dans le graphe de l'exemple, on trouve deux tels nœuds, r et r', un pour chaque bloc de base du programme SSA. Maintenant, afin de pouvoir effectuer la transition d'une région vers une autre, avec *Sea of Nodes* on utilise des nœuds de *contrôle* (les nœuds p et p' dans l'exemple). Les nœuds de contrôle marquent la « fin » d'une région, de façon similaire aux instructions de branchement que l'on trouve habituellement à la fin d'un bloc de base.

Un point à noter est que le nœud ϕ prend une région r' en entrée : on a besoin de cette dépendance pour définir la valeur calculée par ϕ. Sous forme SSA, les prédécesseurs des nœuds de région tout comme les nœuds de données en entrée des nœuds Phi doivent être ordonnés. Leur rang dans cet ordre est utilisé pour déclencher l'évaluation de la bonne entrée d'un nœud Phi. Dans les figures, on précisera donc parfois l'index du prédécesseur en étiquette des arcs de dépendances (voir figure 4.1b).

Hormis les nœuds Phi, les nœuds de données ne dépendent pas, du moins directement, d'une région (voir par exemple les nœuds de données x et y). Les nœuds flottent donc dans le graphe. En particulier, aucune contrainte ne précise si le nœud x doit être évalué avant ou après le nœud y, contrairement à ce que la version sous forme de blocs de base pourrait nous laisser croire.

Définition formelle des nœuds. La table 4.1 regroupe la définition formelle d'un graphe *Sea of Nodes*. Un graphe est une fonction (partielle) d'identifiants de nœuds vers des structures représentant la nature des nœuds, ainsi que leurs nœuds en entrée.

Dans la table, ainsi que dans le reste du manuscrit, nous utilisons des conventions de notation pour exprimer la nature d'un nœud à travers son identifiant (par exemple, les nœuds de données seront notés $x, y, z, \ldots$, tandis que les nœuds de région seront notés $r, r', \ldots$).

Les nœuds de données représentent des instructions qui calculent une valeur numérique. On considère les nœuds suivants :

Cst_N correspond a une constante N. Ce nœud ne dépend pas d'un nœud région (il flotte).

$binop(x_1, x_2)$ correspond à une opération arithmétique binaire. Il prend deux nœuds de données en entrée x_1 et x_2. On ne considère que des opérations arithmétiques qui ne sont pas fautives, c'est-à-dire, en d'autres termes, que l'on ne considère pas d'opérations qui lancent une exception ou qui correspondent à un comportement indéfini

	Graphe	
$g \in$	$\text{id} \hookrightarrow \textit{node}$	graphe
id =	$\mathbb{N}$	identifiant de nœud
	Nœuds	
node ::=	*data*	
	region	
	control	
	branch	
data ::=	Cst_N	constante numérique
	$\textit{binop}(x, y)$	opération binaire
	phi	
binop ::=	Add	addition
	Eq	égalité booléenne
	Mul	multiplication
phi ::=	$\text{Phi}(r, x_1, \ldots, x_m)$	ϕ-node
region ::=	$\text{Region}(p_1, \ldots, p_m)$	région
control ::=	*jump*	
	cond	
	$\text{Return}(r, x)$	`return`
jump ::=	$\text{Jmp}(r)$	sauts
cond ::=	$\text{If}(r, x)$	conditionnelle
branch ::=	$\text{IfT}(\textit{if})$	`then`
	$\text{IfF}(\textit{if})$	`else`
	Notations	
id	$\ni x, y, z, w$	pour *data*
	$\ni \phi$	pour *phi*
	$\ni r$	pour *region*
	$\ni c$	pour *control*
	$\ni \textit{if}$	pour *cond*
	$\ni p$	pour *jump* ou *branch*
	$\ni n, n'$	pour *node*

TAB. 4.1 : *Sea of Nodes* : graphe, syntaxe des nœuds et notations

(comme les divisions par zéro). Ces nœuds ne dépendent pas, du moins directement, d'un nœud de région.

Phi$(r, x_1, \ldots, x_m)$ correspond à une phi-instruction. Il dépend d'un nœud de région r, et de m nœuds de données $x_1, \ldots, x_m$. La dépendance sur r est intrinsèque à SSA : la valeur du nœud Phi est déterminée par le prédécesseur de r d'où l'on arrive.

Portons notre attention sur la description des nœuds liés au flot de contrôle du programme.

Region$(p_1, \ldots, p_m)$ est un nœud de région avec m prédécesseurs en entrée. Les prédécesseurs d'une région sont des nœuds correspondants à des branchements dans le flot de contrôle. Une seule région dans le graphe n'a pas d'entrées : le point d'entrée du graphe.

Il y a trois sortes de nœuds de *contrôle* : Les nœuds Jmp, If et Return. Chaque région a exactement un nœud de contrôle qui en dépend. Ces nœuds marquent la fin d'une région et correspondent aux instructions de branchement traditionnellement trouvées à la fin d'un bloc de base.

Jmp(r) est un nœud de contrôle dépendant d'une région r et correspondant à un saut à la fin de r vers une autre région.

If(r, x) est un nœud de contrôle dépendant d'une région r et correspondant à un branchement conditionnel avec condition x.

Return(r, x) est un nœud de contrôle dépendant d'une région r. Il marque la fin d'exécution de la fonction, retournant la valeur du nœud de données x en entrée.

Il y a aussi deux nœuds auxiliaires pour les branches `then` et `else` d'un nœud If. Ce sont les *nœuds de projection* [21], dont le propos est de distinguer les deux valeurs booléennes possibles du nœud If. Il s'agit d'un choix syntaxique pour gérer le résultat du nœud If qui est intuitivement un couple de valeurs, l'une pour la branche `then` et l'autre pour la branche `else`. Ce choix rend explicites les dépendances de contrôle sur l'une ou l'autre des branches issues du nœud If, ce qui est utile pour des optimisations qui veulent traiter différemment chaque branche. Ainsi que l'a proposé Click [21], on évite d'attacher cette information sous forme d'étiquettes sur les arcs, déplaçant l'information plutôt sur deux nœuds auxiliaires. Ces nœuds de projection ont été précédemment omis dans la figure 4.1 pour plus de clarté.

IfF(if), IfT(if) sont des nœuds de projection dépendant d'un nœud If *if*. Un nœud IfF correspond à l'entrée de la branche prise dans le cas d'une condition `false`, tandis que le nœud IfT correspond au cas `true`.

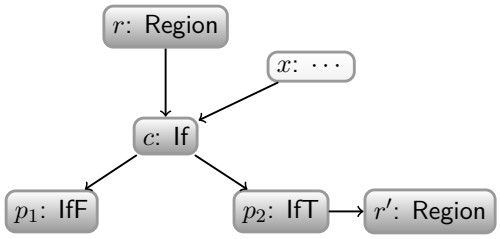


FIG. 4.2 : Conditionnelle et nœuds de projection

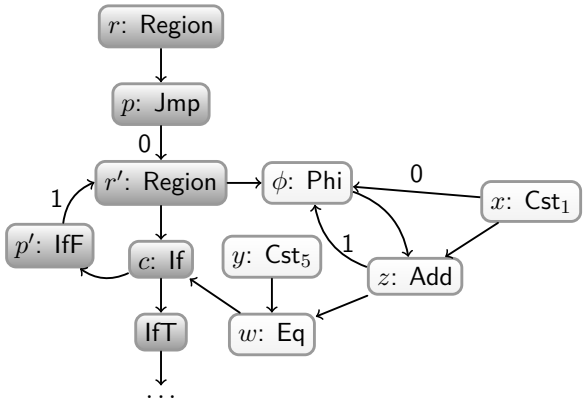


FIG. 4.3 : L'exemple du programme SSA de compteur dans sa forme *Sea of Nodes* avec projections

Illustrons les nœuds projection à l'aide de la figure 4.2. La figure montre deux régions r et r'. Le nœud de contrôle dépendant de r est un nœud If. Ce nœud a deux successeurs qui sont des projections : les nœuds IfF et IfT. L'évaluation du nœud x détermine quelle projection est évaluée. Si cette évaluation donne un résultat `true`, alors le nœud IfT est évalué, et l'exécution se déplace de la région r vers la région r'.

Par ailleurs, la figure 4.3 contient l'exemple du programme de compteur de la figure 4.1, mais sans omettre les projections cette fois-ci.

Conditions de bonne formation. Les définitions syntaxiques de la table 4.1 ne reflètent pas toutes les contraintes qu'un graphe *Sea of Nodes* doit satisfaire pour avoir une sémantique bien définie. Nous listons ci-dessous ces conditions.

Contraintes SSA En forme SSA, l'arité des phi-instructions se trouvant à l'entrée de chaque point de jonction doit être égale au nombre de prédécesseurs du point de jonction. Avec *Sea of Nodes*, comme on le voit sur la figure 4.1b, cette contrainte s'exprime par le fait que, pour chaque nœud ϕ de la forme $\text{Phi}(r, x_1, \ldots, x_m)$ qui dépend de la région r, le nombre de prédécesseurs de r est égal à m, le nombre

d'entrées du nœud ϕ.

Déterminisme Chaque sortie de région doit être non-ambigüe : pour chaque nœud de contrôle c, il doit exister un unique nœud de région successeur r'. Pour un nœud de branchement conditionnel *if*, cette condition est chaînée de la façon suivante : il existe exactement un nœud de branchement IfT et un nœud de branchement IfF qui dépendent de *if* et les deux ont un unique nœud de région comme successeur.

Bonne formation des dépendances de données Le sous-graphe des données de dépendances entre nœuds de données (hormis nœuds Phi) doit être acyclique. Les cycles empêcheraient l'évaluation des nœuds de données.

Strictness La propriété SSA classique de *strictness* doit être vérifiée : chaque utilisation de variable doit être dominée par l'unique définition de cette variable. En *Sea of Nodes*, la définition d'*utilisation*, *dominance* et *définition* doivent être adaptées, mais la propriété reste cruciale pour assurer que l'on peut donner à chaque nœud de données une valeur durant l'exécution. Le chapitre 5 donne une définition formelle de ces notions dans le cadre de *Sea of Nodes*.

4.2 Sémantique

Nous donnons maintenant une sémantique pour notre forme *Sea of Nodes*. Nous utilisons une sémantique dénotationnelle pour évaluer les nœuds de données, tandis que nous utilisons une sémantique petit-pas pour propager l'exécution d'un nœud de région vers un autre. Nous considérons une fonction *Sea of Nodes* avec un graphe g. Tous les nœuds considérés sont des nœuds de ce graphe g.

Environnement. L'évaluation par sémantique dénotationnelle des nœuds de données utilise un environnement ρ, une fonction partielle de nœuds Phi vers des valeurs. En effet, les nœuds Phi sont les seuls nœuds de données dont la valeur dépend directement du flot de contrôle et qui, par conséquent, nécessitent un traitement particulier. Les valeurs des autres nœuds de données (constantes et opération arithmétiques binaires) n'ont pas besoin d'être stockées dans l'environnement ρ.

La sémantique petit-pas met à jour l'environnement à chaque fois qu'un point de jonction est atteint : pour chaque nœud Phi de la région d'arrivée, on actualise la valeur, de façon analogue à ce qui est fait dans la sémantique de CompCertSSA [4].

Notations. Dans le reste de la section, on fait référence à une valeur numérique retournée par une instruction correspondant à un nœud de données par v. Les valeurs spéciales `false` et `true` sont des alias pour 0 et 1.

Un état d'exécution de la fonction, noté σ, est de la forme Start (état initial), $\text{Exec}(r, \rho)$ (état intermédiaire d'exécution), ou $\text{Ret}(v)$ (état de retour), où

- r représente la région courante ;
- ρ est une fonction partielle des nœuds Phi vers des valeurs ;
- v est la valeur de retour de la fonction.

On utilise la notation suivante également :

$\text{op}_{binop}(v_1, v_2)$ représente la fonction sous-jacente correspondant au nœud *binop*. Par exemple, si le nœud est un nœud Add correspondant à une addition, cette fonction est l'opération d'addition +.

4.2.1 Sémantique dénotationnelle : nœuds de données

L'évaluation des nœuds de données est proche de l'interprétation naturelle des nœuds de données en tant qu'expressions dans *Sea of Nodes* et se prête bien à une sémantique dénotationnelle. En effet, le sous-graphe dont la racine est un nœud x et qui est composé des dépendances des nœuds de données (hormis Phi), est un graphe orienté acyclique, où les constantes et les nœuds Phi sont les feuilles, et les *binop* jouent le rôle des nœuds internes. Comme on le voit dans la figure 4.1b, le sous-graphe dont la racine est en w est composé de tous les nœuds gris clair. Cette structure de graphe orienté acyclique correspond à la représentation classique d'une expression, mais avec partage de sous-arbres communs. Les nœuds Phi doivent être vus comme les variables de telles expressions.

Grâce à la bonne formation des dépendances de données telle qu'expliquée à la Section 4.1, on peut calculer récursivement la valeur $[\![x]\!]_\rho$ d'un nœud de données x en utilisant la valeur des nœuds Phi dans l'environnement courant ρ.

$$[\![x]\!]_\rho = \begin{cases} N & \text{si } g(x) = \text{Cst}_N \\ \rho(x) & \text{si } g(x) = \text{Phi}(\ldots) \\ \text{op}_{binop}([\![x_1]\!]_\rho, [\![x_2]\!]_\rho) & \text{si } g(x) = binop(x_1, x_2) \end{cases}$$

Par exemple, pour le nœud w de la figure 4.1b, on obtient :

$$[\![w]\!]_\rho = (5 == \rho(\phi) + 1).$$

La bonne formation des dépendances de données assure la terminaison du calcul de $[\![x]\!]_\rho$, mais elle assure aussi que, pour tout ϕ appartenant au graphe acyclique de dépendance de l'expression associée à x, $\rho(\phi)$ est défini. C'est ici que l'hypothèse de *strictness* entre en jeu : pendant l'exécution d'une fonction, dans chaque région où l'on a besoin d'évaluer $[\![x]\!]_\rho$, les régions auxquelles tous ces nœuds ϕ appartiennent ont déjà été atteintes. Par conséquent, ces nœuds ϕ sont définis pour ρ.

4.2.2 Sémantique opérationnelle : évaluation des régions

Nous décrivons maintenant l'exécution d'une fonction *Sea of Nodes* avec un graphe g en utilisant une sémantique opérationnelle $\rightarrow_{\text{STEP}}$. La relation est de la forme $\sigma \rightarrow_{\text{STEP}} \sigma'$, ce qui correspond à évaluer une région et déplacer l'exécution d'un état vers un autre.

L'environnement ρ dans un état $\text{Exec}(r', \rho)$ a besoin d'être actualisé lorsqu'on atteint une région r avec plus d'un prédécesseur (un point de jonction avec une liste de nœuds Phi). Cette actualisation utilise une sémantique d'évaluation parallèle, et affecte tous les nœuds Phi qui dépendent de la région r.

Formellement, on écrit $p, r, \rho \rightarrow_\phi \rho'$ le jugement qui spécifie comment un environnement ρ est actualisé en un environnement ρ' lorsque l'exécution atteint une région r, en provenant d'un prédécesseur p de r. Le jugement est défini par la règle suivante :

$$\text{PHIS}\ \frac{\begin{array}{c}\text{philist}(r) = [\phi_1, \ldots, \phi_m] \qquad \text{index}(p, r) = k \\ \forall i = 1, \ldots, m,\ \text{ntharg}(\phi_i, k) = x_i \qquad [\![x_i]\!]_\rho = v_i\end{array}}{p, r, \rho \rightarrow_\phi \rho[\phi_i \mapsto v_i \text{ for all } i]}$$

Si n est une entrée de n' dans le graphe, on écrit $\text{index}(n, n')$ l'index de n parmi la liste des entrées du nœud n'. On écrit $\text{ntharg}(\phi, k)$ le k-ième nœud de données en entrée d'un nœud Phi ϕ, et $\text{philist}(r)$ la liste des nœuds Phi qui dépendent directement de r (c'est-à-dire dont la première entrée est r).

La règle se lit comme suit : on actualise chaque nœud Phi qui dépend de r avec la valeur calculée par le k-ième nœud de données en entrée, où k est l'index du prédécesseur p parmi les entrées de la région r.

Dans l'exemple de la figure 4.1, cette règle est utilisée pour actualiser successivement l'environnement à l'entrée de la boucle. Lorsque l'exécution arrive depuis p en r' initialement, on a $\text{index}(p, r') = 0$, donc l'environnement est actualisé de sorte que $\rho_0(\phi) = [\![x]\!]_\rho = 1$. Alors, tant que $[\![w]\!]_{\rho_i} = \texttt{false}$, l'exécution boucle depuis p' vers r'. Puisque dans ce cas $\text{index}(p', r') = 1$, on obtient ceci successivement pour les itérations suivantes : d'abord $\rho_1(\phi) = [\![z]\!]_{\rho_0} = 2$, après $\rho_2(\phi) = [\![z]\!]_{\rho_1} = 3$ et finalement $\rho_3(\phi) = [\![z]\!]_{\rho_2} = 4$.

Actualiser la valeur des nœuds Phi en arrivant dans la région à laquelle ils appartiennent est un choix qui permet d'éviter d'avoir à instrumenter

l'état d'exécution : si l'environnement était actualisé au début de l'évaluation de leur région, il serait nécessaire de conserver l'information sur le prédécesseur à partir duquel l'exécution était venue.

Nous décrivons maintenant en détail les différentes règles pour la sémantique opérationnelle $\rightarrow_{\text{STEP}}$. On écrit $\text{entry}(g)$ pour faire référence au point d'entrée du graphe g. Afin d'alléger les notations, on écrit simplement r, ρ pour les états d'exécution de la forme $\text{Exec}(r, \rho)$, ce qui ne crée aucune confusion possible.

La première règle, START, correspond au début d'exécution de la fonction et à l'initialisation de l'état d'exécution : la région initiale est $\text{entry}(g)$, tandis que l'environnement initial est la fonction partielle ρ_{empty}, dont le domaine est vide.

$$\text{START}\frac{\text{entry}(g) = r_0}{\text{Start} \rightarrow_{\text{STEP}} r_0, \rho_{\text{empty}}}$$

Ensuite, nous présentons des règles pour les états d'exécution intermédiaires. Les règles suivent le motif récurrent suivant :

- Évaluer le nœud de contrôle c de la région courante r ;
- Déterminer la région successeur r' ;
- Actualiser l'environnement ρ pour les nœuds Phi de cette région successeur.

Dans la règle JMP, le nœud de contrôle de r est un nœud Jmp. Dans ce cas, le pas s'effectue simplement vers la région successeur r', dont on sait qu'il est uniquement déterminé, grâce aux conditions de bonne formation sur notre *Sea of Nodes*.

$$\text{JMP}\frac{g(c) = \text{Jmp}(r) \qquad g(r') = \text{Region}(\ldots, c, \ldots) \qquad c, r', \rho \rightarrow_{\phi} \rho'}{r, \rho \rightarrow_{\text{STEP}} r', \rho'}$$

L'actualisation de l'environnement peut être réalisée puisque l'on sait depuis quel prédécesseur de r' l'exécution se fait, en l'occurrence depuis c (deuxième prémisse). Ce motif d'actualisation de l'environnement se retrouve dans toutes les règles pour des nœuds de contrôle qui ont un successeur, c'est-à-dire tous les nœuds de contrôle sauf le nœud Return.

La règle JMP est illustrée dans la figure 4.4. La figure montre deux nœuds de région r et r'. La région r a un nœud Jmp comme nœud de contrôle, qui induit l'évaluation de r'. L'environnement est actualisé pour les nœuds Phi dépendant de r'.

Les règles IFF et IFT correspondent au cas où le nœud de contrôle de la région courante est un nœud If avec des nœuds projection IfF et IfT. Les deux règles sont similaires à la règle JMP, à part que l'actualisation de

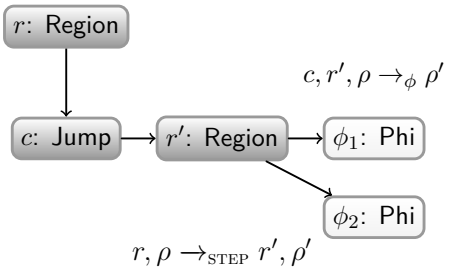


FIG. 4.4 : JMP Rule Illustration

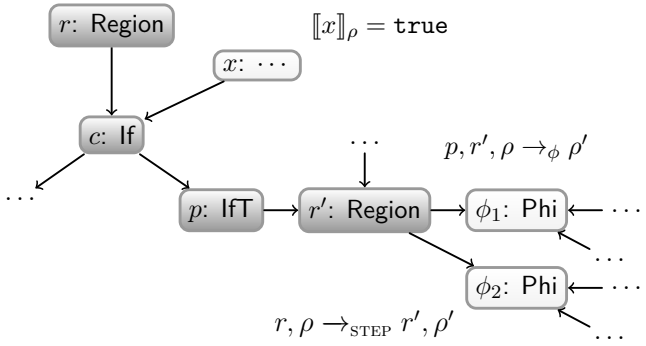


FIG. 4.5 : IFT Rule Illustration

l'environnement considère un nœud de projection, plutôt que le nœud If, en tant que prédécesseur de la région suivante.

$$\text{IFF}\ \frac{g(if) = \text{If}(r, x) \qquad [\![x]\!]_\rho = \texttt{false} \qquad g(p) = \text{IfF}(if) \qquad g(r') = \text{Region}(\ldots, p, \ldots) \qquad p, r', \rho \to_\phi \rho'}{r, \rho \to_{\text{STEP}} r', \rho'}$$

$$\text{IFT}\ \frac{g(if) = \text{If}(r, x) \qquad [\![x]\!]_\rho = \texttt{true} \qquad g(p) = \text{IfT}(if) \qquad g(r') = \text{Region}(\ldots, p, \ldots) \qquad p, r', \rho \to_\phi \rho'}{r, \rho \to_{\text{STEP}} r', \rho'}$$

La figure 4.5 illustre la règle IFT. La figure montre les nœuds de région r et r'. Le premier a un If pour nœud de contrôle. Le nœud x s'évalue vers `true`, donc la branche IfT est évaluée. L'environnement est actualisé pour les nœuds Phi dépendant de r'.

Enfin, il y a la règle RET, applicable lorsque le nœud de contrôle de la région courante est un Return. Cela correspond à un retour de fonction, et la valeur du nœud de données en entrée se retrouve dans l'état de retour :

$$\text{RET}\ \frac{g(c) = \text{Return}(r, x) \qquad [\![x]\!]_\rho = v}{r, \rho \to_{\text{STEP}} \text{Ret}(v)}$$

Exécution de la fonction. On a des règles uniques START pour l'initialisation, et RET pour le retour de fonction. L'exécution d'une fonction est donc de la forme : $\text{Start} \rightarrow_{\text{STEP}} \cdots \rightarrow_{\text{STEP}} \text{Ret}(v)$.

Dans la suite du manuscrit, nous utilisons la valeur retournée par la fonction comme l'unique comportement observable d'une exécution de fonction qui termine. Les preuves présentées peuvent être naturellement étendues à des listes d'évènements observables, tels que les produirait par exemple l'ajout d'une instruction Print.

4.3 Extensions pour notre *Sea of Nodes*

Dans cette section, on discute brièvement de quelques extensions de notre représentation *Sea of Nodes* qui ne sont pas traitées dans le manuscrit.

Par exemple, dans ce manuscrit, nous ne modélisons pas les appels de fonction. On ne considère donc que des programmes avec une seule fonction. Dans le cas général, une approche telle que celle utilisée dans CompCert [46] pourrait être utilisée. On omet également, pour plus de clarté, les opérations liées à la mémoire, même si elles sont traitées dans le développement Coq.

4.3.1 Mémoire

Les opérations liées à la mémoire sont en *Sea of Nodes* gérées en tant que variables SSA. On introduit donc une valeur mémoire spécifique pour représenter les différentes versions de la mémoire durant l'exécution, et on affecte explicitement des variables SSA représentant l'état courant de la mémoire après chaque *load* ou *store*. Ces deux instructions additionnelles n'interviennent pas significativement dans les résultats présentés dans ce manuscrit et c'est pourquoi nous les avons omis de la présentation. Notons que dans le développement Coq sur *Sea of Nodes*, nous avons introduit les opérations liées à la mémoire, pour nous assurer qu'effectivement leur inclusion ne joue pas un rôle dans les optimisations que nous traitons. Voici la forme des nœuds correspondant à ces instructions mémoire, qui sont des nœuds de données :

$\text{Load}(r, mn, x)$ correspond à une instruction *load* dépendant du nœud de région r, prenant en entrée le nœud représentant la mémoire mn (un nœud de données Store ou un nœud de constante) et récupérant la valeur à l'adresse donnée par la valeur du nœud de données x.

$\text{Store}(r, mn, x, y)$ correspond à une instruction *store* dépendant du nœud de région r, prenant en entrée le nœud représentant la mémoire mn et écrivant la valeur du nœud de données y à l'adresse donnée par le nœud de données x.

Notons que des nœuds Phi sont utilisés pour fusionner plusieurs nœuds produisant une valeur mémoire, comme pour les nœuds de données produisant des valeurs numériques.

4.3.2 Appels système

Comme mentionné précédemment, dans le manuscrit nous avons utilisé la valeur retournée par la fonction comme l'unique comportement observable d'une exécution de fonction qui termine. On pense que les preuves présentées peuvent être étendues à des listes d'évènements observables produits par des appels systèmes.

Essentiellement, chaque pas d'exécution produirait une trace, par exemple de valeurs affichées dans des nœuds représentant une instruction d'affichage Print (plus précisément dans un OS POSIX il s'agirait des arguments d'un appel à un `write(2)` en l'occurrence). Il faut donc, au moment de la preuve de simulation d'une transformation, montrer que les valeurs des nœuds en entrée de ces appels sont préservées. Ceci se traite de façon analogue à la preuve que la valeur retournée par la fonction par le nœud Return est la même, c'est-à-dire en appliquant les invariants de la simulation aux nœuds en question.

4.3.3 Opérations bloquantes ou fautives

Certaines opérations, non traitées dans le reste du manuscrit, comme une division par zéro, sont fautives, c'est-à-dire qu'elles correspondent à une exception ou à un comportement indéfini. Une façon de traiter ces dernières est de considérer que, dans ces cas-là, la sémantique bloque. Cependant, ceci pose la question de définir des restrictions au moment du *Global Code Motion* (voir chapitre 7) pour le placement des instructions pouvant conduire à une faute, comme le plus petit dominateur des utilisations d'un nœud. La formalisation de ceci, non traitée dans ce manuscrit, est une piste intéressante pour des travaux futurs.

4.4 Conclusion

Dans ce chapitre, nous avons proposé une sémantique formelle pour *Sea of Nodes*. Elle présente deux caractéristiques fondamentales. Tout d'abord, elle ne lie pas le calcul d'une donnée à un nœud de région particulier : les nœuds de données (et leurs dépendances) sont plutôt évalués à la demande, en utilisant une sémantique dénotationnelle dans un environnement qui garde la valeur actuelle des nœuds Phi. Pendant le travail de formalisation, on a utilisé un prototype d'interpréteur reflétant les règles sémantiques pour vérifier que notre formalisation était en effet exécutable.

Dans les chapitres suivants, nous utilisons cette sémantique pour prouver des propriétés et la correction sémantique d'optimisations.

Chapitre 5

Sea of Nodes : domaines de validité pour valeurs

Beaucoup d'optimisations de compilation, comme GVN [58], la propagation de constantes, ou l'élimination d'assertions de pointeurs non nuls redondants [40] demandent de raisonner, statiquement, sur les valeurs possibles que les variables ou expressions peuvent avoir à certains endroits du programme. En pratique, ces optimisations profitent de la structure des programmes en utilisant des notions telles que la *dominance*, afin de pouvoir décrire plus facilement où, dans le programme, une valeur particulière est valide ou préservée.

Ce chapitre donne, en termes de dominance, une propriété fondamentale qui est utile pour de tels raisonnements dans le contexte de la représentation intermédiaire *Sea of Nodes*. La propriété fondamentale et tous les lemmes sémantiques intermédiaires ont fait l'objet d'un développement Coq [24, 61].

5.1 Propriété de préservation de valeur

Nous donnons maintenant des conditions sous lesquelles la valeur d'un nœud de données est préservée le long d'un chemin d'exécution. Nous considérons, comme dans le chapitre précédent, une fonction *Sea of Nodes* avec un graphe g.

5.1.1 Phi-Dépendances d'un nœud de données

Nous utilisons la notation $\mathrm{phideps}(x)$ pour faire référence à l'ensemble des nœuds Phi apparaissant dans l'expression associée à x. Plus formellement :

$$\text{phideps}(x) = \begin{cases} \emptyset & \text{si } g(x) = \text{Cst}_N \\ \{x\} & \text{si } g(x) = \text{Phi}(\ldots) \\ \text{phideps}(x_1) \cup \text{phideps}(x_2) & \\ & \text{si } g(x) = \mathit{binop}(x_1, x_2) \end{cases}$$

Dans le graphe de l'exemple de la figure 4.1, on a $\text{phideps}(x) = \emptyset$, $\text{phideps}(y) = \emptyset$, $\text{phideps}(z) = \{\phi\}$ et $\text{phideps}(w) = \{\phi\}$.

5.1.2 Nœuds Phi et valeur d'un nœud

Nous énonçons un lemme qui assure que la valeur d'un nœud de données ne dépend que de la valeur des nœuds Phi dont il dépend (au sens de phideps).

Lemme 5.1.1. *Soit x un nœud de données du graphe g. Soit ρ et ρ' deux environnements. Si, pour tout $\phi \in \text{phideps}(x)$ on a $\rho(\phi) = \rho'(\phi)$, alors $[\![x]\!]_\rho = [\![x]\!]_{\rho'}$.*

Démonstration. Notons tout d'abord que, grâce à la bonne formation des dépendances de données, l'hypothèse assure que $[\![x]\!]_\rho$ est défini, parce que tous les $\rho(\phi)$ pour $\phi \in \text{phideps}(x)$ sont définis. La preuve se fait par induction sur la définition de $[\![x]\!]_\rho$: le cas de base de l'induction est assuré par le fait que les deux évaluations coïncident sur les nœuds Phi par hypothèse. □

Le lemme suivant affirme que la valeur d'un nœud de données x est préservée du moment que l'exécution ne se déplace pas dans une région dont un ϕ appartient à $\text{phideps}(x)$.

Lemme 5.1.2. *Soit*

$$\text{Exec}(r_0, \rho_0) \cdots \rightarrow_{STEP} \text{Exec}(r_m, \rho_m)$$

un chemin d'exécution. Soit x un nœud de données tel que $[\![x]\!]_{\rho_0}$ soit défini. Supposons que, pour tout $\phi \in \text{phideps}(x)$, et pour tout $i = 1 \ldots m$, on a $\phi \notin \text{philist}(r_i)$. Alors $[\![x]\!]_{\rho_0} = [\![x]\!]_{\rho_m}$.

Démonstration. On montre par induction sur un chemin de $\text{Exec}(r_0, \rho_0)$ vers $\text{Exec}(r_m, \rho_m)$, que pour tout $\phi \in \text{phideps}(x)$, on a $\rho_0(\phi) = \rho_m(\phi)$. Ceci découle du fait que dans l'application de la règle PHIS le long du chemin d'exécution, il n'y a pas de $\phi \in \text{phideps}(x)$, grâce à l'hypothèse. Le lemme 5.1.1 permet alors de conclure. □

5.2 Formulation en termes de dominance

La *dominance* est une notion utile pour l'écriture des algorithmes de certaines optimisations, en particulier celles mettant en jeu l'élimination de calculs redondants ou de contrôles à l'exécution. Il est donc naturel que des propriétés de flot de contrôle utiles à la preuve de ces optimisations bénéficient d'une formulation en termes de dominance.

5.2.1 Graphe de flot de contrôle

Tout d'abord, nous donnons l'analogue de la notion de CFG avec blocs de base dans le cas de *Sea of Nodes*.

Définition 5.2.1. On appelle graphe de flot de contrôle (CFG) d'un graphe *Sea of Nodes* g, le graphe orienté dont les nœuds sont les nœuds de région, et où il existe un arc d'un nœud r vers un nœud r' si le nœud de contrôle dépendant de r est :

- un nœud Jmp dont la sortie est r' ;
- ou un nœud If dont la sortie est un nœud de projection qui a r' en sortie.

5.2.2 Dominance

Ceci permet de définir sur le CFG de *Sea of Nodes* un analogue de la dominance, de la façon suivante :

Définition 5.2.2. Un nœud de région r domine un nœud de région r' si tout chemin dans le CFG du point d'entrée de g jusqu'à r' passe par r. La dominance est dite *stricte* si $r \neq r'$.

La relation de dominance ci-avant s'applique uniquement aux nœuds de région, c'est-à-dire à un niveau moins granulaire que ce qui est fait traditionnellement dans un CFG de blocs de base. Ci-après, nous étendons cette notion aux nœuds de données également. Informellement, pour qu'un nœud de données domine un nœud de région r, tous les Phi dont il dépend doivent appartenir à des régions qui dominent r. Formellement :

Définition 5.2.3. Un nœud de données x *domine* un nœud de région r si, pour tout ϕ et tout r_ϕ tel que $\phi \in \text{phideps}(x)$ et $g(\phi) = \text{Phi}(r_\phi, \ldots)$, on a que r_ϕ domine r. La dominance est *stricte* si $r_\phi \neq r$ pour chacun de ces ϕ.

Strictness. Ayant défini formellement la notion de dominance pour *Sea of Nodes*, revenons plus en détail sur la notion de *strictness* mentionnée au chapitre précédent, c'est-à-dire que tout nœud de données x doit dominer strictement ses usages. Pour lui donner un sens dans le contexte de *Sea*

of Nodes, il nous reste donc à définir la notion de point d'utilisation d'un nœud de données.

Définition 5.2.4. On dit qu'un nœud de données x est utilisé en un nœud de région r si x intervient dans le graphe acyclique de dépendance de données (nœuds Phi exclus) issu d'un nœud de contrôle c de r, ou issu de la k-ième entrée d'un nœud Phi dépendant d'un nœud de région r' dont r est le k-ième prédécesseur.

Par exemple, dans la figure 4.1, r' est un point d'utilisation des variables ϕ, z, y, x et w. Notons que r' est un point d'utilisation de z du fait que w dépend de z, mais aussi du fait que z est le deuxième argument de ϕ et que la sémantique des nœuds Phi évalue les entrées dans le contexte d'un prédécesseur particulier, en l'ocurrence r' (boucle). Par ailleurs, le nœud de région r est, lui aussi, un point d'utilisation de x, puisque celui-ci est le premier argument de ϕ.

Dominance et évaluabilité. Dans la littérature à propos de formalisations de SSA, le dit lemme équationnel [4] est particulièrement important. Il assure que l'équation définissant une variable x est valide aux points dominés par le point de définition de x. Ici, avec une fonction *Sea of Nodes*, les nœuds de données (hormis les nœuds Phi) n'ont pas de point de définition réel : ils flottent. Cependant, cette propriété particulière, le lemme équationnel, est en fait intégrée dans la sémantique d'évaluation des nœuds de données : la valeur d'un nœud de données est toujours valide, en quelque sorte, du moment qu'elle est définie. Le lemme suivant permet de caractériser, en utilisant la dominance, les nœuds de région dans un graphe *Sea of Nodes* où la valeur d'un nœud de données est en effet définie.

Lemme 5.2.5. *Soit* $\mathrm{Exec}(r, \rho)$ *un état d'exécution atteignable (c'est-à-dire qu'il existe un chemin d'exécution depuis* Start*). Soit* x *un nœud de données qui domine* r*. Alors* $[\![x]\!]_\rho$ *est défini.*

Démonstration. Soit π un chemin d'exécution depuis Start jusqu'à $\mathrm{Exec}(r, \rho)$.

On prouve d'abord la propriété pour un nœud ϕ avec $g(\phi) = \mathrm{Phi}(r', \ldots)$. Le fait que ϕ domine r signifie dans ce cas que r' domine r. Du coup, il existe ρ' tel que $\mathrm{Exec}(r', \rho')$ soit dans π. Notons que r' n'est pas le point d'entrée du graphe, car autrement il n'aurait pas de nœud Phi, d'après les propriétés de bonne formation du graphe. Du fait de notre sémantique d'évaluation des nœuds Phi lorsqu'on arrive sur une région r', il en résulte que $\rho'(\phi)$ est défini, donc $\rho(\phi)$ est défini aussi, puisque notre sémantique d'actualisation d'environnement ne fait qu'étendre la fonction partielle ρ' pendant l'exécution.

Considérons maintenant le cas d'un nœud de données x quelconque. D'après ce qui précède, pour tout $\phi \in \mathrm{phideps}(x)$, on a que $\rho(\phi)$ est défini, car, par définition, un tel ϕ domine r aussi, donc les hypothèses de la

propriété sont vérifiées pour ϕ. La bonne formation des dépendances de données permet alors de conclure la preuve pour x. □

Notons que le lemme ci-avant n'est pas utilisé dans les autres preuves sémantiques du manuscrit et n'a pas fait l'objet d'une preuve Coq.

5.2.3 Application à la préservation de valeur

Nous obtenons le corollaire suivant du lemme 5.1.2 pour la préservation de valeurs en termes de dominance.

Théorème 5.2.6. *Soit*

$$\text{Exec}(r_0, \rho_0) \cdots \rightarrow_{STEP} \text{Exec}(r_m, \rho_m)$$

un chemin d'exécution. Soit x un nœud de données tel que $[\![x]\!]_{\rho_0}$ soit défini. Si, pour tout $i = 1 \ldots m$, on a x qui domine strictement r_i, alors $[\![x]\!]_{\rho_0} = [\![x]\!]_{\rho_m}$.

Démonstration. Puisque x domine strictement r_i, on sait que pour tout $\phi \in \text{phideps}(x)$, on a $\phi \notin \text{philist}(r_i)$. Du coup, on peut appliquer le lemme 5.1.2. □

Ce théorème affirme que, du moment que l'exécution reste dans une zone particulière du graphe, la zone dominée par x, la valeur de x reste inchangée. Par exemple, une application de ce résultat est donnée dans le chapitre suivant pour prouver la préservation sémantique d'une transformation éliminant des *zero-checks* redondants.

Le théorème 5.2.6 est un peu moins général que le lemme 5.1.2, car il est limité à des chemins d'exécution où chaque région est dominée par un nœud de données x donné, tandis que les exécutions dans le lemme 5.1.2 peuvent continuer en dehors de la zone de dominance de x, du moment qu'elles n'atteignent pas de régions des nœuds Phi qui sont dans $\text{phideps}(x)$. Ceci dit, nous pensons que la caractérisation en termes de dominance utilisée dans ce théorème est plus proche de l'intuition que l'on a des optimisations.

5.3 Conclusion

Ce chapitre nous a donc permis d'établir une propriété fondamentale, formulée en termes de dominance, qui donne de l'information sur la valeur des nœuds de données. Dans la section 6.1 du chapitre suivant, nous illustrons comment cette propriété permet de prouver la préservation sémantique d'une élimination de *zero-checks* redondants.

Chapitre 6

Sea of Nodes : optimisations

Ce chapitre présente deux exemples d'optimisations. La section 6.1 traite de l'élimination de *zero-checks* redondants, utile en particulier pour les langages de programmation haut-niveau. La section 6.2 traite de propagation de constantes, tout d'abord une version simple, puis une version conditionnelle tenant compte du fait que certaines branches ne sont jamais prises est présentée à la section 6.3. Mis à part la version conditionnelle de propagation de constantes, les propriétés sémantiques relatives à ces optimisations ont fait l'objet d'un développement en Coq [24, 61].

6.1 Élimination de Zero-Checks redondants

Les langages de programmation haut-niveau nécessitent souvent l'introduction, par le compilateur, de nombreux contrôles explicites à l'exécution, en particulier pour tester si un pointeur est non nul et garantir une exécution sûre (d'un point de vue mémoire) du programme. Ceci complique le flot de contrôle et donc les analyses statiques et, par conséquent, gêne la portée des optimisations utilisant de telles analyses. L'élimination de contrôles redondants permet de mitiger ce problème.

Dans cette section, nous illustrons à l'aide des résultats du chapitre précédent, la correction sémantique d'une transformation d'élimination de *zero-checks* redondants sur *Sea of Nodes*.

6.1.1 Extension du langage : le nœud ZeroCheck

Nous étendons notre *Sea of Nodes* avec un nouveau nœud de contrôle, ZeroCheck(r, x). On lui donne la même sémantique que celle d'un Jmp vers son successeur, à moins que son nœud de données en entrée x s'évalue vers zéro, auquel cas l'exécution va vers un état d'erreur Ret(`retfail`). Ici, `retfail` est une nouvelle valeur spéciale qui permet de modéliser une erreur pendant l'exécution du programme.

On a donc les deux règles sémantiques suivantes :

$$\text{ZCHK1}\ \frac{\begin{array}{c} g(c) = \text{ZeroCheck}(r, x) \quad g(r') = \text{Region}(\ldots, c, \ldots) \\ c, r', \rho \rightarrow_{\phi} \rho' \qquad [\![x]\!]_{\rho} = v \neq 0 \end{array}}{r, \rho \rightarrow_{\text{STEP}} r', \rho'}$$

$$\text{ZCHK2}\ \frac{g(c) = \text{ZeroCheck}(r, x) \qquad [\![x]\!]_{\rho} = 0}{r, \rho \rightarrow_{\text{STEP}} \text{Ret}(\texttt{retfail})}$$

La figure 6.1a donne un exemple de graphe *Sea of Nodes* avec deux nœuds ZeroCheck c_1 et c_3. Le graphe de l'exemple a une boucle et trois branchements conditionnels. Nous l'utilisons dans cette section pour illustrer la présentation. La figure 6.1b donne l'arbre de dominance correspondant au graphe *Sea of Nodes*.

6.1.2 Critère de redondance

Décider si oui ou non un ZeroCheck est redondant est équivalent à décider si un nœud quelconque ne s'évalue jamais vers zéro, ce qui est un problème indécidable en général.

Par conséquent, on a besoin d'une approximation. Ici, nous utilisons le critère structurel de redondance suivant :

Définition 6.1.1. Une région r a un nœud ZeroCheck *redondant* si le fait suivant est vérifié : r est un nœud de région dont le nœud de contrôle est un ZeroCheck(r, x), et il existe un nœud de région r' qui domine strictement r dont le nœud de contrôle est un ZeroCheck(r', x).

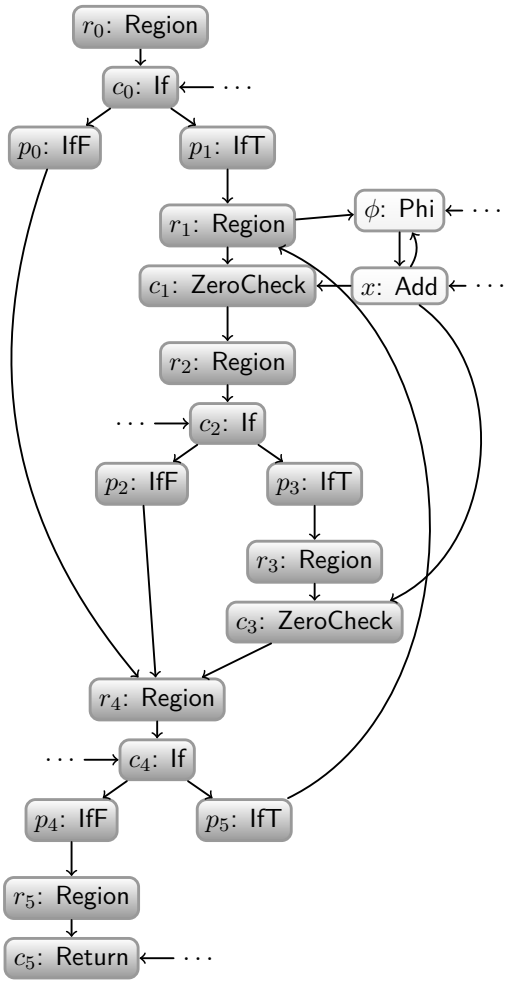
Par exemple, dans la figure 6.1a, le critère est satisfait pour r_3 : en effet, il est strictement dominé par r_1, et les deux nœuds de région ont un nœud de contrôle ZeroCheck dont le nœud de données en entrée est x.

Préservation de valeur appliquée aux *zéro-checks*. Les résultats du chapitre précédent sur des domaines de validité pour les valeurs peuvent être utilisés pour justifier l'intuition sur laquelle est fondée notre critère. On a le corollaire suivant du théorème 5.2.6 :

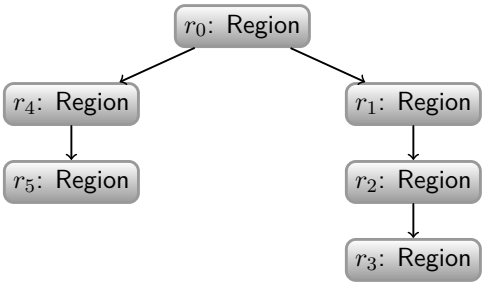
Corollaire 6.1.2. *Soit* Exec(r', ρ') *un état d'exécution atteignable. Soit* r *un nœud de région dont le nœud de contrôle est un* ZeroCheck(r, x). *Si* r *domine strictement* r', *alors* $[\![x]\!]_{\rho'} \neq 0$.

Démonstration. Il existe un chemin d'exécution π jusqu'à l'état Exec(r', ρ') par hypothèse. Puisque r domine r', il existe un état dans π dont la région est r. On considère la fin π' de π :

$$\text{Exec}(r, \rho) \cdots \rightarrow_{\text{STEP}} \text{Exec}(r', \rho')$$



(a) Exemple de programme avec un ZeroCheck redondant



(b) Arbre de dominance correspondant au programme

FIG. 6.1 : Dominance et ZeroCheck redondants

telle que pour tout autre état $\text{Exec}(r'', \rho'')$ après $\text{Exec}(r, \rho)$ dans π', $r \neq r''$. Le fait suivant est vérifié : pour de tels r'', r domine strictement r''. En effet, tout chemin π_1 du point d'entrée jusqu'à r'' peut être étendu en un chemin $\pi_1 \cdot \pi_2$ du point d'entrée jusqu'à r', où π_2 est un suffixe de π'. Comme r domine r', r appartient à $\pi_1 \cdot \pi_2$, mais r n'appartient pas à π_2, donc r appartient à π_1. Ceci prouve que r domine r''.

De plus, les propriétés de bonne formation assurent qu'un nœud domine ses utilisations, donc en particulier x domine r. Du coup, x domine strictement de tels r'' (la dominance est une relation transitive). Du fait de la présence d'un ZeroCheck en r, ajouté au fait que l'exécution ne se termine pas là, on sait que ZCHK1 a été appliquée, donc $[\![x]\!]_\rho \neq 0$. Par le théorème 5.2.6, on a $[\![x]\!]_\rho = [\![x]\!]_{\rho'}$. Dès lors, $[\![x]\!]_{\rho'} \neq 0$. □

Ce corollaire donne des hypothèses, en termes de dominance et d'atteignabilité, sous lesquelles on sait qu'un nœud s'évalue vers une valeur non nulle dans un environnement particulier. Si r' a un ZeroCheck en tant que nœud de contrôle, alors la configuration est exactement celle de notre critère de redondance : le ZeroCheck de r' est redondant. Ce corollaire est clé dans la preuve de préservation sémantique (sous-section 6.1.4).

6.1.3 Algorithme de détection de ZeroCheck redondants

L'algorithme que l'on considère ici est une variante récursive de celui trouvé dans le compilateur Go [65, 52]. Il est fondé sur une unique traversée de l'arbre de dominance du CFG, permettant d'obtenir une complexité linéaire en le nombre de nœuds de région. Nous l'avons simplement adapté au cas de notre représentation intermédiaire. En particulier, l'arbre de dominance encode la relation de dominance entre nœuds de région du graphe *Sea of Nodes*. Nous le définissons précisément dans la suite. La définition est la définition classique sur un CFG : la seule particularité, c'est de remarquer que pour *Sea of Nodes* cela ne concerne que les nœuds de région. Notons que nous ne traitons pas la construction de l'arbre de dominance dans ce manuscrit : un validateur formellement prouvé a déjà été développé dans CompCertSSA [4, 6] pour l'algorithme de Lengauer et Tarjan [45]. Il serait éventuellement intéressant de considérer des algorithmes plus simples et récents comme celui proposé par Cooper et al. [23].

Nous définissons d'abord la notion de dominateur immédiat d'un nœud de région du CFG.

Définition 6.1.3. Un nœud de région r est un *dominateur immédiat* d'un nœud de région r' si $r \neq r'$, r domine r' et r ne domine aucun autre dominateur de r'.

Par exemple, dans la figure 6.1a, le nœud r_2 est un dominateur immédiat du nœud r_3, mais r_1 ne l'est pas, car r_1 domine r_2.

Une propriété classique [45] affirme qu'il existe au plus un dominateur immédiat pour chaque nœud. Ceci, ajouté au fait que la relation de dominance stricte est acyclique, est la base pour assurer que la relation de dominance entre nœuds d'un graphe peut être encodée à l'aide d'une structure d'arbre.

Définition 6.1.4. L'arbre de dominance du CFG d'un graphe *Sea of Nodes* est un arbre où les nœuds sont des nœuds de région et où les enfants d'un nœud sont les nœuds de région dont il est le dominateur immédiat.

L'algorithme que nous considérons est donné dans la figure 6.2 en pseudo-code. En résumé, l'algorithme effectue un unique parcours en profondeur de l'arbre de dominance (`domtree`(g)), tout en maintenant à jour deux tables d'association de nœuds vers des booléens. La première, `nonZero`, garde trace du fait qu'un nœud s'évalue vers une valeur non nulle (`true`) ou bien une valeur potentiellement nulle (`false`). La deuxième table, `redundantZC`, permet de marquer les nœuds de région qui ont un ZeroCheck redondant comme nœud de contrôle.

Les deux tables d'association sont initialisées à `emptymap(node, bool)`, c'est-à-dire tous les nœuds s'envoient sur `false`. Ensuite, durant le parcours, à chaque fois qu'un nœud de région avec un nœud ZeroCheck est rencontré (ligne 6), le nœud ZeroCheck est marqué comme non nul *dans le sous-arbre de l'arbre de dominance* (c'est-à-dire les nœuds de région qu'il domine). L'algorithme procède en trois étapes. Premièrement, on sauvegarde la valeur courante de `nonZero[x]` (ligne 7). Deuxièmement, on affecte `true` à `nonZero[x]` (ligne 9) et on explore le sous-arbre (ligne 11). Finalement, on restaure l'ancienne valeur de `nonZero[x]`, une fois que les sous-arbres ont été parcourus (ligne 12). Lors de la traversée d'un sous-arbre, tout nœud ZeroCheck sur un *même* nœud de données va être marqué comme redondant (ligne 8). Par exemple, dans le cas de la figure 6.1a, lors du traitement du nœud r_1, un nœud c_1 de la forme ZeroCheck(r_1, x) est trouvé avec x en entrée. L'algorithme se souvient de cette information pendant qu'il traite les nœuds dominés par r_1, c'est-à-dire r_2 et r_3. Pendant le traitement de ces nœuds, on a donc `nonZero`$[x]$ = `true`. Lorsque, lors du traitement de r_3, l'algorithme trouve un nœud c_3 de la forme ZeroCheck(r_3, x) avec la même entrée x, il remarque que `nonZero`$[x]$ = `true`, donc il sait qu'il est redondant d'après notre critère.

On prouve que l'algorithme de la figure 6.2 calcule en effet notre critère de redondance pour les nœuds de région.

Lemme 6.1.5. *Soit* `redundantZC` = `Analyse`(g) *et* r *tels que* `redundantZC`$[r]$ = `true`. *Alors* r *a un* ZeroCheck *redondant au sens de la définition 6.1.1.*

Ce lemme ne met pas en jeu des notions sémantiques et la propriété syntaxique ne met pas en valeur des spécificités *Sea of Nodes*, mais on en donne quand même la preuve ici, car l'algorithme est un parcours intéressant de l'arbre de dominance.

```
1  Analyse(g):
2      sdom ← domtree(g)
3      nonZero ← emptymap(node, bool)
4      redundantZC ← emptymap(node, bool)
5      DepthFirst(r):
6          if control(g, r) matches ZeroCheck(r, x):
7              nz ← nonZero[x]
8              redundantZC[r] ← nz
9              nonZero[x] ← true
10             for r' in sdom.successors(r):
11                 DepthFirst(r')
12             nonZero[x] ← nz
13         else:
14             for r' in sdom.successors(r):
15                 DepthFirst(r')
16     DepthFirst(entry(g))
17     return redundantZC
```

FIG. 6.2 : Détection de ZeroCheck Redondants

Démonstration. On utilise la propriété suivante sur l'algorithme :

- $\mathcal{P}_1$: Pour tout r, on a le fait qu'un appel `DepthFirst`(r) laisse `nonZero` inchangé, c'est-à-dire identique au début et à la fin de l'appel.

Cette propriété est prouvable indépendamment du reste du lemme par induction sur l'arbre de dominance, et est due à l'utilisation de la variable temporaire `nz` qui permet de restaurer la valeur de `nonZero`$[x]$ (ligne 12).

Pour démontrer ce lemme de correction, on va prouver, de plus, les invariants suivants, au début et à la fin de chaque appel à `DepthFirst`(r) :

- $\mathcal{P}_2$: Pour tout r'' tel que `redundantZC`$[r''] =$ `true`, r'' a un ZeroCheck redondant.

- $\mathcal{P}_3(r)$: Pour tout x', si `nonZero`$[x'] =$ `true` , alors il existe un nœud de région r'' dont le nœud de contrôle est un ZeroCheck(r'', x') et qui domine strictement r.

On remarque que $\mathcal{P}_2$ implique le résultat du lemme à la fin de l'algorithme. On montre donc les deux invariants dans la suite. Tout d'abord, on remarque que les invariants sont vrais en entrée de l'appel initial, vu que les tableaux associatifs `nonZero` et `redundantZC` renvoient alors `false` pour tout nœud.

Remarquons que l'invariant $\mathcal{P}_2$, pour chaque r', dépend uniquement de `redundantZC`$[r']$, donc il suffira de vérifier qu'il est préservé pour r' à chaque modification de `redundantZC`$[r']$ (ligne 8).

La propriété $\mathcal{P}_1$ a la conséquence suivante : au début d'un appel sur $\texttt{DepthFirst}(r')$, si $\mathcal{P}_3$ est vrai pour r' en début d'appel, il est vrai pour r' au retour de la fonction. Pour prouver la préservation de $\mathcal{P}_3$, il suffira donc de prouver qu'à chaque appel $\texttt{DepthFirst}(r')$, $\mathcal{P}_3$ est vrai pour r' au moment de l'appel.

La preuve de préservation se démontre alors par induction sur l'arbre des appels récursifs, correspondant à l'arbre de dominance.

Le cas de base correspond à un appel $\texttt{DepthFirst}(r)$ où r n'a pas de successeurs dans l'arbre de dominance. D'après la remarque préliminaire sur $\mathcal{P}_2$, il suffit de vérifier la préservation pour r dans le cas où on met $\texttt{redundantZC}[r] = \texttt{true}$. Alors, c'est qu'on a un $\texttt{nonZero}[x] = \texttt{true}$ qui, d'après $\mathcal{P}_3$ nous donne un r'' dominant strictement r avec un $\text{ZeroCheck}(r'', x)$, ce qui assure $\mathcal{P}_2$ pour r.

Dans le cas général, on doit assurer $\mathcal{P}_2$ et $\mathcal{P}_3$ au début de chaque appel récursif afin de pouvoir appliquer l'hypothèse d'induction. La préservation de $\mathcal{P}_2$ jusqu'au premier appel récursif sur un successeur s'obtient par le même raisonnement que dans le cas de base, et l'application de l'hypothèse d'induction permettra d'obtenir la préservation de $\mathcal{P}_2$ pour chaque appel récursif, puisque $\mathcal{P}_2$ ne dépend pas du nœud r' sur lequel se fait l'appel récursif. Le point subtil, c'est d'assurer que $\mathcal{P}_3(r')$, qui dépend du nœud de région r' successeur traité, est vrai pour r' au début de chaque appel récursif $\texttt{DepthFirst}(r')$, afin de pouvoir appliquer l'hypothèse d'induction à chaque appel récursif.

On montre d'abord $\mathcal{P}_3(r')$ au début du premier appel récursif. Pour tout x' avec $\texttt{nonZero}[x'] = \texttt{true}$ en début de fonction, par $\mathcal{P}_3(r)$ on a un r'' avec un $\text{ZeroCheck}(r'', x')$ qui domine strictement r. Or, ce dernier domine strictement chacun de ses successeurs, en particulier r', donc, par transitivité de la dominance stricte, pour ces x', $\mathcal{P}_3(r')$ est vérifiée au premier appel récursif sur un successeur. Il reste le cas $x' = x$ avec $\texttt{nonZero}[x] \neq \texttt{true}$ en début d'appel, mais où r a un $\text{ZeroCheck}(r, x)$, et qu'on effectue donc par la suite $\texttt{nonZero}[x] \leftarrow \texttt{true}$ (ligne 9) : dans ce cas, c'est r lui-même qui domine strictement le premier successeur r' et vérifie les hypothèses de $\mathcal{P}_3(r')$ pour x.

Comme chaque appel récursif se fait sur des fils directs de r, la préservation de $\mathcal{P}_3(r')$ pour un successeur r' par un appel récursif (voir la remarque plus haut sur le fait qu'il suffit de prouver $\mathcal{P}_3$ en début d'appel) permet d'assurer $\mathcal{P}_3$ en entrée de l'appel récursif pour le successeur suivant du fait de la propriété suivante : si r'' domine strictement un successeur r' de r, il domine aussi strictement n'importe quel autre de ses successeurs. □

La réciproque du lemme précédent est vraie également : si un nœud de région avec un ZeroCheck pour nœud de contrôle satisfait notre critère syntaxique de redondance, alors l'analyse le détecte. Par exemple, dans le graphe de la figure 6.1a, on obtient $\texttt{redundantZC}[r_3] = \texttt{true}$. Plus formellement :

Lemme 6.1.6. *Soit* `redundantZC = Analyse(`g`)`. *Si* r *a un* ZeroCheck *redondant, alors* `redundantZC[`r`] = true`.

Démonstration. On appelle $\mathcal{P}_1$ la même propriété que dans la démonstration précédente. On a besoin aussi de la propriété :

- $\mathcal{P}_2(r)$: Pour tout r'' et tout r, si `redundantZC[`r''`]` est modifié par un appel `DepthFirst(`r`)`, alors la seule nouvelle valeur possible est `true`.

Cette propriété découle du fait que pour un r donné, `redundantZC[`r`]`, initialement `false`, n'est modifié potentiellement qu'une seule fois lors de l'unique appel `DepthFirst(`r`)`, avant les appels sur les successeurs.

On utilise également l'invariant suivant :

- $\mathcal{P}_3(r)$: Lors de l'appel `DepthFirst(`r`)`, on a déjà un appel récursif (en cours) sur tous les dominateurs de r.

Ces invariants nous permettent de déduire la propriété voulue. Lorsqu'on fait un appel récursif sur un nœud de région r ayant un ZeroCheck(r, x) redondant, on sait qu'il existe un nœud de région r'' dominant strictement r dont le nœud de contrôle est aussi de la forme ZeroCheck(r'', x). D'après l'invariant $\mathcal{P}_3(r)$, on a déjà fait un appel récursif sur r''. Au cours de cet appel, on a l'affectation `nonZero[`x`]` $\leftarrow$ `true` (ligne 9). D'une part, la propriété $\mathcal{P}_1$ assure qu'un appel terminé sur un descendant de r'' n'a pas modifié `nonZero[`x`]`. D'autre part, si un appel encore en cours a modifié `nonZero[`x`]`, c'est en lui donnant la même valeur `true`, de par la nature de l'affectation (ligne 9). Dans tous les cas, on a du coup `nonZero[`x`] = true` lors de l'appel sur r. Ceci permet de conclure à l'aide de l'affectation à `redundantZC[`r`]` de la valeur `nonZero[`x`]` (ligne 8), et de la propriété $\mathcal{P}_2$ qui assure que la valeur de `redundantZC[`r`]` n'est pas remise à `false` ultérieurement. □

Remarque. Les deux lemmes précédents ne sont pas des propriétés sémantiques de la transformation et n'ont pas fait l'objet d'une preuve Coq. Une preuve directe en Coq n'est pas possible, du fait de l'utilisation de structures de données impératives dans l'algorithme. Une solution possible pour formaliser l'algorithme en Coq serait d'écrire une version purement fonctionnelle avec des structures de données purement fonctionnelles, introduisant un facteur logarithmique en plus dans la complexité. Une autre approche, plus facile et n'introduisant pas de changement dans l'algorithme, serait de valider a posteriori, par un validateur prouvé, le résultat de l'analyse : il suffirait essentiellement de disposer pour cela d'un test de dominance formellement prouvé, comme celui développé par exemple dans le cadre du projet CompCertSSA.

6.1.4 Élimination de ZeroCheck : correction sémantique

Soit `redundantZC = Analyse`(g). L'optimisation d'élimination de ZeroCheck redondants procède en remplaçant, dans le graphe g, tout ZeroCheck(r, x) tel que `redundantZC`$[r]$ `= true` avec un simple nœud Jmp(r). On note g' le graphe résultant de cette transformation. Dans le cas de la figure 6.1a, l'analyse donne `redundantZC`$[r_3]$ `= true`, donc g' s'obtient à partir de g en changeant le nœud c_3 par un Jmp(r_3).

Nous prouvons que cette transformation est correcte sémantiquement. En d'autres termes, que le graphe initial et le graphe transformé ont la même sémantique. On écrit $\rightarrow_{\text{STEP}_g}$ pour désigner un pas d'exécution dans le graphe g. Le théorème que nous prouvons est le suivant :

Théorème 6.1.7. *On a* Start $\cdots \rightarrow_{STEP_g}$ Ret(v) *si, et seulement si,* Start $\cdots \rightarrow_{STEP_{g'}}$ Ret(v).

Par exemple, par rapport à la figure 6.1a, le théorème assure que, ou bien tant l'exécution de g comme celle du graphe transformé g' (où c_3 est devenu un Jmp(r_3)) ne terminent pas, ou bien les deux terminent et retournent la même valeur. En particulier, si l'exécution pour g ne retourne pas Ret(`retfail`), alors celle pour g' non plus.

Nous prouvons le théorème ci-avant en établissant une relation de simulation entre les deux graphes. Dans le cas de l'élimination de ZeroCheck redondants, le lien entre les deux états d'exécution est fort : les états sont tout simplement égaux à chaque pas d'exécution.

Étant donné un état σ, le prédicat reachable$_g(\sigma)$ signifie qu'il existe un chemin d'exécution depuis Start jusqu'à σ dans g.

Lemme 6.1.8. *Soit σ un état d'exécution tel que* reachable$_g(\sigma)$. *Alors, pour tout σ', on a :*

$$\sigma \rightarrow_{STEP_g} \sigma' \textit{ si, et seulement si, } \sigma \rightarrow_{STEP_{g'}} \sigma'.$$

Démonstration. Le seul cas subtil à traiter est celui où $\sigma = \text{Exec}(r, \rho)$ et le nœud de contrôle de r dans g est un ZeroCheck(r, x) qui a été optimisé en un Jmp(r) dans g'.

S'il a été optimisé, alors c'est que `redundantZC`$[r]$ `= true`. Le lemme 6.1.5 assure qu'il existe r' qui domine strictement r dont le nœud de contrôle est un ZeroCheck(r', x).

Supposons que $\sigma \rightarrow_{\text{STEP}_g} \sigma'$. On sait que σ est un état atteignable, donc on peut appliquer le corollaire 6.1.2. Du coup, $[\![x]\!]_\rho = v$ avec $v \neq 0$, et le ZeroCheck passe (la règle applicable est ZCHK1), et g' peut faire un pas correspondant en exécutant son nœud Jmp, qui a le même successeur.

Maintenant, si $\sigma \rightarrow_{\text{STEP}_{g'}} \sigma'$ via le nœud Jmp, on peut faire un pas correspondant dans g à l'aide de la règle ZCHK1, que l'on sait applicable (par le corollaire 6.1.2, puisque σ est atteignable dans g). □

Remarquons que le lemme n'est pas totalement symétrique : l'hypothèse $\text{reachable}_g(\sigma)$ se fait uniquement sur le graphe g. Ceci est dû au fait que l'on applique dans les deux cas le corollaire 6.1.2 pour g dans la preuve. Notons que ceci n'a pas d'incidence pour en déduire le théorème par induction sur les chemins d'exécution, car dans les deux sens l'hypothèse d'induction donne l'atteignabilité dans les graphes g et g' jusqu'à un même état σ et le lemme ci-dessus permet de l'assurer après un nouveau pas (dans g ou g') jusqu'à un même état σ' pour g et g'.

Remarquons également qu'à aucun moment on n'utilise le lemme 5.2.5 pour assurer l'évaluabilité d'un nœud de données. En effet, à chaque fois on prouve une implication, donc on a l'existence d'un pas pour l'un ou l'autre des programmes. L'existence de ce pas donne l'évaluabilité des nœuds de données mis en jeu dans ce pas sémantique pour l'un des programme dans son environnement, et les invariants nous permettent de déduire l'évaluabilité pour l'autre programme si besoin. Ce phénomène est récurrent dans toutes les optimisations pour *Sea of Nodes* que l'on traite dans ce manuscrit.

6.2 Propagation de constantes simple

Dans cette section, on définit une transformation de propagation simple de constantes creuse, accompagnée d'une preuve de préservation sémantique. Dans le contexte de *Sea of Nodes*, l'analyse présentée s'étend naturellement, sous forme d'analyse combinée [20], au cas de la propagation conditionnelle de constantes creuse qui est traitée dans la section 6.3.

Les résultats sémantiques concernant la transformation de propagation simple de constantes creuse décrits dans cette section ont fait l'objet d'un développement Coq, contrairement aux résultats concernant la propagation conditionnelle de constantes creuse.

On note dans cette section $\mathtt{num}$ l'ensemble des valeurs numériques que peut prendre un nœud de données.

6.2.1 Analyse de flot de données pour constantes

On utilise le treillis $L = \mathtt{num} \cup \{\bot, \top\}$ à trois niveaux suivant : $\bot$ correspond au cas où un nœud n'a pas de valeur, un N de type $\mathtt{num}$ à une constante, et $\top$ à une valeur non constante pour l'analyse. L'ordre $<$ sur L est tel que $\bot < N < \top$ pour toute constante N de L. Les éléments de $\mathtt{num}$ ne sont pas comparables entre eux dans L.

On définit sur L la fonction $\sqcup : L \to L \to L$ correspondant à la borne supérieure de deux éléments de L par le tableau suivant :

$\sqcup$	$\bot$	N_0	$\top$
$\bot$	$\bot$	N_0	$\top$
N_1	N_1	si $N_0 = N_1$ alors N_0 sinon $\top$	$\top$
$\top$	$\top$	$\top$	$\top$

L est un treillis et, en particulier, cet opérateur est associatif. On peut donc écrire $v_1 \sqcup \cdots \sqcup v_m$.

L'analyse produit une fonction d'analyse $A : \mathrm{id} \to L$ qui, à chaque nœud de données associe une valeur dans le treillis. Une telle fonction peut être représentée efficacement par un tableau.

Pour tout nœud de données x, on définit une fonction de transfert φ_x qui prend une fonction d'analyse en entrée et renvoie une valeur actualisée dans le treillis, qui correspond à une propagation de l'analyse vers x. L'algorithme itère toutes les fonctions de transfert sur le résultat de l'analyse, en partant d'une fonction d'analyse A_0 telle que pour tout x, $A_0(x) = \bot$. Une solution s'obtient par un point fixe A de toutes les équations $\varphi_x(A) = A(x)$. Si toutes les fonctions φ_x sont monotones, un tel point fixe existe et est atteint avec une stratégie standard d'itération d'équations *data-flow*.

Pour un nœud de données x de nature *Nœud* dépendant des nœuds de données $x_1, \ldots, x_m$, la forme générale des fonctions de transfert est

$$\varphi_x(A) = \mathrm{op}^{\#}_{Nœud}(A(x_1), \ldots, A(x_m)),$$

où $\mathrm{op}^{\#}_{Nœud}$ est une abstraction de l'opération associée au nœud x, à valeurs dans L. On définit $\mathrm{op}^{\#}_{Nœud}$ comme suit en fonction de *Nœud* (et de l'arité).

Pour les constantes, on a :

$$\mathrm{op}^{\#}_{\mathrm{Cst}_N}() = N$$

Pour les opérations arithmétiques binaires :

$$\mathrm{op}^{\#}_{binop}(v_1, v_2) = \begin{cases} \top & \text{si } \exists i, v_i = \top \\ \bot & \text{si } \exists i, v_i = \bot \\ \mathrm{op}_{binop}[v_1, v_2] & \text{sinon (dans num)} \end{cases}$$

Pour un nœud Phi de la forme $g(x) = \mathrm{Phi}(r, x_1, \ldots, x_m)$, on utilise :

$$\mathrm{op}^{\#}_{\mathrm{Phi}}(v_1, \ldots, v_m) = v_1 \sqcup \cdots \sqcup v_m.$$

6.2.2 Monotonie

La propriété de monotonie à prouver est la suivante :

Théorème 6.2.1. *Pour tout x et pour toutes fonctions d'analyse A et A' :*

$$\forall y, A(y) \leq A'(y) \implies \varphi_x(A) \leq \varphi_x(A').$$

Démonstration. On distingue trois cas suivant la nature du nœud x.
Si $g(x) = \text{Cst}_N$, alors $\varphi_x(A) = N = \varphi_x(A')$.
Si $g(x) = binop(x_1, x_2)$:

- Si $\varphi_x(A') = \top$, c'est immédiat.
- Si $\varphi_x(A') = N$, alors pour tout $i = 1, 2$, il existe N_i tel que $A'(x_i) = N_i$, donc $A(x_i) \leq N_i$. Donc soit il existe i tel que $A(x_i) = \bot$, et alors $\varphi_x(A) = \bot$, soit pour tout i, $A(x_i) = N_i$, et $\varphi_x(A) = N$. Dans les deux cas, l'inégalité est vérifiée.
- Si $\varphi_x(A') = \bot$, alors il existe i tel que $A'(x_i) = \bot$, et comme $A(x_i) \leq A'(x_i)$, on a $A(x_i) = \bot$ aussi, et $\varphi_x(A) = \bot$.

Si $g(x) = \text{Phi}(r, x_1, \ldots, x_m)$, l'inégalité résulte de la monotonie de $\sqcup$ (à un argument fixé). □

6.2.3 Algorithme

Un algorithme standard [51] pour résoudre les systèmes d'équations de flot de données utilise un ensemble de travail W. Initialement, il contient l'ensemble des nœuds de données du graphe g. Tant que W n'est pas vide :

- On retire un élément x dans W.
- On réalise l'affectation $A(x) \leftarrow \varphi_x(A)$.
- Si $A(x)$ a changé, on rajoute dans W tous les y dépendant de x (s'ils n'y sont pas déjà).

Quand W est vide, c'est que A est le plus petit point fixe. Pour des raisons de monotonie, il est classique de remarquer qu'affecter $\varphi_x(A)$ à $A(x)$ donne le même résultat qu'une utilisation plus naïve de l'affectation plus générale $A(x) \sqcup \varphi_x(A)$, grâce à l'invariant $\varphi_x(A) \geq A(x)$ pour tout x.

Exemple. Considérons l'exemple en pseudo-code de la figure 6.3. L'ensemble d'équations associées à chaque nœud de ce programme est le suivant :

Nœud	Équation
x_0	$A(x_0) = 1$
y_0	$A(y_0) = 1$
z_0	$A(z_0) = 2$
y_1	$A(y_1) = A(y_0) \sqcup A(y_2)$
x_1	$A(x_1) = \text{op}^\#_\times(A(x_0), A(z_0))$
y_2	$A(y_2) = \text{op}^\#_-(A(x_1), A(y_1))$

```
1  // bloc r0
2  x0 ← 1
3  y0 ← 1
4  z0 ← 2
5  do {
6      // bloc r1
7      y1 ← Phi(y0, y2)
8      x1 ← x0 × z0
9      y2 ← x1 - y1
10 } while ...
11 // bloc r2
12 ...
```

FIG. 6.3 : Pseudo-code de programme à analyser

Voici ci-dessous un tableau correspondant à une stratégie d'itération. Chaque ligne représente l'approximation par l'analyse pour un nœud, et la k-ième colonne donne l'approximation donnée par l'analyse à la k-ième itération utilisant l'équation associée à un nœud choisi dans W. Dans la première ligne de chaque colonne est écrit le nom du nœud associé à l'équation choisie pour itérer.

	init	x_0	y_1	y_0	z_0	x_1	y_2	y_1	y_2
x_0	$\bot$	1	1	1	1	1	1	1	1
y_0	$\bot$	$\bot$	$\bot$	1	1	1	1	1	1
z_0	$\bot$	$\bot$	$\bot$	$\bot$	2	2	2	2	2
x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	2	2	2	2
y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1	1
y_2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1

Après la dernière itération, le résultat de l'analyse est stable par toutes les équations.

6.2.4 Conséquences des équations de point fixe

On suppose dorénavant que A est un point fixe de l'analyse, c'est-à-dire que pour tout x, $\varphi_x(A) = A(x)$. En développant, on obtient :

- Si $g(x) = \mathrm{Cst}_N$, alors

$$A(x) = N.$$

- Si $g(x) = \mathit{binop}(x_1, x_2)$, alors

$$A(x) = N \Leftrightarrow \left(\forall i, \exists N_i, A(x_i) = N_i\right) \wedge N = \mathrm{op}_{binop}(A(x_1), A(x_2)).$$

- Si $g(x) = \mathrm{Phi}(r, x_1, \ldots, x_m)$, alors

$$A(x) = N \Leftrightarrow \exists i, A(x_i) = N \wedge \forall j, A(x_j) \in \{\bot, N\}.$$

Ces relations et équations sont utiles dans la preuve.

Remarque. En fait pour le résultat de cette analyse de constantes simple, pour tout y, on a $A(y) \neq \bot$ du fait des propriétés de bonne formation des dépendances de données. Intuitivement, l'existence de y tel que $A(y) = \bot$ permettrait d'exhiber un cycle de dépendances qui contredirait la bonne formation des dépendances de données. Le développement Coq fait usage de cette propriété pour des raisons pratiques pour simplifier la preuve. Dans le manuscrit, cependant, on ne va pas utiliser cette propriété dans la suite, car cela permettra d'étendre plus facilement au cas de la propagation conditionnelle de constantes dans la section suivante.

6.2.5 La transformation et sa spécification formelle

On définit le graphe transformé g' du graphe g par la formule suivante s'appliquant aux nœuds de données :

$$g'(x) = \begin{cases} \mathrm{Cst}_N & \text{si } A(x) = N \\ g(x) & \text{sinon.} \end{cases}$$

Sur les nœuds qui ne sont pas de données les deux graphes coïncident.

Préservation des identifiants de nœuds. Tout comme pour l'élimination de ZeroCheck redondants, la transformation ne fait que changer la nature de certains nœuds, en l'occurrence certains nœuds de données deviennent des nœuds Cst. On conserve donc le même ensemble d'identifiants de nœuds dans g', puisque seule la nature de certains nœuds a changé par rapport à g. Dans le cas de la propagation de constantes, certains nœuds peuvent devenir alors du code mort (plus aucun nœud n'en dépend) : une passe postérieure d'élimination de code mort, que nous ne traitons pas dans ce manuscrit, doit donc être réalisée.

Remarquons qu'en particulier, le CFG sous-jacent à g et g' est le même.

6.2.6 Préservation sémantique

Nous montrons la préservation sémantique de cette transformation pour la sémantique $\rightarrow_{\text{STEP}}$ par une simulation. L'analyse faite dans ce chapitre étant creuse, le cadre général sur les analyses de flot de données tel que l'on peut le retrouver par exemple dans le livre de Nielson et al. [51] ne s'applique pas directement. Une adaptation générique de l'état de l'art au cas des techniques d'analyses creuses dépassant les objectifs de formalisation de ce manuscrit, nous avons préféré opter pour une preuve plus ad hoc.

On définit un invariant $\sim$ sur les états d'exécution de g et g' ainsi :

- on a Start $\sim$ Start ;
- on a $\text{Ret}(v) \sim \text{Ret}(v)$ pour tout v ;
- on a $\text{Exec}(r, \rho) \sim \text{Exec}(r, \rho')$ si trois conditions sont vérifiées :

  CPMATCH1 : pour tout x, si $[\![x]\!]_{\rho,g} = v$ (évaluation dans le graphe g) alors $[\![x]\!]_{\rho',g'} = v$.

  CPMATCH2 : pour tout nœud ϕ on a la propriété $\mathcal{P}_{\phi,\rho}$: si $A(\phi) = N$ pour une certaine constante N, et si $[\![\phi]\!]_{\rho,g}$ est défini, alors on a $[\![\phi]\!]_{\rho,g} = N$.

  CPMATCH3 : pour tout ϕ, si $[\![\phi]\!]_{\rho,g}$ est défini, alors $A(\phi) \neq \bot$.

Un point à noter, c'est que les propriétés concernant les nœuds Phi CPMATCH2 et CPMATCH3 impliquent les mêmes propriétés pour les nœuds de données en général. C'est un résultat intuitif, puisque la connaissance de la valeur des nœuds Phi dans l'environnement suffit à évaluer n'importe quel autre nœud de données. Les deux lemmes suivants formalisent ce résultat.

Lemme 6.2.2. *Soit ρ un environnement. Supposons que, pour tout nœud ϕ, la propriété $\mathcal{P}_{\phi,\rho}$ est vraie. Alors, pour tout nœud x, la propriété $\mathcal{P}_{x,\rho}$ est vraie.*

Démonstration. La propriété se démontre facilement par induction sur la définition d'évaluation d'un nœud de données x. Si x est un nœud de constante, c'est immédiat. Si x est un nœud $binop(x_1, x_2)$, alors il existe N_1, N_2 tels que $A(x_1) = N_1$ et $A(x_2) = N_2$ avec $N = \text{op}_{binop}(N_1, N_2)$. De plus, comme $[\![x]\!]_{\rho,g}$ est défini, on a aussi $[\![x_1]\!]_{\rho,g}$ et $[\![x_2]\!]_{\rho,g}$ qui sont définis. On peut ainsi appliquer l'hypothèse d'induction à x_1 et x_2, donc on peut conclure. Si x est un nœud Phi, la propriété $\mathcal{P}_{x,\rho}$ est vraie par hypothèse. □

Lemme 6.2.3. *Supposons que pour tout ϕ, si $[\![\phi]\!]_{\rho,g}$ est défini, alors $A(\phi) \neq \bot$. Alors, pour tout x, si $[\![x]\!]_{\rho,g}$ est défini, alors $A(x) \neq \bot$.*

Démonstration. La propriété se démontre similairement par induction sur la définition d'évaluation d'un nœud de données x. Le cas d'une constante

est immédiat, et celui d'un nœud Phi est vrai par hypothèse. Il reste donc à traiter le cas d'un nœud $binop(x_1, x_2)$. De façon analogue au lemme précédent, on peut appliquer l'hypothèse d'induction à x_1 et x_2. L'équation de point fixe pour le cas d'un nœud *binop* permet de conclure. □

Théorème 6.2.4. *Soient σ_g et $\sigma_{g'}$ des états d'exécution tels que $\sigma_g \sim \sigma_{g'}$. Alors, pour tout σ'_g, on a : si $\sigma_g \rightarrow_{STEP_g} \sigma'_g$ alors $\sigma'_g \rightarrow_{STEP_{g'}} \sigma'_{g'}$ où $\sigma'_{g'}$ est tel que $\sigma'_g \sim \sigma'_{g'}$.*

Démonstration. La preuve se fait par étude de cas sur $\rightarrow_{\text{STEP}_g}$.

Le cas où σ_g est Start s'obtient facilement, puisque l'environnement initial est vide. L'invariant CPMATCH1 s'obtient par induction sur la définition d'évaluation dans le cas d'un environnement vide : intuitivement, les seuls nœuds évaluables à ce moment sont ceux qui ne dépendent d'aucun nœud Phi et vont en fait correspondre à des nœuds de données constants pour l'analyse. Les invariants CPMATCH2 et CPMATCH3 sont, eux, immédiats avec un environnement vide.

Le cas où σ'_g est $\text{Ret}(v)$ découle de l'invariant CPMATCH1 appliqué au nœud de données dont dépend le Return.

Dans les autres cas, $\sigma_g = \text{Exec}(r, \rho_1)$ et $\sigma'_g = \text{Exec}(r', \rho_2)$ et le point subtil est de démontrer qu'après chaque application de la règle PHIS, les invariants sont préservés.

Montrons la préservation de CPMATCH2, c'est-à-dire que $\mathcal{P}_{\phi,\rho_2}$ pour tout ϕ. Si ϕ ne dépend pas de r', on a $\rho_1(\phi) = \rho_2(\phi)$, donc c'est immédiat. Sinon, on a donc un ϕ avec $A(\phi) = N$, et $\rho_2(\phi)$ est défini par la valeur actualisée par la règle PHIS, c'est-à-dire $[\![x_i]\!]_{\rho_1,g}$ pour une certaine entrée x_i de ϕ. Par CPMATCH3 et le lemme 6.2.3, on sait que $A(x_i) \neq \bot$, donc en fait $A(x_i) = N$ par l'équation de point fixe. Remarquons maintenant que, grâce au lemme 6.2.2, CPMATCH2 implique $\mathcal{P}_{x,\rho_1}$ pour tout x avec l'environnement ρ_1. On peut appliquer alors simplement la propriété $\mathcal{P}_{x_i,\rho_1}$ pour conclure.

Ceci va nous permettre de prouver la préservation de CPMATCH1. Nous prouvons cette préservation d'abord pour les nœuds ϕ de g', c'est-à-dire que pour tout ϕ, si $[\![\phi]\!]_{\rho_2,g} = v$ pour un certain v, alors $[\![\phi]\!]_{\rho'_2,g'} = v$. Soit donc ϕ un nœud $\text{Phi}(r'', x_1, \ldots, x_m)$ dans g'. Si ϕ ne dépend pas de r' (c'est-à-dire $r'' \neq r'$), alors sa valeur dans l'environnement ne change ni pour ρ_1 ni pour ρ'_1 et la préservation est immédiate. Sinon, la préservation découle de l'invariant CPMATCH1 (pour les environnements ρ_1 et ρ'_1) appliqué à l'entrée x_i de ϕ qui permet d'actualiser la valeur de l'environnement pour ϕ dans la règle PHIS.

On va maintenant montrer la préservation de CPMATCH1 dans le cas général pour un nœud de données quelconque. On procède par induction sur la définition de l'évaluation d'un nœud de données x. Si x est un nœud de constante, c'est immédiat. Si x est un nœud Phi dans g et g', cela découle de la préservation de CPMATCH1 pour les nœuds ϕ. Si x est un nœud Phi dans

g, mais un Cst_N dans g', alors $A(x) = N$, et on applique $\mathcal{P}_{x,\rho_2}$ (préservation de CPMATCH2 et lemme 6.2.2), qui assure $[\![x]\!]_{\rho_2,g} = N$ également. Si x est un nœud *binop* dans g et g', l'hypothèse d'induction permet de conclure. Si x est un nœud *binop* dans g et un Cst_N dans g', alors $A(x) = N$, et on applique $\mathcal{P}_{x,\rho_2}$, qui assure $[\![x]\!]_{\rho_2,g} = N$ également.

Montrons, enfin, la préservation de CPMATCH3. Soit ϕ un nœud $\text{Phi}(r'', x_1, \ldots, x_m)$ tel que $[\![\phi]\!]_{\rho_2,g}$ est défini. Si $r'' \neq r'$, alors $[\![\phi]\!]_{\rho_1,g}$ était déjà défini et donc $A(\phi) \neq \bot$ par CPMATCH3. Sinon, $[\![\phi]\!]_{\rho_2,g} = [\![x_i]\!]_{\rho_1,g}$ pour un certain x_i, auquel on peut appliquer le lemme 6.2.3, donc $A(x_i) \neq \bot$ et, par conséquent, $A(\phi) \neq \bot$ également d'après l'équation de point fixe. □

Discussion. Comme mentionné à la section 3.3, des arguments standard utilisant le déterminisme du langage permettent de déduire l'équivalence sémantique à partir de ce théorème [46]. Dans le cas de l'analyse de constantes, il est plus pratique de recourir à ces arguments plutôt que de prouver directement une équivalence comme on l'a fait dans le cas de l'élimination de ZeroCheck redondants.

6.3 Propagation conditionnelle de constantes

L'analyse simple de constantes de la section précédente ne tient pas compte du fait que certaines branches soient exécutables ou non. Si la condition d'un If est constante, la même branche est toujours prise et, en prenant ceci en compte, on obtient plus d'information pour l'analyse et on trouve plus de constantes. Plus précisément, cela signifie en particulier que la valeur d'un Phi pourrait être ainsi prouvée constante, même si toutes les entrées ne sont pas constantes pour l'analyse simple de constantes.

Par exemple, considérons l'exemple en pseudo-code suivant :

```
1     // bloc r_0
2     x_0 ← 2
3     y ← false
4     if y {
5         // bloc r_1
6         x_1 ← 3
7         ...
8     }
9     // bloc r_2
10    x_2 ← Phi(x_0, x_1)
```

On a $A(x_0) = 2$ et $A(x_1) = 3 \neq 2$, et l'analyse simple de constantes donne donc $A(x_2) = \top$, même si x_1 n'est en fait jamais évalué et que l'évaluation de x_2 donnera toujours 2. Une analyse conditionnelle de constantes permet de détecter ce genre de cas.

6.3.1 Analyse

Avec une représentation *Sea of Nodes*, une analyse conditionnelle de constantes creuse peut s'exprimer naturellement sous forme d'une analyse issue d'une combination de l'analyse de propagation de constantes simple et d'une analyse de code mort [20]. En particulier, contrairement à des approches plus classiques [71], cette version de la propagation conditionnelle de constantes n'étiquette pas les arcs avec de l'information d'exécutabilité, mais seulement les nœuds. On donne dans la suite les principales adaptations à faire.

L'analyse étendue utilise les mêmes fonctions de transfert et les mêmes valeurs de treillis $v \in L$ pour les nœuds de données considérés à la section précédente, à une modification près de la fonction de transfert pour les Phi, que l'on va détailler plus bas. Par contre, on doit donner aussi des valeurs aux nœuds Region, If, Jmp et aux projections. Pour ceux-ci, l'analyse peut donner les valeurs $\bot$ (non exécutable) ou $\top$ (exécutable). L'analyse peut aussi donner la valeur `false` ou `true` à un nœud If, correspondant à la valeur d'analyse de la condition dans le cas où elle est constante, ce qui correspond au cas où uniquement une des branches peut être prise. Pour le If, $\top$ signifie que pour l'analyse les deux branches sont exécutables. On utilise donc le même treillis L que précédemment, mais on étend les fonctions de transfert à de nouveaux nœuds.

Remarquons que, suivant la nature du nœud, $\bot$ et $\top$ n'ont pas exactement la même signification : par exemple, pour un nœud de contrôle, $\top$ signifie que le nœud est exécutable pour l'analyse, alors que pour un nœud de données cela signifie que le nœud n'est pas constant pour l'analyse. On fusionne donc en fait deux treillis différents, mais cela ne pose pas de problème, car chaque sorte de nœud ne prend ses valeurs que dans un des treillis.

On rappelle que, pour un nœud n dépendant de $n_1, \ldots, n_m$ et de forme *Nœud*, la forme générale des fonctions de transfert est

$$\varphi_n(A) = \mathrm{op}^{\#}_{Nœud}(A(n_1), \ldots, A(n_m)),$$

où $\mathrm{op}^{\#}_{Nœud}$ est une abstraction, au sens de la propagation conditionnelle de constantes, associée au nœud n, à valeurs dans L. On définit $\mathrm{op}^{\#}_{Nœud}$ pour les autres nœuds et les Phi comme suit :

- Pour $g(n) = \mathrm{Region}(p_1, \ldots, p_m)$, on utilise :

  $$\mathrm{op}^{\#}_{\mathrm{Region}}(v_1, \ldots, v_m) = v_1 \sqcup \cdots \sqcup v_m$$

  En d'autres termes, un nœud de région est potentiellement exécutable si au moins l'un de ses prédécesseurs l'est.

  Dans le cas où $m = 0$ (aucun prédécesseur), si $\mathrm{entry}(f) = n$ alors $\mathrm{op}^{\#}_{\mathrm{Region}}() = \top$, sinon $\mathrm{op}^{\#}_{\mathrm{Region}}() = \bot$ (code mort). *Remarque :* on rap-

pelle que le nœud initial n'a pas de prédécesseurs pour une fonction *Sea of Nodes* bien formée.

- Pour $g(n) = \text{Jmp}(r)$, on utilise :

$$\text{op}^{\#}_{\text{Jmp}}(v) = v.$$

Il s'agit donc simplement d'une copie de la valeur de l'analyse en r : le Jmp est évalué si r l'est.

- Pour $g(n) = \text{If}(r, x)$, on utilise :

$$\text{op}^{\#}_{\text{If}}(v_1, v_2) = A(v_1) \sqcap A(v_2).$$

où $\sqcap$ est l'opérateur de borne inférieure défini par le tableau suivant :

$\sqcap$	$\bot$	N_0	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
N_1	$\bot$	si $N_0 = N_1$ alors N_0 sinon $\bot$	N_1
$\top$	$\bot$	N_0	$\top$

La valeur calculée est donc celle de la condition dans le cas où r est exécutable, et $\bot$ sinon. Notons qu'une région ne prend que ces deux valeurs par l'analyse, donc la deuxième ligne du tableau n'est pas utilisée.

- Pour $g(n) = \text{IfF}(if)$, on utilise :

$$\text{op}^{\#}_{\text{IfF}}(v) = \begin{cases} \top & \text{si } A(v) \in \{\top, \texttt{false}\} \\ \bot & \text{sinon} \end{cases}$$

- Pour $g(n) = \text{IfT}(if)$, on utilise :

$$\text{op}^{\#}_{\text{IfT}}(v) = \begin{cases} \top & \text{si } A(v) \in \{\top, \texttt{true}\} \\ \bot & \text{sinon} \end{cases}$$

- Enfin, pour $g(n) = \text{Phi}(r, x_1, \ldots, x_m)$, il faut considérer la région, $g(r) = \text{Region}(p_1, \ldots, p_m)$. On utilise :

$$\text{op}^{\#}_{\text{Phi}}(w_1, \ldots, w_m, v_1, \ldots, v_m) = (w_1 \sqcap v_1) \sqcup \cdots \sqcup (w_m \sqcap v_m)$$

où $w_1, \ldots, w_m$ correspondent aux valeurs données par l'analyse pour $p_1, \ldots, p_m$. On prend donc maintenant en compte la valeur donnée par l'analyse aux prédécesseurs du nœud de région r. Ainsi, dans le calcul, on ne considérera une entrée x_i que lorsque l'entrée correspondante du nœud de région a une valeur w_i qui est $\top$.

6.3.2 Monotonie et algorithme

Il n'y a pas de changements notables pour ce qui est de la preuve de monotonie par rapport à l'analyse de constantes simple, ni dans l'algorithme de l'analyse en soi.

6.3.3 Conséquences des équations de point fixe

Les relation précédentes restent les mêmes, sauf pour les Phi, et on a quelques nouvelles relations utiles pour la preuve :

- Si $g(n) = \text{Region}(p_1, \ldots, p_m)$, alors $A(n) = A(p_1) \sqcup \cdots \sqcup A(p_m)$.
- Si $g(n) = \text{Jmp}(r)$, alors $A(n) = A(r)$.
- Si $g(n) = \text{If}(r, x)$, alors $A(n) = A(r) \sqcap A(x)$.
- Si $g(n) = \text{IfT}(if)$, alors :

$$A(n) = \top \Leftrightarrow A(if) \in \{\top, \texttt{true}\}.$$

- Si $g(n) = \text{IfF}(if)$, alors :

$$A(n) = \top \Leftrightarrow A(if) \in \{\top, \texttt{false}\}.$$

- Si $g(n) = \text{Phi}(r, x_1, \ldots, x_m)$ et $g(r) = \text{Region}(p_1, \ldots, p_m)$, alors :

$$A(n) = N \Leftrightarrow \begin{cases} \exists i, A(x_i) = N, A(p_i) = \top \\ \\ \wedge \forall j, (A(p_j) \neq \top \ \vee \ A(x_j) \in \{\bot, N\}). \end{cases}$$

6.3.4 Spécification de la transformation

La transformation reste globalement la même. Notons que les nœuds If peuvent avoir une valeur constante booléenne, mais on ne veut pas que la transformation les modifie. La transformation ne change potentiellement que des nœuds de données, comme précédemment, même si l'analyse tient compte de tous les nœuds cette fois-ci.

Remarquons en particulier, que cela signifie que les branches non exécutables ne sont pas supprimées par l'algorithme et qu'elles doivent être ensuite éliminées séparément par une transformation ultérieure.

6.3.5 Préservation sémantique

L'énoncé des lemmes et théorèmes reste le même que pour la propagation simple de constantes. Il faut par contre ajouter un invariant en plus pour $\text{Exec}(r, \rho) \sim \text{Exec}(r, \rho')$:

CPMATCH4 : $A(r) = \top$.

En d'autres termes, le nœud de région est exécutable pour l'analyse.

On ne détaille pas de nouveau les preuves, mais on donne l'intuition des points où les arguments doivent être actualisés.

La nouvelle condition dans l'invariant est utile pour raisonner sur les Phis, la valeur que leur donne l'analyse ne coïncidant avec le *join* des entrées que dans le cas où le bloc est exécutable, c'est-à-dire $A(r) = \top$, ou si toutes les entrées sont à $\bot$. En particulier, dans la preuve de préservation de l'invariant CPMATCH3, pour prouver $A(\phi) \neq \bot$, où $g(\phi) = \text{Phi}(r', p_1, \ldots, p_m)$, le fait qu'il existe une entrée telle que $A(x_i) = N$ ne suffit plus, on a maintenant aussi besoin de $A(p_i) = \top$ pour conclure que $A(\phi) = N$. Les équations de point fixe pour les nœuds de contrôle et de projection permettent de voir que cela revient à ce que l'analyse donne $\top$ pour le nœud de région précédant r dans le CFG, ce qui découle du nouvel invariant CPMATCH4. Dans le cas d'un nœud If, il faut appliquer également le lemme 6.2.3 au nœud de données de condition pour pouvoir conclure.

Il suffit, du coup, de montrer la préservation du nouvel invariant CPMATCH4 dans la preuve du théorème 6.2.4, c'est-à-dire que pour le successeur r' de r dans le CFG, après avoir effectué un pas dans la sémantique (JMP, IFF ou IFT), on a aussi $A(r') = \top$.

- Dans le cas de la règle JMP, la propagation de la valeur $\top$ de $A(r)$ vers $A(r')$ découle des équations de point fixe par simple copie et *join*.

- Dans le cas d'un nœud If, il y a deux cas symétriques suivant la valeur de la condition. On traite le cas IFT (voir figure 4.5), où on a un nœud *if* de la forme If(r,x) tel que $[\![x]\!]_{\rho_1} = \texttt{true}$. Il suffit de montrer que $A(\mathit{if}) \in \{\top, \texttt{true}\}$. En effet, si tel est le cas, du fait des équations de point fixe c'est que $A(p) = \top$ pour la projection IfT successeur p et donc $A(r') = \top$ également. On sait déjà d'après CPMATCH3 que $A(x) \neq \bot$, puisque $[\![x]\!]_{\rho_1} = \texttt{true}$. Si $A(x) = \top$, alors $A(\mathit{if}) = \top$ par équation de point fixe. Si $A(x) \in \{\texttt{false}, \texttt{true}\}$, on a $g'(x) = \text{Cst}_{A(x)}$ par définition. Comme $[\![x]\!]_{\rho_1} = \texttt{true}$, on a aussi $[\![x]\!]_{\rho_2} = \texttt{true}$ par CPMATCH1, et, par définition de l'évaluation des constantes, on obtient $A(x) = \texttt{true}$. Par équation de point fixe, $A(\mathit{if}) = \texttt{true}$.

6.4 Conclusion

Dans ce chapitre, nous avons abordé deux optimisations : un algorithme d'élimination de *zero-checks* redondants et deux versions de propagation de constantes creuse. L'élimination de *zero-checks* redondants nous a permis de mettre en évidence l'utilité de la propriété fondamentale établie au chapitre 5 et formulée en termes de dominance. La propagation simple de constantes creuse a été l'occasion de tester le raisonnement à l'aide de notre sémantique *Sea of Nodes* sur un exemple d'optimisation classique. La version conditionnelle de la propagation de constantes, vue comme une combination de la propagation simple de constantes et d'une analyse de code mort, est un exemple intéressant d'optimisation tirant parti des caractéristiques particulières de la forme *Sea of Nodes* pour simplifier l'analyse : une simplification qui facilite le raisonnement formel.

Chapitre 7

Sea of Nodes : retour au bloc de base

Avec une représentation *Sea of Nodes*, hormis les Phi, les nœuds de données ne dépendent pas d'un nœud de région en particulier. Cette représentation, pratique pour raisonner sur les optimisations, doit cependant être rendue séquentielle avant la génération de code. Autrement dit, il faut passer de cette représentation en termes de dépendances de données vers une représentation séquentielle des instructions en vue de produire de l'assembleur.

Une première étape est alors de rattacher chaque nœud à un nœud de région. Pour ce faire, il est possible d'utiliser par exemple l'algorithme de *Global Code Motion* dû à Click [19]. L'ensemble des nœuds dépendant d'une région forment alors un bloc non ordonné d'instructions. L'affectation de chaque nœud de données à un nœud de région doit respecter des contraintes de bonne formation, en particulier un nœud de données doit dominer ses usages, c'est-à-dire que son nœud de région doit dominer celui des nœuds qui l'utilisent.

La deuxième étape est alors de séquentialiser les instructions à l'intérieur de chaque bloc en respectant les dépendances. Les dépendances de données doivent être respectées à l'intérieur de chaque bloc : si x et y sont deux nœuds rattachés au même bloc, et x dépend de y dans le graphe *Sea of Nodes*, alors y doit apparaître avant x dans la séquentialisation des instructions de ce bloc.

7.1 Retour au bloc de base non ordonné

Notons m une application qui, à chaque nœud de données, associe un nœud de région (le futur bloc de base), avec la condition que si x dépend d'un bloc r, alors $m(x) = r$. Avec notre forme *Sea of Nodes*, un tel x est un Phi, mais dans une version étendue, x pourrait être un nœud de mémoire ou une instruction fautive, tel que discuté dans la section 4.3 sur les extensions.

Pour simplifier les raisonnements de dominance relatifs à la *strictness*, on étend de plus l'application m aux nœuds de contrôle avec la condition que si c est un nœud de contrôle dépendant du nœud de région r, alors $m(c) = r$. On impose uniquement que m respecte la propriété de *strictness* suivante :

Propriété 7.1.1. *Pour tous nœuds de données ou contrôle n et n' tels que n' dépend de n, on a $m(n)$ qui domine $m(n')$.*

Cette application représente une solution possible pour le problème d'affectation d'un nœud de région à chaque nœud de données. Cette solution pourrait par exemple être le résultat du *Global Code Motion* de Click : cet algorithme est conçu justement de sorte que le résultat vérifie la propriété ci-dessus. Notons cependant qu'une version de l'algorithme avec des heuristiques d'optimisation poussées pourrait, dans certains cas, dupliquer certains nœuds afin de les placer uniquement dans certaines branches auprès de leurs utilisations : nous considérons dans la suite que le calcul de m est issu d'un algorithme qui ne se permet pas ce genre d'optimisation poussée et se contente d'associer chaque nœud de données à un nœud de région. Afin de prouver formellement que cette application m est une solution correcte, on a besoin de définir une sémantique qui reflète l'appartenance à un bloc grâce à m, et ensuite de prouver l'équivalence du programme pour les deux sémantiques étant donnée une application m satisfaisant les propriétés ci-avant.

Dans ce manuscrit, on ne formalise pas la construction de m en soi, on prouve seulement que ces propriétés sont suffisantes pour une équivalence sémantique. Notons, cependant, qu'il est aisé de construire un validateur formellement vérifié pour les propriétés ci-avant de m, et qu'une telle approche aurait l'avantage d'être indépendante de n'importe quelle heuristique dans le choix d'affectation à un nœud de région, comme par exemple sortir certains calculs en dehors d'une boucle, ou bien placer des calculs uniquement à l'intérieur de la branche où le résultat va être utilisé.

Exemple. Prenons l'exemple de la figure 7.1, qui est essentiellement une reprise de la figure 4.3. Dans ce graphe, certains nœuds de données sont plus ou moins libres. Par exemple, le nœud x flotte : la seule restriction, c'est que $m(x)$ doit dominer $m(z)$, puisque z dépend de x. Le raisonnement est similaire pour y. D'autre part, on a $m(c) = r'$, par définition de m sur les nœuds de contrôle. Il en découle que $m(w)$ doit dominer r', puisque c dépend de w. En propageant le raisonnement sur les dépendances de données et par transitivité de la relation de dominance, on a que $m(x), m(z), m(y), m(\phi)$ dominent r'. D'autre part, comme z dépend de ϕ, on a $m(\phi)$ qui domine $m(z)$. Or, $m(\phi) = r'$ par définition de m sur les nœuds Phi. En propageant le raisonnement, r' domine $m(z)$ et $m(w)$, donc finalement $m(z) = m(w) = r'$ est la seule valeur possible. Par contre, on a deux valeurs possibles pour $m(x)$ et $m(y)$: r et r'. En effet, la seule contrainte sur $m(x)$ et $m(y)$ est

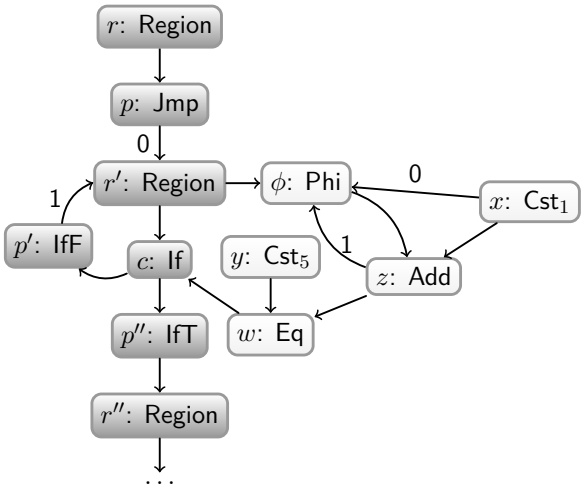


FIG. 7.1 : Exemple de programme de compteur sous forme *Sea of Nodes*

qu'ils doivent dominer $m(w)$ et $m(z)$, c'est-à-dire r'. Par contre, $m(x) \neq r''$, puisque r'' ne domine pas r', et de même pour $m(y)$.

7.1.1 Sémantique par blocs

Nous allons définir une nouvelle sémantique $\rightarrow_{\text{STEP}'}$ pour l'exécution d'une fonction f *Sea of Nodes* au graphe g augmentée de cette application m. Cette sémantique va ressembler essentiellement à notre sémantique pour *Sea of Nodes*, mais avec un traitement différent des dépendances de données. Lors d'un pas $(r, \rho) \rightarrow_{\text{STEP}'} (r', \rho')$, on actualise la valeur de l'environnement non pas uniquement pour les nœuds Phi du bloc r', mais également pour les autres nœuds de données x tels que

$$m(x) = r.$$

On sauvegarde donc une valeur dans l'environnement pour tous les nœuds de données. L'évaluation ultérieure de x utilise une sémantique d'évaluation de nœud de données similaire à précédemment, sauf qu'on utilise l'environnement non pas uniquement pour les nœuds Phi, mais aussi pour tous les autres nœuds tels que $m(x) \neq r''$, où r'' est le nœud de région où l'on évalue : la sémantique d'évaluation prend en effet maintenant du sens uniquement dans le contexte d'un nœud de région particulier.

Évaluation générale des nœuds de données. La nouvelle formule d'évaluation d'un nœud x dans un environnement ρ au nœud de région r est la suivante :

$$[\![x]\!]_{\rho,r} = \begin{cases} \rho(x) & \text{si } g(x) = \text{Phi}(\ldots) \text{ ou } m(x) \neq r \\ N & \text{si } g(x) = \text{Cst}_N \\ \text{op}_{binop}([\![x_1]\!]_{\rho,r}, [\![x_2]\!]_{\rho,r}) & \text{si } g(x) = binop(x_1, x_2) \end{cases}$$

Actualisation de l'environnement pour les nœuds Phi. La règle pour l'actualisation de l'environnement pour les Phi est presque identique, la seule différence est la précision du bloc à partir duquel on considère l'évaluation des dépendances de données pour les entrées des nœuds Phi.

$$\text{PHISGCM}\ \frac{\begin{array}{c}\text{philist}(r) = [\phi_1, \ldots, \phi_m] \qquad \text{index}(p, r) = k \\ \forall i = 1, \ldots, m,\ \text{ntharg}(\phi_i, k) = x_i \qquad [\![x_i]\!]_{\rho,r} = v_i\end{array}}{p, r, \rho \rightarrow_\phi \rho[\phi_i \mapsto v_i \text{ for all } i]}$$

Actualisation de l'environnement pour les autres nœuds de données. En plus de la règle pour les nœuds Phi, qui reste presque la même, il y a maintenant une règle DATAS pour l'évaluation des nœuds de données d'un bloc. Cette règle est similaire à la règle PHIS et permet d'actualiser, de façon parallèle, la valeur des nœuds de données qui ne sont pas des nœuds Phi. On note datanodelist(r) la liste non ordonnée des nœuds de données x (non Phi) tels que $m(x) = r$.

$$\text{DATAS}\ \frac{\begin{array}{c}\text{datanodelist}(r) = [x_1, \ldots, x_n] \\ \forall i, [\![x_i]\!]_{\rho,r} = v_i\end{array}}{r, \rho \rightarrow_{DATA} \rho[x_i \mapsto v_i \text{ for all } i]}$$

Les autres règles de la sémantique. La règle START ne change pas. Pour RET, il s'agit uniquement d'évaluer les dépendances de données en tenant compte du bloc courant.

La règle JMP doit être adaptée pour conserver la nouvelle valeur des instructions du bloc :

$$\text{JMPGCM}\ \frac{\begin{array}{c}r, \rho \rightarrow_{DATA} \rho' \qquad g(c) = \text{Jmp}(r) \qquad g(r') = \text{Region}(\ldots, c, \ldots) \\ c, \rho', r' \rightarrow_\phi \rho''\end{array}}{r, \rho \rightarrow_{\text{STEP}'} r', \rho''}$$

On adapte de façon similaire la règle IFF :

$$\text{IFF}\ \frac{\begin{array}{c}r, \rho \rightarrow_{DATA} \rho' \qquad g(if) = \text{If}(r, x) \qquad \rho'(x) = \texttt{false} \\ g(p) = \text{IfF}(if) \qquad g(r') = \text{Region}(\ldots, p, \ldots) \\ p, r', \rho' \rightarrow_\phi \rho''\end{array}}{r, \rho \rightarrow_{\text{STEP}'} r', \rho''}$$

De façon analogue, on a la nouvelle règle IFT.

7.1.2 Équivalence sémantique

On utilise une relation auxiliaire $\sim$ entre états d'exécution pour la sémantique *Sea of Nodes* et cette nouvelle sémantique ainsi :

- On a Start $\sim$ Start
- On a $\text{Ret}(v) \sim \text{Ret}(v)$ pour tout v.
- On a $\text{Exec}(r, \rho_1) \sim \text{Exec}(r, \rho_2)$ si les conditions suivantes sont vérifiées :

  GCMMATCH1 : Les environnements ρ_1 et ρ_2 coïncident sur la restriction aux nœuds Phi.

  GCMMATCH2 : Pour tout x tel que $m(x)$ domine strictement r, on a $[\![x]\!]_{\rho_1} = \rho_2(x)$.

  Notons que, dans GCMMATCH2, $[\![x]\!]_{\rho_1}$ ne dépend pas de r, puisqu'il s'agit de la sémantique d'évaluation du chapitre 4 et non de celle introduite dans la section précédente.

Simulation. On note σ_1 un état d'exécution pour la sémantique *Sea of Nodes* et σ_2 un état d'exécution pour la sémantique par blocs. Le théorème de simulation à démontrer est le suivant :

Théorème 7.1.2. *Si $\sigma_1 \sim \sigma_2$, que $\sigma_1 \rightarrow_{STEP} \sigma_1'$, alors il existe σ_2' tel que $\sigma_2 \rightarrow_{STEP'} \sigma_2'$ et $\sigma_1' \sim \sigma_2'$.*

Démonstration. Étant donné que les fonctions sont les mêmes et seule la sémantique change, il s'agit essentiellement de prouver qu'on préserve les invariants.

Le cas où σ_1 est un Start (et donc σ_2 aussi par hypothèse) est facile. GCMMATCH1 est immédiat puisqu'aucun Phi n'a encore de valeur. GCMMATCH2 est immédiat puisqu'aucun nœud ne domine strictement le point d'entrée.

Le cas où σ_1 est un $\text{Ret}(v)$ ne peut pas arriver (plus de transition possible).

Il reste donc à traiter le cas où $\sigma_1 = \text{Exec}(r, \rho_1)$. L'hypothèse permet de déduire que dans ce cas $\sigma_2 = \text{Exec}(r, \rho_2)$ avec ρ_2 qui coïncide sur les Phi avec ρ_1 et tel que pour tout x tel que $m(x)$ domine strictement r, on a $[\![x]\!]_{\rho_1} = \rho_2(x)$.

La formule d'évaluation $[\![x_i]\!]_{\rho_2,r}$ d'un x_i intervenant dans la règle DATAS fait intervenir la valeur de l'environnement uniquement pour les nœuds Phi ou les nœuds de données x avec $m(x) \neq r$. Cette dernière inégalité combinée à la propriété 7.1.1 de *strictness* sur m assure, pour un tel x, que $m(x)$ domine strictement r. Ceci permet de lui appliquer l'invariant GCMMATCH2, c'est-à-dire $[\![x]\!]_{\rho_1} = \rho_2(x)$. Du coup, par une induction sur la définition de

l'évaluation des nœuds de données, on déduit que $[\![x_i]\!]_{\rho_1} = [\![x_i]\!]_{\rho_2,r}$. Il en résulte que, si ρ'_2 est l'environnement après application de la règle DATAS, on a $[\![x]\!]_{\rho_1} = \rho'_2(x)$ pour tout x dominant r, donc pour tout x dominant strictement r'. Ceci correspond à la préservation de GCMMATCH2 en r' avec les environnements ρ_1 et ρ'_2.

Cette dernière remarque permet ensuite de prouver la préservation de GCMMATCH1 (et donc aussi de GCMMATCH2 pour les nœuds Phi). En effet, lors de l'application de la règle PHIS pour r', chaque entrée de nœud Phi x_i considérée domine r, donc d'après la formule du paragraphe précédent, $[\![x_i]\!]_{\rho_1} = \rho'_2(x_i)$. Si ρ'_1 et ρ''_2 sont les environnements obtenus après PHIS, on a donc $\rho'_1(\phi) = \rho''_2(\phi)$ pour tout ϕ. De plus, l'application de la règle PHIS n'a pas modifié la valeur d'évaluation des nœuds de données dominant strictement r, donc GCMMATCH2 reste vrai en r' pour les nouveaux environnements ρ'_1 et ρ''_2. □

7.2 Séquentialisation des instructions d'un bloc

Le retour au bloc de base décrit dans la section précédente permet d'affecter un nœud de région (bloc) à chaque nœud de données. Cependant, dans chaque bloc, l'ordre d'exécution des instructions reste flexible en fonction des dépendances de données. Afin de générer du code ultérieurement, il faut séquentialiser les instructions. L'algorithme de séquentialisation peut être un simple tri topologique sur le graphe acyclique des dépendances de données des instructions du bloc. Des heuristiques additionnelles, pour optimiser par exemple le *pipeline*, peuvent être utilisées. Ce genre d'algorithme a fait l'objet d'études formelles par validation a posteriori [68].

Considérons, de nouveau, l'exemple de la figure 7.1. Supposons, par exemple, que parmi les solutions valides pour m, l'algorithme en a donné une qui vérifie les équations suivantes :

$$m(y) = m(x) = m(z) = m(w) = r'.$$

Dans ce cas, il faut donc ordonner ces quatre nœuds, au sein d'un même bloc, dans un ordre qui respecte les dépendances. En particulier, x, y et z doivent apparaître avant w dans la séquentialisation. Par contre, on a plusieurs choix possibles pour ordonner x, y et z entre eux : $[x, z, y]$ ou $[x, y, z]$ ou $[y, x, z]$.

La difficulté est d'ordre essentiellement sémantique : il s'agit de prouver que la sémantique par blocs sur un programme est équivalente à une sémantique séquentielle sur une séquentialisation du bloc respectant les contraintes imposées par les dépendances. Ce problème n'est pas traité dans ce manuscrit. Une piste à regarder est le travail effectué par Blech et al. [8], qui étudient l'utilisation de graphes de termes, ainsi que d'ensembles relationnels. Les premiers leur servent à capturer syntaxiquement les dé-

pendances de données à l'intérieur d'expressions. Les deuxièmes leur permettent de définir une stratégie d'évaluation pour des blocs de base SSA à partir d'ordres partiels. Cette seconde méthode leur permet d'intégrer dans leur sémantique la notion d'ordonnancement des instructions via l'utilisation de dépendances additionnelles, permettant un passage entre une sémantique SSA considérant flexiblement les dépendances de données dans chaque bloc et une sémantique séquentielle. On pense que leur travail pourrait être adapté pour prouver une séquentialisation des nœuds à partir de notre sémantique par blocs.

7.3 Conclusion

Dans ce chapitre, nous avons présenté une sémantique, que l'on a appelé sémantique par blocs, afin de pouvoir donner un sens sémantique précis au retour à une représentation utilisant des blocs de base via *Global Code Motion* dans le contexte de *Sea of Nodes*. Nous avons ensuite cerné des propriétés, concernant l'affectation de chaque nœud de données à un nœud de région, qui permettent d'établir l'équivalence entre la nouvelle sémantique et la sémantique du chapitre 4. Enfin, nous avons discuté de possibles approches sémantiques du problème de séquentialisation des instructions à l'intérieur d'un bloc.

Chapitre 8

Destruction SSA : élimination des Phi

Dans ce chapitre, nous décrivons une destruction réaliste de la forme SSA, prouvée formellement et intégrée dans le compilateur vérifié CompCertSSA. Cette phase de destruction s'inscrit juste après la phase de retour au bloc de base étudiée au chapitre précédent.

8.1 Introduction

8.1.1 Remarques préliminaires

Le travail présenté dans ce chapitre a fait l'œuvre d'un article publié à CC'06 [26]. Comme ce travail a été guidé essentiellement par l'intégration dans CompCertSSA, la forme SSA présentée diffère syntaxiquement des autres formes SSA telles que présentées dans le reste du manuscrit et est plus proche du langage RTL de CompCert (graphe de flot de contrôle d'instructions avec une infinité de registres). En résumé, au lieu d'avoir un graphe de nœuds (ou points de programme) représentant des instructions dans lequel les entrées représentent d'autres nœuds, on a un graphe d'instructions où chaque instruction met en jeu des *variables* (ou registres), qui vont représenter aussi bien les entrées d'une instruction que la destination : par exemple $z := x + y$ est une instruction d'addition en un certain nœud du programme mettant en jeu trois variables x, y et z. Entre autres, la propriété de définition unique n'est pas intégrée dans la syntaxe mais est représentée par un invariant supplémentaire du langage.

Enfin, il est particulièrement important de noter que l'on n'utilise *pas* de blocs de base, mais simplement un CFG d'instructions représenté par une structure d'arbre purement fonctionnelle.

8.1.2 Destruction SSA

La destruction de la forme SSA (c'est-à-dire l'élimination des Phi) peut être vue comme une façon de compiler les ϕ-instructions, ou de les implémenter avec des instructions disponibles du langage, en l'occurrence des copies. Éliminer les Phi de façon non naïve (sans introduire trop de copies) est notoirement un problème délicat. En effet, la transformation pour sortir de SSA a été révisée à multiples reprises, pour des questions de correction et de performances, depuis son introduction au début des années 90 jusqu'à récemment. Nous donnons un bref résumé de ces révisions, dans l'ordre chronologique. On peut trouver un résumé plus complet dans [30].

La destruction telle que proposée initialement par Cytron et al. [25] consiste essentiellement dans l'introduction d'une copie $x \leftarrow x_i$ a chaque i-ième prédécesseur d'un point de jonction contenant un nœud $\text{Phi}(x_1, \ldots, x_m)$. Les copies sont ensuite fusionnées en utilisant du *coalescing* à la Chaitin [18]. Briggs et al. [14] remarquent par la suite que cette destruction est incorrecte en présence d'arcs critiques dans le graphe de flot de contrôle après un analyse de *Global Value Numbering* ou une propagation de copies agressive (le dit problème du *lost-copy*). On parle d'arc critique entre un nœud parent et un fils successeur lorsque le parent a plus d'un successeur et le fils a lui-même plusieurs prédécesseurs. D'autres cas donnant lieu à des bugs peuvent survenir après propagation de copies si la sémantique parallèle des Phi d'un bloc n'est pas considérée avec attention (le problème du *swap*). Ils proposent deux nouveaux algorithmes pour corriger ces problèmes.

Les travaux de Sreedhar et al. [62] constituent une réelle avancée dans la compréhension du problème de la destruction de SSA. Ils identifient une sous-classe de SSA, le *Conventional SSA* (CSSA), dans laquelle l'élimination des Phi est, pour ainsi dire, clairement correcte. En fait, CSSA est *défini* dans [62] comme une forme SSA dans laquelle toutes les variables d'un même Phi peuvent être fusionnées ensemble sans changer la sémantique du programme. Ils observent que, même si la génération de SSA en soi assure la propriété CSSA, beaucoup d'optimisations cassent cette propriété (en particulier la propagation de copies et l'élimination de sous-expressions communes basée sur le *Global Value Numbering* [15]). Du coup, ils proposent une destruction de SSA via une transformation préliminaire vers CSSA, en insérant des copies fraîches pour assurer une propriété de ϕ-congruence : toutes les variables liées (transitivement) par des Phi doivent avoir des *live-ranges* disjoints. Ils raffinent progressivement leur stratégie d'insertion de copie pour minimiser le nombre de copies insérées.

Plus récemment, Boissinot et al. [9] reconsidèrent le problème, mettant l'accent sur la nécessité d'une solution conceptuellement simple pour la destruction de SSA, afin d'assurer à la fois la correction et la performance de la destruction. Afin d'assurer la correction de la destruction, ils proposent d'abord de générer du CSSA à l'aide de la version naïve de Sreedhar et al. d'insertion de copies (nous revenons sur cette transformation plus loin

en détail). La phase de destruction est alors abordée comme un problème générique de *coalescing*, dans lequel la notion de non-interférence est raffinée pour inclure, en plus de l'information de *liveness*, une information supplémentaire sur les valeurs qui peut être facilement calculée dans SSA. Notons que les copies introduites sont des *copies parallèles* dans l'esprit du modèle parallèle d'exécution des Phi d'un même bloc, ce qui offre plus d'opportunités pour éliminer des copies. Les copies parallèles restantes après *coalescing* sont séquentialisées dans une dernière étape. Boissinot et al. [9] montrent un clair intérêt pour l'aspect correction de la transformation, signalant certaines imprécisions dans les travaux de Sreedhar et al. [62] relatives à l'information de *liveness* utilisée pour tester l'interférence entre variables. Ils donnent également quelques preuves haut-niveau à propos de la correction de la génération de CSSA et de leur critère de non-interférence.

8.1.3 Destruction SSA et Vérification

Même si Sreedhar et al. [62] et Boissinot et al. [9] donnent des arguments haut-niveau pour justifier la correction de leur stratégie de destruction SSA, il n'en ressort pas clairement que la non-interférence des variables des Phi soit une condition suffisante pour prouver la correction de l'élimination des Phi : la transformation pourrait nécessiter des invariants plus forts pour des cas limites non identifiés.

De fait, jusqu'à présent, l'état de l'art en compilation vérifiée soit ignorait, soit traitait partiellement ou naïvement la destruction SSA. En particulier, la destruction telle que dans CompCertSSA [4, 28] était partielle, nécessitant que les Phi d'un bloc puissent être exécutés indifféremment en parallèle ou séquentiellement. Sur les programmes qui ne vérifient pas ce critère, le compilateur échoue. Cette restriction empêche des optimisations. De plus, sur les programmes qui vérifient le critère, la destruction se fait via une introduction massive de copies impactant l'efficacité de la phase ultérieure d'allocation de registres ce qui, en pratique, donne lieu à une quantité substantielle d'instructions de *spill/reload*. Du coup, la plupart du gain que l'on peut espérer des optimisations SSA est perdu à ce niveau. Le projet Vellvm [75, 74], pour sa part, ne considère simplement pas le problème, traitant uniquement de la construction SSA et d'optimisations. La destruction n'appartient pas à la partie vérifiée formellement du projet.

8.1.4 Contributions

L'intégration complète (sans échec du compilateur) et non-naïve de la destruction SSA dans un compilateur vérifié soulève des questions intéressantes. En effet, prouver la correction de l'élimination des Phi demande d'établir un résultat de *précision* de l'analyse de *liveness* sous-jacente. Ceci

est à contraster avec la plupart des analyses statiques dans les compilateurs vérifiés, où l'on se contente de prouver la correction, puisque un manque de précision conduit seulement à du code moins optimisé. Dans le cas de la destruction SSA, la situation est différente : si les variables d'un Phi ont des *live-ranges* qui s'entrelacent, ou si l'analyse est trop grossière pour détecter qu'ils sont disjoints, il n'est pas possible de prouver qu'éliminer les Phi préserve la sémantique du programme. Ici, on observe que la génération de CSSA (méthode I décrite par Sreedhar et al., et Boissinot et al.) assure que l'on peut obtenir la précision nécessaire, grâce à l'insertion de copies fraîches pour toutes les variables en jeu dans un Phi (entrées et destination), que l'on nommera ϕ-ressources dans le reste du chapitre.

Nous expliquons dans la suite l'implémentation d'une telle destruction dans le compilateur vérifié CompCertSSA, dont la preuve de correction est réalisée à l'aide de l'assistant de preuve Coq. On implémente la destruction à la Boissinot et al. [9]. Ceci comprend la génération de CSSA en utilisant des copies parallèles, l'implémentation d'un algorithme de *coalescing* de variables dans CSSA en utilisant une notion affinée d'interférence et la séquentialisation des copies parallèles restantes. D'un point de vue technique, les contributions sont les suivantes :

- Nous implémentons la génération de CSSA et prouvons qu'elle préserve la sémantique. Malgré sa simplicité conceptuelle (introduire des variables fraîches et des copies parallèles, puis renommer les instructions Phi en accord), la preuve est assez technique.

- Nous prouvons que, dans le programme CSSA généré, les ϕ-ressources d'une même ϕ-instruction ont des *live-ranges* disjoints. Ce résultat de précision, combiné à la propriété de fraîcheur des copies insérées, permet de prouver, indépendamment de l'algorithme de *coalescing* utilisé, qu'éliminer les ϕ-instructions du CSSA généré préserve la sémantique.

- Nous implémentons un algorithme de *coalescing* et prouvons qu'il est correct. À cette fin, nous formalisons la notion affinée de non-interférence de Boissinot et al. [9], qui prend en compte la *SSA-valeur*, une approximation symbolique de la valeur à l'exécution d'une variable SSA. Afin de ne pas dépendre d'une heuristique de *coalescing* particulière, nous utilisons la validation *a posteriori* (voir la section 3.3.1).

- D'un point de vue validation expérimentale, notre algorithme élimine, en moyenne, plus de 99% des copies introduites, ce qui est le résultat attendu étant donné les optimisations actuelles de CompCertSSA qui n'introduisent que très peu d'interférences au niveau des Phi. Ceci conduit à des résultats encourageants en ce qui concerne le *spilling* et le *reloading* pendant la phase post-SSA d'allocation de registres.

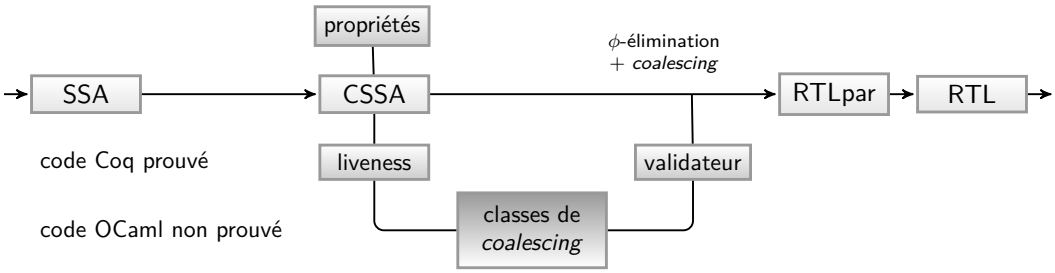


FIG. 8.1 : Aperçu de la destruction basée sur CSSA

Le *middle-end* de CompCertSSA, étendu avec la nouvelle destruction SSA est illustré dans la figure 8.1. Il est inséré dans CompCert au niveau du langage intermédiaire RTL, c'est-à-dire un langage 3-adresses, non structuré, représenté à l'aide d'un graphe de flot de contrôle d'instructions utilisant des registres virtuels. Après une phase de normalisation, le code SSA est produit à partir du code RTL, et des optimisations SSA sont réalisées. La génération et les passes d'optimisation sont décrites dans [4, 28, 6]. Le travail présenté dans ce chapitre est représenté dans la figure. Il comprend la génération du *Conventional SSA* (CSSA), et l'algorithme de *coalescing* pour revenir de SSA vers RTL. Dans un premier temps, on élimine les ϕ-instructions et fusionne des variables. Dans un deuxième temps, on séquentialise les copies parallèles restantes (s'il y en a) en utilisant la formalisation de Rideau et al. [55]. Nous introduisons entre ces deux phases un langage intermédiaire RTLpar, une variante de RTL étendue avec des blocs de copies parallèles.

Structure du chapitre. Dans la section 8.2, on présente une formalisation de la forme CSSA (syntaxe et sémantique), ainsi que sa propriété importante des *live-ranges* disjoints pour les ϕ-ressources. Dans la section 8.3, on présente une destruction basique de CSSA qui ne fait qu'éliminer les ϕ-instructions, sans *coalescing* additionnel. Nous donnons les grandes lignes de sa preuve de correction sémantique. La destruction de CSSA présentée dans la section 8.4 est une amélioration qui intègre une notion plus affinée de non-interférence qui tient compte de la SSA-valeur des variables. Dans la section 8.5, on présente quelques résultats expérimentaux mesurant l'efficacité de ce *coalescing* vérifié sur des programmes de test. Finalement, la section 8.6 conclut et discute de possibles directions pour des développements futurs en lien avec le sujet de ce chapitre.

Dans ce chapitre, on choisit de ne pas présenter les résultats sous leur forme Coq, pour plus de clarté. Le développement formel est disponible en ligne à l'adresse `http://compcertssa.gforge.inria.fr/`.

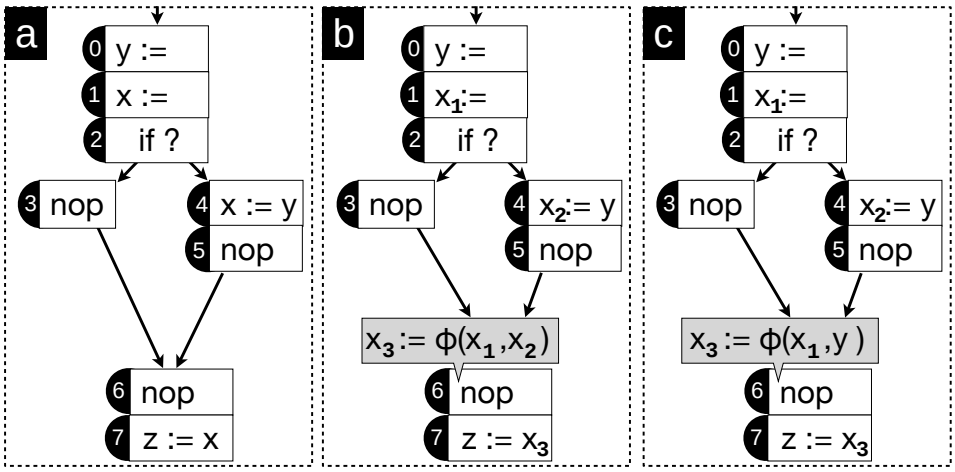


FIG. 8.2 : Exemples de programmes. Le programme a) est le programme RTL initial normalisé, le programme b) est la version SSA et le programme c) est obtenu en propageant la copie du nœud 4 au nœud 6.

8.2 *Conventional SSA*

Avant de donner une définition formelle de CSSA, commençons par un exemple. Le programme b) de la figure 8.2 donne un exemple de fonction SSA construite à partir de la fonction RTL normalisée du programme a) (nous expliquons la normalisation plus loin dans le chapitre). Certaines instructions ont été collées, pour plus de clarté, mais nous rappelons que CompCertSSA n'utilise pas de blocs de base, mais un simple CFG d'instructions. Les variables sont définies une fois seulement, et la ϕ-instruction au point de jonction sélectionne, à l'exécution, parmi x_1 et x_2, celle qui est affectée à x_3 en fonction du chemin d'exécution. Dans l'exemple, si la branche gauche de la condition est exécutée, x_3 va recevoir la valeur de x_1, et, si la branche de droite est prise, il va recevoir la valeur de x_2. Observons, dans le programme b), comment après la transformation vers SSA, les variables x_1, x_2 et x_3 pourraient être fusionnées en une seule variable et, donc, la ϕ-instruction pourrait être enlevée, sans affecter la sémantique de la fonction, du fait que les *live-ranges* sont disjoints.

Considérons maintenant le programme c) de la figure 8.2. Il est obtenu après simple propagation de copie dans le programme b). Dans ce programme, enlever la ϕ-instruction, et fusionner les variables x_3, x_1 et y serait incorrect. En effet, le programme c) est un programme SSA qui n'est plus *conventional* : les variables x_1 et y (arguments de la ϕ-instruction au point 6) sont toutes les deux vivantes au point 2, donc leurs *live-ranges* s'intersectent et il serait incorrect de fusionner les variables y et x_1. En effet, l'affectation de x_1 écraserait celle de y.

Le but de la transformation vers CSSA est de réétablir la propriété

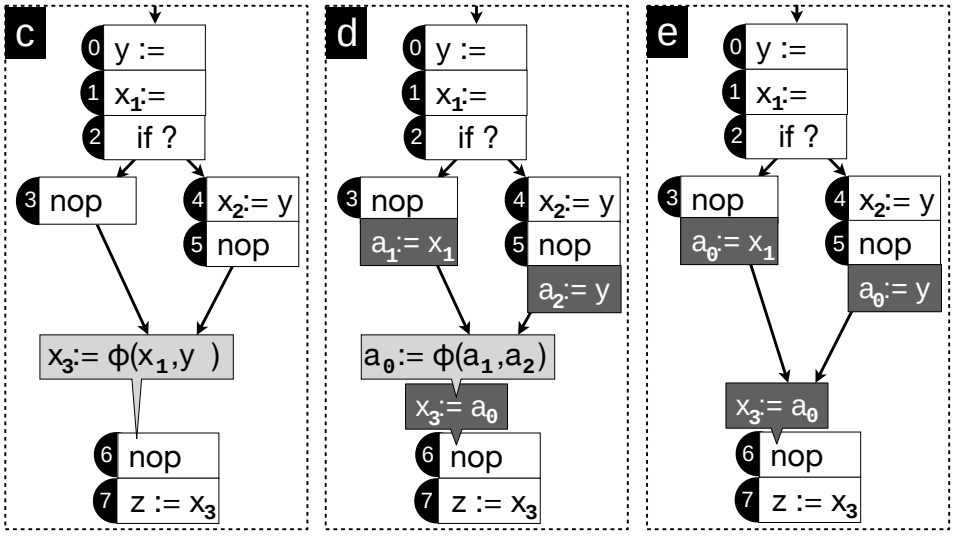


FIG. 8.3 : Exemples de programmes. Le programme c) est un programme SSA optimisé, le programme d) s'obtient en transformant vers CSSA et le programme e) s'obtient en éliminant les ϕ-instructions.

de *live-ranges* disjoints pour les ϕ-ressources. L'idée de l'algorithme proposé par Boissinot et al. [9] est d'introduire des variables fraîches et des copies parallèles des ϕ-ressources aux points de jonction et leurs prédécesseurs, de sorte que les ϕ-instructions deviennent isolées. C'est-à-dire que, pour chaque ϕ-instruction $x_0 := \phi(x_1, \ldots, x_n)$ au point de jonction pc, on introduit des variables fraîches $a_0, \ldots, a_n$, puis on procède comme il suit :

- La ϕ-instruction est remplacée par $a_0 := \phi(a_1, \ldots, a_n)$.
- Une copie $x_0 := a_0$ est ajoutée au bloc de copies parallèles situé en pc.
- Une copie $a_i := x_i$ est ajoutée au bloc de copies parallèles situé au i-ième prédécesseur de pc, pour tout $i \in \{1, \ldots, n\}$.

Dans le programme d) de la figure 8.3, des variables fraîches a_0, a_1, a_2 sont introduites pour remplacer la ϕ-instruction $x_3 := \phi(x_1, y)$, et des copies parallèles sont insérées aux nœuds 3, 5 et 6 (en gris foncé). Comme attendu, les variables a_i ont toutes des *live-ranges* disjoints.

Dans le reste de cette section, nous présentons une modélisation de la syntaxe, de la sémantique et des propriétés de CSSA. La plupart des propriétés sont héritées de SSA et préservées par la génération de CSSA. La propriété distinctive de CSSA, les *live-ranges* disjoints, est présentée dans la section 8.2.4.

pc, pc', s	$\ni$	node	
r, x, y	$\ni$	reg	
ι	::=	$pc \mapsto$ instr	instructions
φ	::=	$pc \mapsto \overrightarrow{\text{phi} - \text{instr}}$	ϕ-blocs
μ	::=	$pc \mapsto \overrightarrow{\text{pcopy}}$	copies parallèles

instr	::=	nop(s)	non-opération
		| iop$(op, x_d, \vec{x}, s)$	opération arithmétique
		| copy(x_d, x, s)	copie $(x_d := x)$
		| if$(c, \vec{x}, s_{true}, s_{false})$	conditionnelle
		| ...	
phi − instr	::=	phi$(x_d, \vec{x})$	phi-instr. $(x_d := \phi(\vec{x}))$
pcopy	::=	mv(x_d, x)	copie $(x_d := x)$

$$f \ni \text{function}_{\text{CSSA}}$$

$$f ::= \left\{ \begin{array}{ll} \text{name} = id; & \text{params} = \vec{x}; \\ \text{entry} = pc; & \text{code} = \iota; \\ \text{phicode} = \varphi; & \text{parcode} = \mu \end{array} \right\}$$

FIG. 8.4 : Syntaxe de CSSA (extrait)

8.2.1 Syntaxe abstraite

La syntaxe d'une fonction CSSA est donnée dans la figure 8.4. Une fonction CSSA a un nom, une liste de paramètres, un point d'entrée et une fonction code d'un ensemble de nœuds (points de programme) vers des instructions (nop, opérations arithmétiques, copies, conditionnelles ... [1]). Chaque instruction sait quels sont ses nœuds successeurs. Ainsi, le CFG de la fonction est complètement déterminé par le graphe code. La fonction CSSA a aussi une fonction phicode des nœuds vers des ϕ-blocs (des listes de ϕ-instructions de la forme $x_d := \phi(\vec{x})$), et une fonction parcode des nœuds vers des blocs de copies parallèles (des listes de pcopy, voir la figure 8.4).

Pour simplifier le raisonnement dans nos preuves, on utilise certaines contraintes structurelles sur $\texttt{function}_{\texttt{CSSA}}$. Dans notre développement, ces contraintes sont soit prouvées directement, soit validées a posteriori.

Propriétés basiques de bonne formation structurelle. La fonction phicode est uniquement définie aux points de jonction. Similairement, la fonction parcode est définie aux points de jonction et leurs prédécesseurs uniquement. Chaque ϕ-instruction a exactement le bon nombre d'arguments. On suppose que chaque point du CFG est atteignable depuis le point d'entrée, et qu'il n'y a pas d'instruction qui pointe vers un nœud qui n'aurait pas d'instruction. Finalement, le point d'entrée n'a pas de prédécesseur et n'est pas le prédécesseur d'un point de jonction.

Normalisation de code. Premièrement, la normalisation de RTL assure que seulement un nop peut conduire à un point de jonction, et que le point d'entrée est également un nop. Ceci est hérité de CompCertSSA et permet de simplifier la définition de la sémantique [4]. Cette normalisation évite aussi, un peu accidentellement, des cas pathologiques comme les arcs critiques, même si la construction de CSSA utilisée, celle de Boissinot et al. [9], est en fait conçue pour tenir compte de ces cas-là ; il s'agit en fait de la raison pour laquelle les copies parallèles au niveau du point de jonction ont été introduites à l'origine.

Deuxièmement, nous supposons qu'il y a une instruction $\texttt{nop}(s)$ à tous les points de jonction. Cet invariant est introduit pour simplifier les raisonnements de *liveness* aux points de jonction.

Enfin, il ne peut pas y avoir deux points de jonction à la suite dans le graphe de flot de contrôle, ce qui permet d'utiliser un seul graphe parcode pour les blocs de copies parallèles.

1. Nous présentons ici uniquement les instructions essentielles, sans donner beaucoup de détails techniques du langage dans sa totalité, comme les accès mémoire et les appels de fonction. Notre formalisation gère l'ensemble des instructions, permettant la compilation de programmes C respectant le standard ISO C90 / ANSI-C.

$$\begin{array}{llll} \sigma & \ni & \texttt{state} & v \ni \texttt{value} \qquad \rho \ni \texttt{regset} \\ \sigma & ::= & (f, pc, \rho) & \rho ::= x \mapsto v \quad \text{env. local} \end{array}$$

$$\text{NopNJP}\ \frac{f.\texttt{code}(pc) = \texttt{nop}(pc') \quad \neg(\texttt{joint\_point}\ f\ pc')}{(f, pc, \rho) \xrightarrow{\epsilon}_{\texttt{CSSA}} (f, pc', \rho)}$$

$$\text{NopJP}\ \frac{f.\texttt{code}(pc) = \texttt{nop}(pc') \quad (\texttt{joint\_point}\ f\ pc') \quad \texttt{index\_pred}(f, pc, pc') = k \quad f.\texttt{parcode}(pc) = \texttt{parcb} \quad f.\texttt{phicode}(pc') = \texttt{phib} \quad f.\texttt{parcode}(pc') = \texttt{parcb}'}{(f, pc, \rho) \xrightarrow{\epsilon}_{\texttt{CSSA}} (f, pc', [\![\texttt{parcb}']\!] \circ [\![\texttt{phib}]\!]_k \circ [\![\texttt{parcb}]\!]\rho)}$$

Parallel-copy semantics

$$\begin{array}{lcl} [\![\texttt{nil}]\!]\rho & = & \rho \\ [\![\texttt{mv}(x_d, x) :: \texttt{parcb}]\!]\rho & = & ([\![\texttt{parcb}]\!]\rho)[x_d \leftarrow \rho(x)] \end{array}$$

Phi-blocks semantics

$$\begin{array}{lcl} [\![\texttt{nil}]\!]_k\rho & = & \rho \\ [\![\texttt{phi}(x_d, \vec{x}) :: \texttt{phib}]\!]_k\rho & = & ([\![\texttt{phib}]\!]_k\rho)[x_d \leftarrow \rho(\vec{x}.k)] \end{array}$$

FIG. 8.5 : sémantique CSSA (extrait)

8.2.2 Sémantique

On utilise les notations suivantes. On écrit $(f \circ g)(x) = f(g(x))$ pour la composition de fonctions. Si $\vec{x}$ est un vecteur de variables $(x_1, \ldots, x_n)$, alors pour $k \in \{1, \ldots, n\}$, on écrit $\vec{x}.k$ pour x_k.

La figure 8.5 présente les règles définissant la sémantique opérationnelle petit-pas de CSSA, exprimée à l'aide d'un système de transitions étiquetées entre états d'exécution, dont les transitions sont de la forme $\sigma \xrightarrow{e}_{\texttt{CSSA}} \sigma'$. Nous donnons uniquement les transitions pour les instructions qui diffèrent de celles de CompCertSSA et qui jouent un rôle significatif dans la suite. Pour celles-ci, l'évènement observable e en étiquette de la transition sera l'évènement silencieux ϵ[2], et les états d'exécution σ (définis dans la figure 8.5) sont de simples triplets (f, pc, ρ) avec la fonction f exécutée, le point courant du programme pc (un nœud dans le code donnant l'instruction suivante à exécuter), et l'environnement local courant ρ, associant les variables à des valeurs. On écrit $\rho[x \leftarrow y]$ pour l'actualisation, dans ρ, de la valeur de x vers celle de y.

Dans la figure 8.5, la règle NopNJP est la plus simple. Elle exprime l'exécution d'une instruction $\texttt{nop}(pc')$ lorsque pc' n'est pas un point de jonction (comme l'indiquent les prémisses). Un tel pas change uniquement le point courant du programme (il se déplace vers pc') et, en particulier, ne

2. La formalisation complète du langage gère les transitions non silencieuses produisant une trace.

modifie pas l'environnement ρ.

Lorsque le successeur pc' de l'instruction $\texttt{nop}(pc')$ est un point de jonction (règle NopJP), tant le bloc de copies parallèles `parcb` au point pc, comme le ϕ-bloc `phib` au point pc', et le bloc de copies parallèles `parcb'` au nœud pc' doivent être exécutés, modifiant l'environnement ρ. Ceci est défini par $[\![\texttt{parcb}']\!] \circ [\![\texttt{phib}]\!]_k \circ [\![\texttt{parcb}]\!]\rho$: les trois blocs sont exécutés en un seul « petit » pas entre le nœud pc et le nœud pc'. Plus précisément, les ϕ-blocs et les blocs de copies parallèles utilisent une sémantique parallèle (voir la figure 8.5). Exécuter un bloc `parcb` dans un environnement ρ, ce que l'on écrit $[\![\texttt{parcb}]\!](\rho)$, modifie l'environnement ρ de sorte que pour chaque copie $x_d := x$ dans le bloc, x_d reçoit la valeur de x dans l'environnement initial ρ. La sémantique d'un ϕ-bloc `phib` dans un environnement ρ, $[\![\texttt{phib}]\!]_k(\rho)$ est définie similairement, donnant à la destination x_d de chaque ϕ-fonction la valeur (dans ρ) du k-ième argument de la ϕ-fonction ($\vec{x}.k$).

Ce « petit » pas sémantique au niveau des prédécesseurs de point de jonction est hérité de CompCertSSA et utile pour la même raison, c'est-à-dire ne pas avoir à instrumenter la sémantique avec l'index du dernier prédécesseur. Cette sémantique est, de plus, proche de celle de RTLpar (à part pour les ϕ), ce qui facilite la preuve de la transformation.

Comme mentionné précédemment, les règles de transition pour les autres instructions sont les mêmes que dans CompCertSSA et n'interviennent pas significativement dans les sujets que nous abordons et nous ne donnons donc pas les détails. En effet, grâce à la normalisation du code, ces instructions ne conduisent pas à un point de jonction et ne demandent donc pas l'exécution d'un ϕ-bloc ou d'un bloc de copies parallèles. Par exemple, la règle pour les opérations arithmétiques est de la forme :

$$\text{Iop}\ \frac{\begin{array}{c} f.\texttt{code}(pc) = \mathit{Some}(\texttt{iop}(op, args, r, pc') \\ \texttt{eval\_op}(op, \rho, args) = \mathit{Some}(v) \end{array}}{(f, pc, \rho) \xrightarrow{\epsilon}_{\texttt{CSSA}} (f, pc', \rho[r \leftarrow v])}$$

où op est une opération (addition par exemple), $args$ les arguments de l'opération, et r la variable recevant le résultat v de l'opération.

8.2.3 Définition unique et *strictness*

Tout comme dans SSA, chaque variable d'une fonction CSSA a un point unique de définition (soit à un point avec une instruction régulière, soit dans une ϕ-instruction, soit comme copie dans un bloc de copies parallèles). Dans notre formalisation, on définit formellement cette notion et on prouve qu'elle est assurée. Ici, pour alléger la présentation, nous omettons la définition formelle et considérons que l'on dispose d'une fonction $\text{def}_f() : \texttt{reg} \to \texttt{node}$ qui calcule l'unique nœud dans une fonction f où une variable est définie.

La notion de *strictness* est plus intéressante : dans une fonction CSSA, chaque usage de variable doit être strictement dominée par son point de définition. Cependant, pour simplifier la sémantique et éviter de modifier le CFG, les blocs de copies parallèles au point de jonction sont, comme nous l'avons vu, attachés au même point du CFG que le bloc de ϕ-instructions. Afin de prendre en compte cette spécificité, nous utilisons la définition suivante.

Définition 8.2.1. Dans la fonction f, une variable x *domine strictement* un nœud pc, ce que l'on note $x \succ pc$, lorsqu'une des conditions suivantes est vérifiée :

- $\text{def}_f(x)$ domine strictement pc dans le CFG de f ;
- x est la destination d'une ϕ-instruction au point pc ;
- x est la destination d'une copie d'un bloc de copies parallèles en pc et pc est un point de jonction.

Cette dernière condition, analogue à la deuxième, est spécifique à CSSA du fait de l'ajout des blocs de copies parallèles aux points de jonction. Du fait de la normalisation qui place un nop à tous les points de jonction, elle n'est pas intrinsèquement nécessaire — ces copies ne peuvent pas être utilisées au même point après leur définition. Cependant, elle simplifie, dans le développement Coq, l'énoncé de certains lemmes intermédiaires utilisant le concept de dominance stricte, car elle permet de considérer séparément les copies parallèles des instructions normales. En effet, la sémantique exécute les copies parallèles lors du pas qui arrive sur leur point de définition, contrairement aux instructions normales du CFG qui sont exécutées lors du pas qui quitte leur point de définition.

Étant donnée cette définition, on peut prouver la propriété suivante :

Lemme 8.2.2. *Dans une fonction CSSA f, si une variable x est utilisée en un nœud pc, alors $x \succ pc$. De plus, aucune variable n'est à la fois définie et utilisée dans un même bloc de copies parallèles.*

Notons que la propriété analogue pour les ϕ-blocs est une conséquence de la *strictness* générale sur le CFG, car les arguments d'une ϕ-instruction au niveau d'un point de jonction sont considérés comme utilisés au prédécesseur, comme il est standard de faire avec SSA [25].

8.2.4 Séparation des *live-ranges*

Nous concentrons notre attention sur les propriétés importantes établies lors de la transformation vers CSSA. Tout d'abord, nous observons que dans [9], les esquisses de preuve justifiant la correction de la destruction reposent informellement sur la façon précise dont les copies fraîches

sont insérées afin d'isoler chaque ϕ-instruction dans un même ϕ-bloc. En particulier, des noms frais sont introduits pour chaque ϕ-instruction. On appelle ceci la propriété des ϕ-ressources disjointes.

Définition 8.2.3. Une fonction CSSA f satisfait la propriété des ϕ-ressources disjointes si toutes les variables apparaissant dans des ϕ-fonctions (n'importe où dans f.phicode) sont deux à deux distinctes.

En particulier, ceci assure qu'aucune variable ne peut apparaître à plus d'une reprise dans un ϕ-bloc. Remarquons maintenant que la définition 8.2.3 capture uniquement le fait que les ϕ-fonctions n'ont pas de conflits en ce qui concerne les noms de variables. Il reste à formaliser que les ϕ-fonctions sont en effet *isolées* du reste du code en termes de *live-ranges*. Dans la littérature, ceci est souvent appelée la séparation des *live-ranges* (en anglais *live-range splitting*) de la transformation vers CSSA. Nous utilisons la définition suivante :

Définition 8.2.4. Dans une fonction f, les variables x et y ont des *live-ranges* disjoints (ce que l'on note $x \perp y$) lorsque $x \notin \mathsf{live}_f^{\mathsf{out}}(\mathsf{def}_f(y))$, $y \notin \mathsf{live}_f^{\mathsf{out}}(\mathsf{def}_f(x))$ et $\mathsf{def}_f(x) \neq \mathsf{def}_f(y)$.

Dans cette définition, nous définissons formellement l'information de *liveness* $\mathsf{live}_f^{\mathsf{out}}(pc)$ comme un prédicat inductif, caractérisant l'ensemble des variables x, dans le CFG de f, tels qu'il existe un chemin de pc vers un point d'utilisation de x, sans redéfinition de x en chemin. La condition $\mathsf{def}_f(x) \neq \mathsf{def}_f(y)$ est un besoin technique simplifiant le raisonnement à propos des copies parallèles et les blocs de ϕ-instructions (voir discussion plus loin). Grâce à la façon dont les copies fraîches sont introduites lors de la transformation vers CSSA, cette condition additionnelle est vérifiée, par construction, pour toutes les variables apparaissant dans une ϕ-instruction.

En effet, on peut prouver que la forme CSSA générée par l'algorithme satisfait les deux propriétés ci-avant.

Lemme 8.2.5. *Soit f une fonction produite par la transformation vers CSSA. Alors les deux propriétés suivantes sont vérifiées :*

- *f satisfait la propriété des ϕ-ressources disjointes ;*
- *pour tout nœud pc tel que f.fn_phicode(pc) = phib, on a que, pour toute ϕ-instruction $x_0 := \phi(x_1, \ldots, x_n)$ dans* phib*, et tout $0 \leq i, j \leq n$, $i \neq j \implies x_i \perp x_j$.*

Démonstration. La preuve de ce lemme résulte essentiellement de la fraîcheur des nouvelles variables dans les copies insérées. Une fois la propriété de fraîcheur prouvée, la preuve de la non-interférence découle facilement. Nous prouvons d'abord que toute ϕ-ressource (c'est-à-dire les sources et la destination d'une ϕ-fonction) est définie et utilisée une seule fois au même point. Du coup, comme elles ne sont utilisées nulle part ailleurs, elles deviennent mortes en sortant du point de jonction. □

Dans la section 8.3, on utilise ce lemme pour prouver que l'élimination des ϕ-instructions (par fusion de toutes les variables d'une même ϕ-instruction) est correcte.

Discussion. Les deux propriétés ci-avant sont plus fortes que la propriété dite de ϕ-congruence utilisée par Sreedhar et al. Avec ces propriétés, on rend explicite le fait que la génération de CSSA (Method 1) produit des *live-ranges* courts pour les ϕ-ressources. En revanche, la propriété de ϕ-congruence est une propriété globale sur le CFG qui, nous semble-t-il, rendrait un raisonnement local plus difficile. En particulier, la définition de classes de ϕ-congruence [62] permet des cas où $x_d := \phi(z, x_2)$ et $y_d := \phi(y_1, z)$ appartiennent au même ϕ-bloc. Être capable de prouver qu'éliminer ces deux ϕ-intructions est correct demanderait de justifier que x_d et y_d pourraient aussi être remplacés par un représentant commun. Cependant, raisonner sur leur valeur à l'exécution demanderait de considérer deux chemins d'exécution a priori non corrélés. Boissinot et al. ont remarqué qu'il est plus simple de raisonner sur la génération naïve de CSSA, sachant que l'algorithme de *coalescing* appliqué ultérieurement fusionne la plupart des copies introduites. Nos résultats expérimentaux (voir section 8.5) confirment ceci. On décide donc de profiter de ces deux propriétés permettant de raisonner localement sur les points de jonction.

8.3 Destruction sans *coalescing*

Dans cette section, on présente une destruction simple de CSSA vers RTLpar. La syntaxe de RTLpar est la même que celle de CSSA sans le champ `phicode` dans la syntaxe de la fonction :

$$
\begin{array}{lcl}
tf & \ni & \texttt{function}_{\texttt{RTLpar}} \\
tf & ::= & \left\{ \begin{array}{ll} \texttt{name} = id; & \texttt{params} = \vec{x}; \\ \texttt{entry} = pc; & \texttt{code} = \iota; \\ \texttt{parcode} = \mu & \end{array} \right\}
\end{array}
$$

Les sémantiques de RTLpar et CSSA sont similaires, la seule différence étant la sémantique de la règle NopJP, qui est plus simple dans RTLpar du fait de l'absence des ϕ-blocs.

La destruction simple ne fait qu'éliminer les ϕ-instructions en fusionnant toutes les variables qui apparaissent dans une même ϕ-instruction. Ceci nous permet de caractériser les propriétés essentielles qui permettent une élimination des ϕ-instructions. La destruction utilisant du *coalescing* est présentée dans la section suivante comme une extension de celle-ci.

8.3.1 L'algorithme

Ici, afin d'alléger la présentation, on suppose que les fonctions CSSA sont structurellement bien formées, et satisfont le lemme 8.2.5. L'algorithme pour cette destruction simple est le suivant :

$$
\begin{array}{l}
\texttt{destruct}(f) = \\
\quad \text{let } R := \texttt{classes}(f) \text{ in} \\
\quad\quad \{\ \texttt{name} := f.\texttt{name}; \\
\quad\quad\ \ \texttt{params} := \text{map } R\ f.\texttt{params}; \\
\quad\quad\ \ \texttt{entry} := f.\texttt{entry}; \\
\quad\quad\ \ \texttt{code} := pc \mapsto \text{map}_{instr}\ R\ (f.\texttt{code}(pc)); \\
\quad\quad\ \ \texttt{parcode} := pc \mapsto \text{map}_{mov}\ R\ (f.\texttt{parcode}(pc))\}
\end{array}
$$

Tout d'abord, une fonction R est calculée par la fonction `classes`. Cette fonction associe à chaque variable un représentant unique. Ensuite, la destruction consiste à :

- enlever complètement le ϕ-code de la fonction ;
- appliquer la fonction R (vue comme un renommage de variables) à tous les paramètres et instructions de la fonction f (dans le graphe d'instructions f.`code`, ainsi que dans le graphe de blocs de copies parallèles f.`parcode`).

8.3.2 Propriétés du renommage de variables

Plus précisément, la fonction calculée R envoie chaque variable apparaissant comme argument d'une ϕ-instruction sur la destination de la ϕ-instruction. Les autres variables sont envoyées simplement sur elles-mêmes. Notons que ce calcul n'a de sens que si la même variable n'apparaît pas comme ϕ-argument de deux ϕ-instructions différentes (autrement elle serait envoyée dans deux destinations différentes). C'est effectivement le cas avec notre forme CSSA (par la définition 8.2.3). Un autre point important à noter est qu'une analyse de *liveness* n'est pas requise pour cette destruction. En effet, *par construction* de CSSA (lemme 8.2.5), on sait que les *live-ranges* des variables utilisées dans les ϕ-fonctions sont disjoints. Plus formellement, on construit un renommage caractérisé par le lemme-définition suivant :

Lemme-Définition 8.3.1. *Pour toute ϕ-instruction $x := \phi(x_1, \ldots, x_n)$ d'un ϕ-bloc* `phib` *au nœud pc, et pour tout $k \in \{1, \ldots, n\}$, on a $R(x_k) = x$. Pour toutes les variables qui ne sont utilisées dans aucune ϕ-instruction, on a $R(x) = x$.*

On prouve ensuite la propriété principale de notre renommage R, c'est-à-dire que, si une variable est renommée pendant la destruction, alors elle est renommée en une variable qui n'interfère pas avec elle.

Lemme 8.3.2. *Pour chaque paire de variables distinctes x et x' de la fonction f, si $R(x) = R(x')$, alors $x \perp x'$.*

Démonstration. De tels x et x' sont nécessairement ϕ-ressources d'une même ϕ-instruction par le lemme-définition 8.3.1. On conclut en utilisant la propriété des *live-ranges* disjoints assurée par le lemme 8.2.5. □

8.3.3 Preuve de correction

La preuve que la fonction `destruct` est sémantiquement correcte se fait en exhibant une simulation *lock-step* (voir le lemme 3.3.1) entre les deux fonctions. Intuitivement, on veut montrer que les exécutions des deux fonctions correspondent, pas à pas. Pour ce faire, on a besoin de maintenir comme invariant que la valeur de toutes les variables nécessaires (c'est-à-dire les variables vivantes en entrant au point courant du programme) est préservée par le renommage.

Souvent, lorsqu'on raisonne sur de l'analyse de programmes ou d'optimizations SSA, on peut se contenter de préserver un invariant de correction à propos des variables dont la définition domine strictement le point courant *pc* du programme.

Dans le cas présent, on réalise une transformation vers un langage qui n'est plus sous forme SSA et il faut assouplir l'invariant. On a besoin de se retreindre à ne considérer que la préservation des valeurs de l'ensemble des variables vivantes, c'est-à-dire une propriété strictement plus forte. En effet, dès qu'une variable devient morte, on ne peut plus prouver que sa valeur est préservée, même dans une région du code qui est dominée par sa définition, parce que son nom pourrait avoir été fusionné avec celui d'une autre variable.

La relation de simulation $\sim$ entre états d'exécution des fonctions f et f' est définie comme suit :

Définition 8.3.3. Soit $f' = \texttt{destruct}(f)$ la transformée de la fonction f. Alors $(f, \rho, pc) \sim (f', \rho', pc')$ si :

- $pc = pc'$
- $\forall x,\ x \in \mathsf{live}^{\mathsf{in}}_f(pc) \cup Djp_f(pc) \implies \rho(x) = \rho'(R(x))$.

Ici, $Djp_f(pc)$ fait référence à l'ensemble des variables qui sont définies au point de jonction *pc*, c'est-à-dire dans un ϕ-bloc ou dans un bloc de copies parallèles dans un point de jonction. Les variables de cet ensemble ne sont pas forcément vivantes. Du coup, on maintient l'invariant pour un sur-ensemble de ces variables. Techniquement, cela nous permet de travailler avec une définition plus simple de la *liveness* au niveau des points de jonction, en en restant à la granularité du CFG. Nous donnons maintenant les idées principales de la preuve que la relation $\sim$ permet de réaliser une simulation *lock-step* (voir lemme 3.3.1).

Lemme 8.3.4. *Soient σ_1 et σ_2 des états d'exécution de f, et supposons que $\sigma_1 \xrightarrow{t}_{\mathsf{CSSA}} \sigma_2$. Soit σ'_1 un état d'exécution de f' tel que $\sigma_1 \sim \sigma'_1$. Alors, il existe un état σ'_2 tel que $\sigma'_1 \xrightarrow{t}_{\mathsf{RTLpar}} \sigma'_2$ et $\sigma_2 \sim \sigma'_2$.*

Démonstration. La preuve se fait par analyse de cas sur la relation de transition. Nous présentons ici uniquement les cas les plus intéressants. Soit $\sigma_1 = (f, pc, \rho)$ et $\sigma'_1 = (f', pc, \rho')$ deux états satisfaisant les hypothèses du lemme précédent.

Considérons d'abord le cas où σ_1 est tel que l'instruction suivante à exécuter est une simple copie $x := y$. Soit r une variable vivante en sortant de pc (c'est-à-dire, vivante à l'entrée de son successeur). On veut prouver que les environnements locaux correspondent après l'exécution de la copie, c'est-à-dire que $\rho[x \leftarrow y](r) = \rho'[R(x) \leftarrow R(y)](R(r))$. Il y a plusieurs cas à considérer :

- Si $r = x$, la preuve est facile : on peut appliquer à y l'hypothèse d'induction parce que y est vivant à l'entrée de pc, étant utilisé en pc et n'étant pas défini en pc du fait de la propriété de *strictness* du lemme 8.2.2.
- Si $r \neq x$ et $R(r) \neq R(x)$, alors r est déjà vivant en entrant en pc, et la copie ne modifie pas r ou $R(r)$, donc l'égalité résulte de la relation de correspondance entre ρ et ρ'.
- Si $r \neq x$ mais $R(r) = R(x)$, r est comme précédemment déjà vivant à l'entrée de pc, mais la valeur de $R(r)$ pourrait avoir changé. Du fait de $r \neq x$, $R(r) = R(x)$ signifie que $r \perp x$, ce qui est impossible, puisque r est vivant en sortant du point de définition de x.

Le cas le plus difficile est celui où l'instruction en pc est un $\mathtt{nop}(pc')$, et pc' est un point de jonction pc' (règle NopJP dans la figure 8.5), ce qui demande de simuler l'exécution d'un bloc de copies parallèles `parcb` en pc, suivi d'un ϕ-bloc `phib` et d'un bloc de copies parallèles `parcb'` en pc' en un seul pas. En fait, on procède en deux étapes, en prouvant que la relation de correspondance entre environnements locaux est préservée, mais avec certains ajustements vis-à-vis des variables sur lesquelles la correspondance se fait.

Premièrement, on prouve un lemme qui propage les invariants intermédiaires nécessaires après l'application de `parcb`. Principalement, il propage l'invariant sur les environnements pour les variables qui sont vivantes en entrant en pc' ou affectées en `parcb`. Moralement, la simulation d'un bloc de copies parallèles est similaire à celle d'une simple copie. En particulier, on profite du fait qu'avec cette notion d'interférence, on sait qu'après renommage aucune variable n'est affectée deux fois dans le bloc, puisque $x \perp y$ ne peut être vérifié pour des variables définies en un même point. On se repose sur deux lemmes auxiliaires. Le premier lemme caractérise l'effet

d'un bloc de copies parallèles renommé sur le représentant $R(r)$ d'une variable r qui n'est pas affectée dans le bloc `parcb`.

Lemme 8.3.5. *Soit r une variable qui n'est pas une destination dans le bloc de copies parallèles* `parcb`. *Si pour toute copie $x := y$ dans* `parcb` *on a $R(r) \neq R(x)$, alors $[\![map_{mov}\ R\ \texttt{parcb}]\!]\rho'(R(r)) = \rho'(R(r))$.*

Le deuxième lemme est similaire, mais pour des variables qui sont affectées dans le bloc.

Lemme 8.3.6. *Soit r une variable qui soit destination d'une copie $r := y$ du bloc* `parcb`. *Si pour une autre copie $x := y'$ on a $R(r) \neq R(x)$, alors $[\![map_{mov}\ R\ \texttt{parcb}]\!]\rho'(R(r)) = \rho'(R(y))$.*

Ceci signifie, en fait, que les copies d'un bloc de copies parallèles restent en effet parallèles après le renommage avec R.

Nous donnons un aperçu de la preuve de propagation de cet invariant après application de `parcb`. Soit r une variable vivante en entrant au point de jonction pc', successeur de pc. On procède par disjonction de cas en fonction de si la variable est affectée ou non dans `parcb`.

- Si r est affectée dans une copie $r := y$ dans `parcb`, alors $R(r) \neq R(x)$ pour chaque copie $x := y'$ de `parcb` avec $x \neq r$, car autrement r et x seraient définies au même point, donc $r \perp x$ ne pourrait être vérifiée. On peut, du coup, appliquer le lemme 8.3.6.

- Si r n'est pas affecté dans `parcb`, on a aussi $R(r) \neq R(x)$ pour toute copie $x := y'$ de `parcb`, car r est vivant en sortant du point pc (puisque vivant en entrant dans pc'), et x est défini en pc, donc $r \perp x$ ne pourrait pas être vérifiée. On peut donc appliquer le lemme 8.3.5.

Deuxièmement, on propage plus loin la correspondance entre environnements après exécution du ϕ-bloc `phib`, pour les variables qui sont vivantes en pc' ou affectées dans `parcb` ou affectées dans `phib`. Ici, le lemme-définition 8.3.1 est crucial. En effet, le fait que, dans chaque ϕ-instruction, toutes les variables sont envoyées vers le même représentant signifie essentiellement qu'une ϕ-instruction $x_0 := \phi(x_1, \ldots, x_n)$ serait renommée en $x_0 := \phi(x_0, \ldots, x_0)$, ce qui ne modifierait donc pas l'environnement. Ceci justifie l'élimination du ϕ-bloc.

Enfin, on termine la propagation de l'invariant après application du deuxième bloc de copies parallèles `parcb'`, en utilisant un raisonnement similaire qu'avec le premier. □

Cette destruction ne fait qu'éliminer les ϕ-instructions, sans *coalescing* de variables. En comparaison avec la destruction précédente dans CompCertSSA, ceci permet au compilateur d'accepter plus de programmes : il ne va pas échouer dans la compilation d'un programme, même après une optimisation SSA qui romprait les propriétés des *live-ranges* disjoints et des

ϕ-ressources disjointes. Dans la section suivante, on étend cette destruction de sorte à pouvoir effectuer du *coalescing* agressif de variables.

8.4 Destruction CSSA avec *coalescing*

Cette destruction étend la précédente en permettant d'éliminer les copies inutiles. En revanche, on utilise davantage de validateurs *a posteriori* au lieu de preuves directes. Cette approche ne garantit pas formellement que le résultat de l'algorithme est toujours accepté par le validateur : dans nos tests, c'est le cas, comme attendu. Cependant, l'approche donne les mêmes garanties de correction *sur le code généré*, tout en ayant deux avantages principaux : la preuve est généralement plus simple, mais aussi surtout les validateurs sont robustes vis-à-vis d'ajustements concernant les heuristiques des calculs. Par exemple, l'utilisation de priorités de *coalescing*, telles que traiter les points de jonction avant les prédécesseurs comme nous le faisons (car il s'agit d'un chemin d'exécution plus fréquent), n'affecte pas la preuve.

8.4.1 L'algorithme

L'algorithme pour la version étendue de la destruction est le suivant :

```
destruct(f) =
    let live := live_analysis(f) in
    let V_f := cssa_value_ext(f) in
    let ninterfere := ninterfere_test(f, live, V_f) in
    let (R, classes) := build_classes_ext(f, ninterfere) in
    if check(R, classes, ninterfere) && check_v(f,V_f) then
      { name := f.name;
        params := map R f.params;
        entry := f.entry;
        code := pc ↦ map_instr R (f.code(pc));
        parcode := pc ↦ clean(map_mov R (f.parcode(pc)))}
    else Error
```

Comme précédemment, la destruction consiste à calculer un renommage de variables R, puis à l'appliquer à l'ensemble du code de la fonction. Il y a cependant deux différences notables.

Premièrement, R est maintenant calculée par un programme OCaml externe, non prouvé, nommé `build_classes_ext`, dont le résultat est ensuite validé a posteriori par rapport à une spécification, comme indiqué par `check`. Si le validateur passe, alors on continue vers la phase suivante. Sinon, la phase de destruction échoue. En pratique, le validateur n'échoue jamais.

Les détails du calcul des classes, ainsi que des propriétés assurées par le validateur, sont présentées dans la section 8.4.3, et dépendent d'une information de *liveness* pré-calculée (`live`, calculé et prouvé en Coq) et d'une information de CSSA-valeur (V_f, calculée en OCaml et vérifiée a posteriori par un validateur formellement prouvé `check_v ), que l'on définit dans la section suivante.`

Deuxièmement, en plus d'appliquer le renommage de variables et d'éliminer les ϕ-instructions, on élimine quelques copies (`clean`) dans les blocs de copies parallèles renommés. On enlève les copies triviales de la forme $x := x$, c'est-à-dire des copies dont les variables ont été fusionnées. On enlève aussi les copies redondantes, c'est-à-dire des copies qui ont la même destination, mais qui ne sont pas nécessairement triviales (voir 8.4.4). En fait, on doit assurer que de telles copies redondantes sont enlevées avant la phase de sérialisation, de sorte que les blocs de copies parallèles satisfassent la condition *windmill* de Rideau et al. [55].

8.4.2 Non-interférence affinée avec la CSSA-valeur

Depuis les premiers travaux de Chaitin et al. [18], le critère ultime pour décider si deux variables peuvent être fusionnées est le fait que leurs *live-ranges* soient disjoints, ou qu'elles aient la même valeur à l'exécution. Comme le proposent Boissinot et al. [9], la deuxième condition peut être approchée facilement en utilisant la notion de SSA-valeur. La SSA-valeur est une approximation symbolique (c'est-à-dire une expression) de la valeur à l'exécution d'une variable SSA. Elle peut être calculée simplement grâce à la propriété de définition unique de SSA. Ici, comme dans [9], on particularise la SSA-valeur aux copies de variables. Plus formellement :

Définition 8.4.1. La fonction de CSSA-valeur V_f d'une fonction CSSA f est une fonction de l'ensemble des variables dans lui-même qui satisfait les propriétés suivantes :

- Si $x := y$ est une copie (parallèle ou non) de f, alors $V_f(x) = V_f(y)$.
- Si $x := ins(\vec{y})$ où ins n'est pas une copie, alors $V_f(x) = x$.

On calcule une telle fonction V_f en OCaml à l'aide d'un parcours en profondeur du graphe de flot de contrôle de f comme dans Boissinot et al. [9]. Le résultat V_f est validé a posteriori par un validateur vérifié `check_v` qui assure que V_f satisfait la définition 8.4.1.

La propriété principale d'une CSSA-valeur (telle que spécifiée dans la définition 8.4.1), sur laquelle on s'appuie pour prouver la correction de la destruction, est qu'une variable et sa CSSA-valeur s'évaluent vers la même valeur à l'exécution en tout point dominé par le point de définition de la variable.

Lemme 8.4.2. *Pour tout état d'exécution atteignable $\sigma = (f, pc, \rho)$ de la fonction CSSA f, et toute variable r telle que $r > pc$, on a $\rho(r) = \rho(V_f(r))$.*

Démonstration. La preuve se fait par induction sur le nombre de pas d'exécution pour atteindre cet état, avec un raisonnement fondé sur la dominance, tel que présenté dans [28], et en utilisant la propriété que $\text{def}_f(V_f(r))$ domine $\text{def}_f(r)$ qui est prouvée en utilisant la définition 8.4.1 par induction sur les chemins du CFG conduisant à $\text{def}_f(r)$. □

Comme corollaire immédiat on obtient :

Lemme 8.4.3. *Pour tout état d'exécution atteignable $\sigma = (f, pc, \rho)$ d'une fonction CSSA f, pour toutes paires de variables x et y de f telles que $x > pc$, $y > pc$ et $V_f(x) = V_f(y)$, on a $\rho(x) = \rho(y)$.*

Ce lemme est utile pour la preuve de correction de la transformation, pour prouver qu'il est correct de fusionner deux variables avec la même CSSA-valeur. Par exemple, si une variable r est fusionnée avec une variable distincte x apparaissant dans une copie $x := y$, et r est vivant en sortant de ce point, on veut être capable de montrer que la variable issue de la fusion de x et r n'est pas modifiée par cette copie.

On peut maintenant définir la notion étendue de non-interférence de Boissinot et al. :

Définition 8.4.4. Deux variables x et y d'une fonction CSSA f n'interfèrent pas, ce que l'on écrit $x \perp_v y$, si $x \perp y$ ou $V_f(x) = V_f(y)$.

Le test de non-interférence `ninterfere` est implémenté en Coq, en utilisant les calculs de *liveness* et de CSSA-valeur, et est prouvé correct vis-à-vis de la définition 8.4.4.

8.4.3 Propriétés du renommage de variables

L'algorithme de *coalescing* `build_classes_ext` commence par mettre toutes les variables apparaissant dans une même ϕ-instruction dans une même classe. Les autres variables du programme sont des classes singleton. Alors, chaque bloc de copies parallèles est traversé et, pour chaque copie $x := y$, on teste (avec `ninterfere`) si l'on peut fusionner les classes de *coalescing* de x et y, c'est-à-dire, s'il n'y a pas d'interférence entre variables de la classe de *coalescing* de x et y. Le renommage résultant R associe à chaque variable r le représentant de sa classe. Le calcul des classes de *coalescing* fournit deux fonctions : une de l'ensemble des représentants vers les classes de *coalescing* (`classes`), et R. La validation *a posteriori* des classes de coalescing `check` assure que, dans chaque classe, les variables n'interfèrent pas et que chaque variable r appartient à la classe de $R(r)$ (et donc n'interfère avec aucun membre de la classe). Le validateur actuel

check est un algorithme quadratique (en la taille de la classe) qui teste les interférences entre toutes les paires de variables de la classe.

Les propriétés principales établies sur le renommage R par le validateur peuvent être énoncées comme il suit :

Lemme 8.4.5. *Pour chaque paire de variables r et r' d'une fonction CSSA f, si $R(r) = R(r')$, alors $r \perp_v r'$. Pour chaque x, y dans une même ϕ-instruction, on a $R(x) = R(y)$.*

Ce lemme est analogue au lemme-définition 8.3.1. Maintenant on doit prendre en compte la CSSA-valeur et le fait que R peut agir sur des variables non utilisées dans une ϕ-instruction et apparaissant seulement en tant que source ou destination d'un bloc de copies parallèles. La validation *a posteriori* est donc pratique, car plus simple et plus maintenable (des changements sur les heuristiques de *coalescing* ne demandent pas de changer les preuves).

Discutons un peu du lemme 8.4.5 ainsi que de la définition 8.4.4. Pour des variables x et y d'une même ϕ-instruction, la non-interférence est due à $x \perp y$, comme dans le cas sans *coalescing*. Des variables qui ne sont pas utilisées dans une ϕ-instruction, mais qui apparaissent dans des copies parallèles peuvent être ajoutées par l'algorithme à la classe de *coalescing* des variables d'une ϕ-instruction si elles n'interfèrent avec aucune des variables de la classe. Deux ϕ-instructions distinctes de ϕ-blocs distincts peuvent être fusionnées si chaque paire de variables les formant n'interfère pas. Comme extension future, deux ϕ-instructions $x := \phi(x_1, \ldots, x_n)$ et $y := \phi(y_1, \ldots, y_n)$ d'un même ϕ-bloc pourraient éventuellement être fusionnées si pour tout $k \in \{1, \ldots, n\}$, $V_f(x_k) = V_f(y_k)$, mais avec notre définition actuelle d'interférence, les variables x et y interfèrent.

8.4.4 Preuve de correction

La preuve de correction de la destruction étendue suit la même architecture que la version minimale de la section 8.3. On réalise une simulation *lock-step* entre les fonctions source et générées. Il est intéressant de noter que la relation de simulation $\sim$ est la même. La différence dans la preuve réside dans la spécification du renommage de variables qui inclut maintenant la possiblité de fusion de variables qui ont la même CSSA-valeur, ainsi que l'élimination des copies parallèles triviales de la forme $x := x$. Ceci nous demande de généraliser certains lemmes, en particulier ceux qui caractérisent la sémantique des blocs de copies parallèles.

Cas d'une simple instruction de copie. Soit $x := y$ une copie en pc, et r une variable qui est vivante en sortant de pc. On doit prouver $\rho[x \leftarrow y](r) = \rho'[R(x) \leftarrow R(y)](R(r))$.

Le seul cas qui diffère significativement est celui avec $r \neq x$ et $R(r) = R(x)$. Alors r est déjà vivant en entrant en pc, mais la valeur de $R(r)$ pourrait avoir changé. En fait, $R(r) = R(x)$ signifie que r et x n'interfèrent pas, mais comme r est vivant en sortant de pc, cela signifie que r et x n'interfèrent pas (lemme 8.4.5) grâce au fait qu'ils ont la même CSSA-valeur, donc r et y ont la même CSSA-valeur aussi (une copie propage la CSSA-valeur). La propriété de CSSA-valeur du lemme 8.4.3 assure alors que $\rho(r) = \rho(y)$, mais on a aussi $\rho(y) = \rho'(R(y))$ (car y est vivant en pc), ce qui permet de conclure.

Cas d'un prédécesseur de point de jonction. Dans ce cas, pc conduit à un point de jonction pc' (règle NopJP). Comme dans la section 8.3, on doit prouver des propagations intermédiaires de l'invariant $\sim$ après application de `parcb`, `phib` et `parcb'`.

Alors que les invariants CSSA garantissent qu'aucune variable n'est affectée plus d'une fois, ceci n'est plus le cas pour le bloc renommé : plusieurs destinations de copies pourraient être envoyées sur le même représentant, non du fait de *live-ranges* disjoints, mais du fait d'avoir la même CSSA-valeur. En effet, deux copies $x := y$ et $x' := y'$ de `parcb` peuvent être telles que $R(x) = R(x')$ si $V_f(x) = V_f(x')$, ce qui arrive lorsque $V_f(y) = V_f(y')$ (par propagation de CSSA-valeur via des copies). On prouve donc des versions généralisées des lemmes 8.3.5 et 8.3.6 :

Lemme 8.4.6. *Soit r une variable qui n'est pas une destination dans le bloc* `parcb`*. Alors pour toute copie $x := y$ apparaissant dans* `parcb`*, on a $R(r) \neq R(x)$ ou $\rho'(R(r)) = \rho'(R(y))$, donc*

$$[\![clean(map_{mov}\ R\ \texttt{parcb})]\!]\rho'(R(r)) = \rho'(R(r))$$

En particulier, maintenant, dans le cas où $R(r) = R(x)$, le lemme implique que la valeur de $R(r)$ ne change pas.

Lemme 8.4.7. *Soit r une variable qui est la destination d'une copie $r := y$ d'un bloc* `parcb`*. Si pour toute autre copie $x := y'$, on a $R(r) \neq R(x)$ ou $\rho'(R(y')) = \rho'(R(y))$, alors :*

$$[\![clean(map_{mov}\ R\ \texttt{parcb})]\!]\rho'(R(r)) = \rho'(R(y)).$$

Un autre point à noter est que $(\text{clean}(\text{map}_{mov}\ R\ \texttt{parcb}))$ représente le bloc `parcb` auquel non seulement on a appliqué R à toutes les variables, mais auquel on a aussi enlevé les copies triviales de la forme $x := x$. Cette élimination est subtile à justifier : si x apparaît dans le même bloc de copies parallèles en tant que destination d'une autre copie $x := y$, on a besoin de savoir que les sources x et y ont la même valeur (autrement le bloc de copies parallèles ne serait pas bien défini sémantiquement).

La preuve de l'invariant de propagation après application de `parcb` suit le même schéma que dans le cas sans *coalescing*, mais on doit prendre en compte la CSSA-valeur. Par exemple, dans le cas où r est dans une copie $r := y$ dans `parcb` et qu'il existe une autre copie $x := y'$ en `parcb` telle que $R(r) = R(x)$, on veut prouver que $\rho'(R(y')) = \rho'(R(y))$ comme avant. D'après le lemme 8.4.5, r et x n'interfèrent pas. Comme ils sont définis au même nœud, on ne peut pas avoir $r \perp x$ comme dans le cas sans *coalescing*, donc r et x ont la même CSSA-valeur. Par propagation de la CSSA-valeur à travers les copies, y et y' ont la même CSSA-valeur aussi : comme elles sont vivantes en *pc*, on peut appliquer le lemme 8.4.3. On peut conclure en appliquant le lemme 8.4.7. Les autres cas sont similaires.

8.5 Résultats expérimentaux

Les preuves présentées jusqu'ici établissent la correction sémantique de la destruction, mais n'apportent pas d'information sur la *qualité* du *coalescing*. Dans cette section, on évalue les gains pratiques obtenus en intégrant une telle destruction dans CompCertSSA.

À cette fin, on utilise le mécanisme d'extraction de Coq, qui produit du code OCaml à partir de la formalisation, afin d'obtenir une version exécutable du compilateur vérifié CompCertSSA.

Programmes pour les *benchmarks*. On utilise un ensemble de programmes de tests de la suite de tests de CompCert, les *benchmarks* de référence de SPEC2006 et WCET. Ceux-ci représentent autour de 192,600 lignes de code C, chaque programme allant de quelques milliers de lignes à des dizaines de milliers. Ceci inclut des petits programmes comme certaines fonctions cryptographiques comme `aes.c`, ou des programmes plus conséquents, comme des algorithmes de compression, ou le prouveur de logique du premier ordre *spass* (`http://www.spass-prover.org/`) avec des dizaines de milliers de lignes.

Critère d'évaluation. On veut évaluer l'impact de la destruction avec *coalescing* sur le code généré de façon aussi indépendante que possible du reste de la chaîne de compilation. En particulier, évaluer la performance globale du compilateur CompCertSSA en soi serait prématuré et au-delà des prétentions des travaux présentés.

On va donc comparer trois compilateurs différents, mais similaires : CompCert[3] sans le *middle-end* SSA, CompCertSSA en utilisant l'ancienne destruction partielle (deSSA) et CompCertSSA en utilisant la nouvelle destruction complète avec *coalescing* (CSSA).

3. Notre développement est basé sur CompCert 2.1 et nous ne considérons que l'architecture x86.

Un critère possible pour mesurer les effets du coalescing est le temps d'exécution des programmes compilés. Cependant, les programmes dans la suite de *benchmarks* que nous avons ont, pour la plupart, des temps d'exécution trop courts pour rendre les variations observées significatives.

Du coup, on utilise des mesures plus fines : le nombre de copies restantes (pour les deux variantes de CompCertSSA) et l'impact sur le *spilling* et le *reloading* dans la phase ultérieure d'allocation de registres (pour CompCert et les deux variantes de CompCertSSA).

Nombre de copies restantes. Les résultats sont donnés dans la figure 8.6. Pour chaque programme, on donne le nombre de copies parallèles

- introduites lors de la transformation vers CSSA ;
- introduites par la destruction précédente de SSA (deSSA) ;
- restantes dans RTLpar après *coalescing*.

En moyenne, plus de 99% des copies introduites sont éliminées. Plus précisément, en moyenne sur l'ensemble des copies introduites sur tous les programmes de la figure 8.6, on obtient 99.96% de copies éliminées. En moyenne par fichier, 99.93% des copies sont éliminées. En ce qui concerne *spass*, on obtient presque 100% de copies éliminées sur un total de 24574 copies introduites (seulement 4 copies restent). Les résultats sont similaires pour *bzip2* et *raytracer*.

Ce haut pourcentage de copies éliminées peut s'expliquer par le fait que dans le compilateur CompCertSSA actuel, les optimisations n'introduisent que peu d'interférences au niveau des ϕ-blocs. En effet, ceci aurait fait échouer l'ancienne destruction lors de la compilation des programmes.

Une raison qui peut empêcher la fusion de certaines copies, en l'état courant de CompCertSSA, est le cas où deux arguments distincts d'une même ϕ-fonction dans SSA pourraient être vivants en sortant du point de jonction (par exemple après propagation de copie). Alors, ces deux variables interfèrent pour des raisons de *liveness* et ne peuvent donc pas être mises dans la même classe de *coalescing*. Des optimisations faisant apparaître une même variable à plusieurs reprises dans un ϕ-bloc pourraient aussi conduire à des copies non éliminées : par exemple, si une variable r apparaît dans SSA comme argument de deux ϕ-instructions dans un même bloc, $x_d := \phi(\vec{x})$ et $y_d := \phi(\vec{y})$, alors r apparaît comme source de deux copies parallèles dans CSSA et seule l'une d'entre elles sera finalement éliminée. En effet, éliminer les deux copies demanderait de fusionner les classes de x_d et y_d. Actuellement, ceci n'est pas possible avec notre définition de non-interférence. Cependant, comme mentionné dans la section 8.4.3, cette limitation pourrait être surmontée en étendant la propagation de CSSA-valeur à travers les ϕ-fonctions.

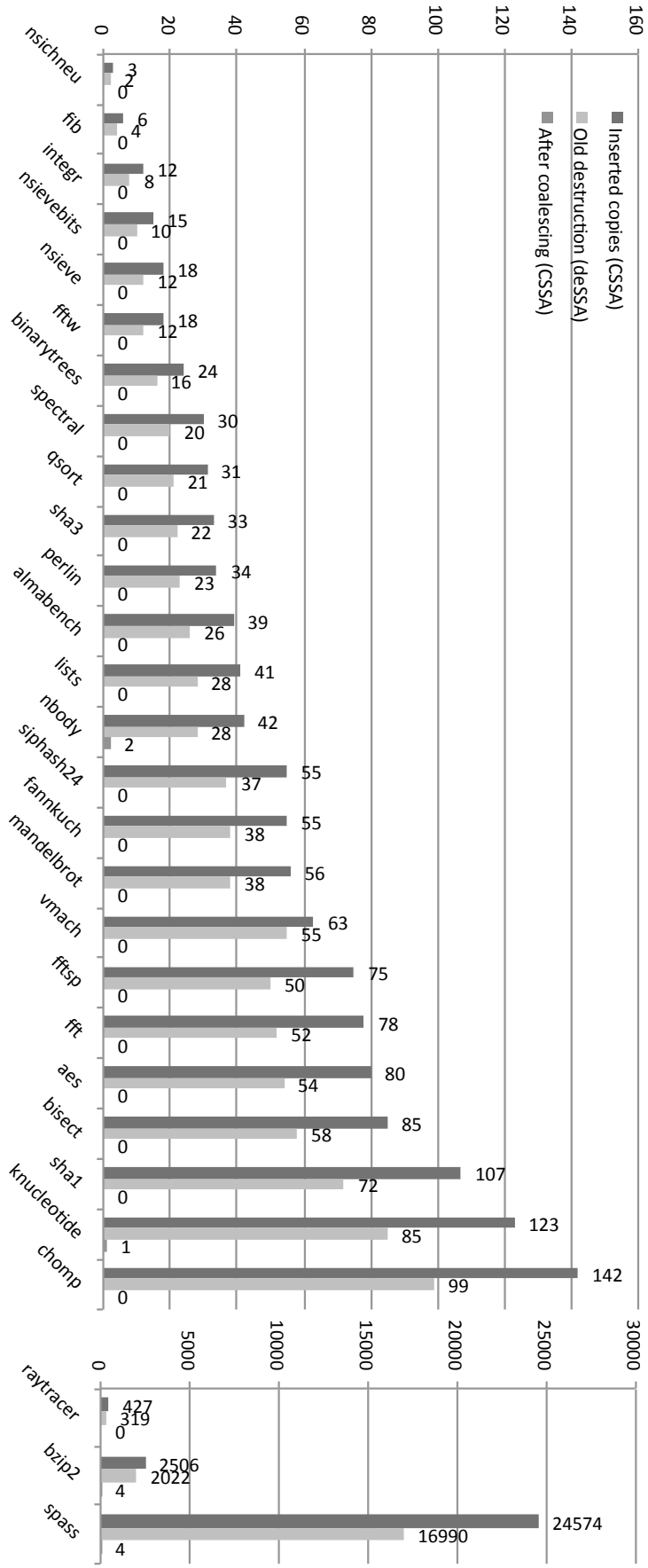


FIG. 8.6 : Résultats du coalescing

***Spilling* et *Reloading*.** La figure 8.7 regroupe des résultats sur le nombre d'instructions de *spilling* et *reloading* pour les programmes *spass*, *raytracer*, *bzip2* et une somme sur les programmes de la suite de tests de CompCert. Pour *spass*, le nombre de *spills* diminue de 4300 à 3453 (une amélioration de 20%), et le nombre de *reloadings* de 6742 à 5430 (une amélioration de 19%). Pour ce qui est de *bzip2*, le nombre de *spills* passe de 2410 à 450 (une amélioration de 81%). Pour *raytracer*, on observe un légère régression de 261 à 271 (une régression de 4%). On observe une diminution des *spills* de 842 à 576 sur la suite de tests de CompCert (32% d'amélioration). En somme, la nouvelle destruction permet d'atteindre des taux de *spilling* et *reloading* qui sont proches de ceux de CompCert 2.1 (sans SSA) : par exemple, 7% de spilling en moins pour *spass*, mais 15% en plus pour *bzip2*. Notons qu'il s'agit d'une amélioration significative par rapport au 520% de *spilling* additionnel qu'on obtenait avec l'ancienne destruction sans *coalescing*.

Discussion. Les résultats en termes de copies restantes sont satisfaisants et, en général, l'amélioration sur le *spilling* et le *reloading* est significative par rapport à la destruction précédente. En fait, la destruction avec *coalescing* conduit à des résultats comparables à ceux de CompCert 2.1. Une régression du *spilling* mineure sur *raytracer* signifie cependant qu'il y a encore de la marge pour des améliorations. En particulier, il serait intéressant d'étudier si ceci est dû à un *coalescing* trop agressif accroissant les *live-ranges* [11, 33]. Enfin, notre validation empirique pourrait bénéficier de *benchmarks* plus poussés, mais ces résultats préliminaires sont encourageants.

8.6 Conclusion et travail futur

Dans ce chapitre, nous avons présenté une formalisation de la destruction de SSA fondée sur une transformation vers le *conventional* SSA, et suivie d'un algorithme de *coalescing*. Tout d'abord, nous avons identifié les propriétés de CSSA utiles pour une destruction sans *coalescing*, en utilisant la propriété des *live-ranges* disjoints des ϕ-ressources et des ϕ-blocs disjoints. Nous avons alors prouvé une extension de cette destruction qui utilise un algorithme de *coalescing* pour éliminer les copies introduites inutiles. Les résultats en termes de copies éliminées sont très satisfaisants et correspondent à nos attentes, avec plus de 99% de copies éliminées et des taux de *spilling* globalement significativement inférieurs.

Taille du développement. La table 8.1 donne un aperçu de la taille du développement Coq, tel que donné par le programme `coqwc`, organisé thématiquement. Nous donnons le nombre de lignes de code source et le

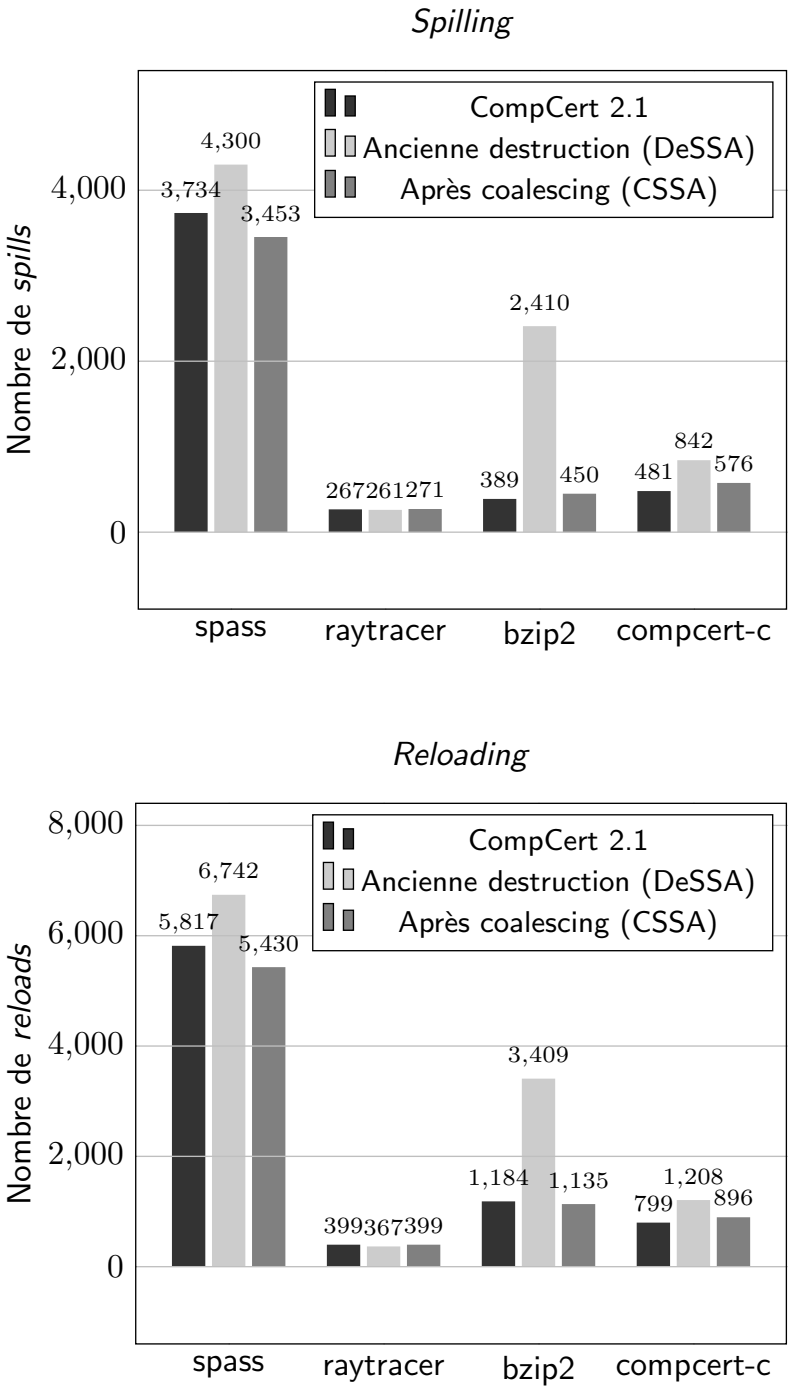


FIG. 8.7 : *Spilling* (haut) et *Reloading* (bas)

Tab. 8.1 : Lignes de code source

	Total	Proofs
Syntaxe et sémantique de CSSA et RTLpar	793	15
Génération CSSA	8372	5545
Destruction sans *coalescing*	3689	2723
Destruction avec *coalescing*	6612	4567
Dé-parallélisation	2657	1453
Lemmes utiles	1407	822
Total	23530	15125

nombre de lignes de preuve. Ceci donne une idée de l'effort de preuve, même si ces nombres ne sont pas toujours corrélés linéairement avec la difficulté de la preuve : certaines preuves sont longues et pénibles sans réelle difficulté technique (comme la génération de CSSA), et d'autres preuves pourraient sans doute être factorisées.

Validation et preuve directe. Nous avons utilisé deux approches pour vérifier nos transformations : la preuve directe des algorithmes en Coq et la validation *a posteriori* d'algorithmes non prouvés par des validateurs formellement prouvés, ce qui donne les mêmes garanties de correction pour le code généré. La validation *a posteriori* conduit parfois à des preuves plus simples et plus maintenables, du fait que la preuve du validateur dépend moins des détails de l'algorithme en soi. Ceci est particulièrement vrai pour l'algorithme de *coalescing*, où nous avons vraiment profité de l'approche par validation lors de l'ajustement des heuristiques de *coalescing*.

Transformations non détaillées. Parmi les preuves non détaillées dans ce chapitre, il y a la préservation sémantique de la génération de CSSA à partir de SSA, et la garantie que les invariants CSSA sont vérifiés. L'essentiel de l'effort se situe dans le raisonnement à propos de la fraîcheur des variables introduites, et aussi dans la preuve de la propriété de définition unique qui nous a demandé de prouver que dans chaque ϕ-bloc les variables apparaissaient une seule fois, mais aussi le fait que deux ϕ-blocs distincts n'ont pas de variables communes. Une autre phase non détaillée ici est la séquentialisation des blocs de copies parallèles. Cette partie revient essentiellement à réutiliser la formalisation de Rideau et al. [55].

Travail futur. Dans ce travail, nous nous sommes focalisés sur la formalisation et la caractérisation des propriétés justifiant la correction de la destruction SSA, laissant au second plan des considérations sur les temps de compilation et l'utilisation mémoire. Tout d'abord, utiliser Coq nous force à utiliser des structures de données purement fonctionnelles qui ne

sont pas optimales, introduisant souvent un facteur logarithmique dans les algorithmes, par exemple en utilisant des structures d'arbres au lieu d'une table de hachage. Nous calculons aussi occasionellement des structures de données sans que cela soit strictement nécessaire. Par exemple, Boissinot et al. [9] et Sreedhar et al. [62] proposent des techniques pour introduire des copies uniquement lorsqu'elles sont vraiment nécessaires, au lieu de les introduire d'abord, puis les éliminer a posteriori. Dans certains cas, nous implémentons des algorithmes non optimaux. En particulier, le calcul (OCaml) et la validation (Coq) des classes de *coalescing* sont actuellement quadratiques en la taille des classes. Cependant, il existe un algorithme linéaire pour tester l'interférence entre deux classes de variables. Par exemple Boissinot et al. [9] proposent un tel algorithme qui utilise un ordre particulier sur les variables dans une classe. Un futur pas pour améliorer notre destruction pourrait être de formaliser un tel algorithme. Un autre exemple est le calcul de l'information de *liveness* : le calcul est fait actuellement par une analyse de flot de données standard, mais comme l'ont montré Boissinot et al. [10], dans le cas particulier de SSA, il est possible d'utiliser un algorithme plus spécifique qui profite des propriétés structurelles particulières de SSA et donne de meilleurs résultats en termes d'utilisation mémoire, tout en restant compétitif en termes de temps de compilation.

Enfin, la destruction SSA que l'on a étudiée dans ce travail est faite avant l'allocation de registres. Bien que cette stratégie soit la plus courante dans les compilateurs courants, d'autres approches ont été proposées. En particulier, il existe des travaux et al. [11, 33] qui proposent de considérer le *spilling* et l'allocation de registres directement sur la forme SSA. Cependant, le *spilling* peut conduire une variable à être chargée en plusieurs points du programme, cassant la propriété de définition unique de SSA et demandant une re-conversion vers SSA à la volée. Quoique cette approche soit intéressante, ce besoin de re-conversion vers SSA lors d'un *spill* casse temporairement les invariants, donc semble problématique à introduire dans CompCertSSA.

Chapitre 9

Conclusion et Perspectives

9.1 Conclusion

Dans cette thèse, nous avons d'abord fourni une syntaxe et sémantique formelles pour une représentation intermédiaire *Sea of Nodes*. La sémantique que nous donnons épouse naturellement la propriété fondamentale SSA, mais aussi, et surtout, la flexibilité en termes de dépendances de données entre instructions. Nous avons eu l'occasion de tester notre sémantique en prouvant la correction d'optimisations. En particulier, nous avons raisonné sémantiquement sur une élimination de *zero-checks* redondants, ce qui a permis de mettre en évidence des propriétés comme la dominance, qui prend une forme particulière dans le contexte de *Sea of Nodes*. La vérification de la propagation de constantes creuse a, quant à elle, permis d'évaluer notre sémantique dans le cadre d'une optimisation classique fondée sur une analyse de flot de données.

D'autre part, nous avons également étudié, d'un point de vue sémantique, des propriétés suffisantes pour une transformation correcte de *Sea of Nodes* vers une représentation SSA utilisant des blocs de base. Enfin, nous avons présenté des travaux, effectués dans le cadre de CompCertSSA, sur une destruction réaliste de la forme SSA. Ces deux étapes devraient être utiles en vue de l'intégration de la forme *Sea of Nodes* dans un compilateur vérifié comme CompCert.

En résumé, nous pensons que cette thèse a permis de mettre en place les fondements nécessaires à la compréhension d'un point de vue sémantique et formel de la forme *Sea of Nodes*. En effet, la mise en évidence de propriétés sémantiques et l'utilisation de celles-ci dans la preuve de préservation sémantique d'optimisations nous encouragent à penser que *Sea of Nodes*, et par extension d'autres représentations similaires, sont abordables et intéressantes d'un point de vue formel. Nous espérons donc que ces travaux seront utiles pour faire avancer l'intégration de techniques avancées

d'optimisation dans les compilateurs vérifiés.

9.2 Perspectives

Dans cette section, nous donnons quelques perspectives sur des pistes possibles pour des travaux futurs.

9.2.1 Travaux en vue de l'intégration dans un compilateur vérifié

Bien que ce manuscrit ait mis en place les bases fondamentales pour envisager la forme *Sea of Nodes* comme représentation intermédiaire dans un compilateur vérifié, un certain nombre de points restent à traiter en vue de l'intégration effective dans un tel compilateur.

Génération de la forme *Sea of Nodes*. Une étape en vue de l'intégration de *Sea of Nodes* dans un compilateur vérifié qui n'est pas traitée dans ce manuscrit est sa génération. Click et al. [21] utilisent un algorithme incrémental pour construire la forme *Sea of Nodes* tout en réalisant des optimisations au moment de l'analyse syntaxique. Les algorithmes incrémentaux sont de manière générale un sujet délicat à étudier d'un point de vue compilation vérifiée, car il est nécessaire de prouver que les modifications incrémentales préservent les invariants du langage. Comme premier pas vers une vérification formelle, une approche plus simple serait de se reposer sur des travaux antérieurs sur la génération SSA [73, 4] et de générer la forme *Sea of Nodes* à partir d'une forme SSA utilisant des blocs de base : la transformation aurait juste à éliminer les dépendances de contrôle superflues et à ne garder que les dépendances de données et de contrôle appropriées. Bien sûr, on s'attend à ce que l'approche de Click avec optimisations incrémentales lors de la construction et n'introduisant pas d'étapes ou langages supplémentaires soit plus performante, donc c'est un sujet qui mérite d'être étudié, malgré la difficulté. En particulier, une propagation de constante simple, élimination de code mort et propagation de copies peuvent être réalisées lors de la construction, ainsi qu'une élimination des nœuds Jmp.

***Global Value Numbering*.** Il serait intéressant d'appliquer notre sémantique pour *Sea of Nodes* à d'autres optimisations, comme l'élimination de sous-expressions communes fondée sur le *Global Value Numbering* [58, 19], ou l'élimination de code mort. Ces optimisations ne sont pas spécifiques à la forme *Sea of Nodes*, mais leur traitement dans le cadre de *Sea of Nodes* pourrait faire ressortir des propriétés spécifiques.

Propagation de constantes creuse. Il serait intéressant de comparer en détail la mécanique des preuves de propagation de constantes creuse du manuscrit à celle des preuves des optimisations correspondantes dans CompCertSSA [28]. Ceci dit, il nous est déjà possible de faire deux remarques. Premièrement, *Sea of Nodes* permet d'écrire la version conditionnelle comme extension naturelle de la propagation simple, sans avoir à introduire d'étiquettes sur les arcs. Deuxièmement, la preuve que nous donnons est essentiellement ad hoc, alors que la preuve de CompCertSSA utilise un cadre plus général pour les analyses creuses dans SSA.

Preuves Coq. Comme nous l'avons vu, certaines transformations liées à *Sea of Nodes*, n'ont pas fait l'objet d'un développement Coq. Un tel développement serait bien sûr nécessaire en vue de finaliser l'intégration dans un compilateur vérifié. Nous faisons donc un bilan de l'état des développements effectués et manquants.

D'une part, nous donnons les sujets principaux qui ont fait l'objet d'un développement en Coq :

- la syntaxe et sémantique pour *Sea of Nodes* présentées au chapitre 4 ;
- la dominance au sens *Sea of Nodes* pour les nœuds de données, et le théorème 5.2.6 du chapitre 5, ainsi que tous les lemmes intermédiaires (sauf la dominance à la CompCertSSA sur le CFG donnée à la définition 5.2.2, qui est elle axiomatisée) ;
- les propriétés sémantiques nécessaires à la preuve de préservation sémantique d'une élimination de *zero-checks* du moment qu'elle vérifie le critère de redondance du lemme 6.1.5 du chapitre 6 ;
- la préservation sémantique d'une analyse simple de constantes creuse satisfaisant la spécification utilisée en section 6.2 ;
- tous les résultats du chapitre 8 concernant la destruction vérifiée de SSA dans CompCertSSA.

D'autre part, voici les développements Coq qu'il reste encore à effectuer :

- Le lemme 5.2.5 du chapitre 5, concernant l'évaluabilité des nœuds de données et dont aucun autre résultat prouvé ne dépend ;
- la preuve — directe ou par validation a posteriori — que les résultats donnés par les algorithmes d'élimination de *zero-checks* et de propagation de constantes satisfont les spécifications ; en particulier, il s'agit du critère syntaxique de redondance du lemme 6.1.5 pour le premier ;
- la correction de la version conditionnelle de la propagation de constantes détaillée à la section 6.3 ;

- l'ensemble des sujets traités au chapitre 7 concernant le retour au bloc de base ;
- la construction de la forme *Sea of Nodes*.

Résultats expérimentaux. Les efforts pour adapter les techniques de compilation au contexte de la compilation vérifiée ne permettent parfois pas une traduction directe de ces techniques. Notamment, des compromis vis-à-vis des temps de compilation ou utilisation mémoire sont souvent faits : utilisation de structures de données purement fonctionnelles introduisant un facteur logarithmique par rapport à leurs équivalents impératifs ; validateurs a posteriori pouvant potentiellement être coûteux ; introduction de langages intermédiaires et étapes supplémentaires qui facilitent le raisonnement mais ne permettent pas de profiter de certaines transformations et optimisations incrémentales. L'ensemble de ces points signifie qu'une évaluation expérimentale d'une implémentation vérifiée de *Sea of Nodes* serait intéressante, car elle apporterait potentiellement des résultats différents de ceux attendus pour une implémentation non vérifiée des mêmes techniques.

9.2.2 Extension à d'autres représentations similaires

Plus généralement et au-delà des sujets traités dans ce manuscrit, nous espérons que les résultats présentés pourront être utiles pour étudier certaines des extensions de SSA [67]. En particulier, nous pensons que des extensions mentionnées au chapitre 2, par exemple une forme SSI [60] utilisant un graphe à la *Sea of Nodes*, pourraient utiliser ce travail comme point de départ. D'autres représentations plus éloignées, comme le *Value State Dependence Graph* [39], bien qu'utilisant des considérations similaires pour les dépendances de données, demanderont sans doute une approche significativement différente pour la partie contrôle ; on peut espérer cependant que, pour la partie dépendances de données, notre approche pourra servir d'inspiration.

Bibliographie

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, 1970.

[3] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, NY, USA, 2004.

[4] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM TOPLAS*, pages 1–35, 2014.

[5] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, Workshop TYPES 2004*, Lecture Notes in Computer Science, pages 66–81, 2006.

[6] S. Blazy, D. Demange, and D. Pichardie. Validating dominator trees for a fast, verified dominance test. In *Proc. of ITP'15*, LNCS, 2015.

[7] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, pages 200–209, 2005.

[8] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing code generation from SSA form : A comparison between two formal correctness proofs in Isabelle/HOL. In *Proc. of COCV'05*, ENTCS, pages 33–51, 2005.

[9] B. Boissinot, A. Darte, F. Rastello, B. Dupont de Dinechin, and C. Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proc. of CGO'09*, pages 114–125, 2009.

[10] B. Boissinot, S. Hack, D. Grund, B. Dupont de Dine hin, and F. Rastello. Fast liveness checking for SSA-form programs. In *Proc. of CGO'08*, pages 35–44, 2008.

[11] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation and spill complexity under ssa, 2005.

[12] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In *Proc. of CC'13*, LNCS, pages 102–122, 2013.

[13] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical report, Karlsruhe Institute of Technology, 2011.

[14] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, pages 859–881, 1998.

[15] P. Briggs, K.D. Cooper, and L.T. Simpson. Value numbering. *SPE*, pages 701–724, 1997.

[16] S. Buchwald, D. Lohner, and S. Ullrich. Verified Construction of Static Single Assignment Form. In *Proc. of CC'16*, pages 67–76, 2016.

[17] Cakeml. A verified implementation of ML, 2012. https://cakeml.org/.

[18] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, pages 47 – 57, 1981.

[19] C. Click. Global code motion / global value numbering. In *Proc. of PLDI'95*, pages 246–257, 1995.

[20] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM TOPLAS*, pages 181–196, 1995.

[21] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Proc. of IR'95*, pages 35–49, 1995.

[22] Companion web page, 2012. http://compcertssa.gforge.inria.fr.

[23] K. Cooper, T. Harvey, and K. Kennedy. A fast, simple dominance algorithm, 2006.

[24] Site de l'assistant de preuve Coq, 1999. https://coq.inria.fr/.

[25] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, pages 451–490, 1991.

[26] D. Demange and Y. Fernandez de Retana. Mechanizing Conventional SSA for a Verified Destruction with Coalescing. In *Proc. of CC'16*, CC 2016, pages 77–87, 2016.

[27] D. Demange, Y. Fernandez de Retana, and D. Pichardie. Semantic Reasoning about the Sea of Nodes. In *Proc. of CC'18*, CC 2018, 2018.

[28] D. Demange, L. Stefanesco, and D. Pichardie. Verifying Fast and Sparse SSA-based Optimizations in Coq. In *Proc. of CC'15*, LNCS, pages 233–252, 2015.

[29] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proc. of VMIL'13*, pages 1–10, 2013.

[30] B. Dupont de Dinechin. Using the SSA-form in a code generator. In *Proc. of CC'14*, LNCS, pages 1–17, 2014.

[31] N. E. Johnson. Code size optimization for embedded processors. 01 2004.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, pages 319–349, 1987.

[33] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *Proc. of CC'06*, LNCS, pages 247–262, 2006.

[34] P. Havlak. Construction of thinned gated single-assignment form. In *Languages and Compilers for Parallel Computing*, pages 477–499, 1994.

[35] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, pages 422–448, 1983.

[36] Hol interactive theorem prover, 2012. https://hol-theorem-prover.org/.

[37] Homepage of the hotspot project, 1999. http://openjdk.java.net/groups/hotspot.

[38] Site de l'assistant de preuve Isabelle, 1986. https://isabelle.in.tum.de/.

[39] N. Johnson and A. Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Proc. of CC'03*, CC 2003, pages 1–16, 2003.

[40] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. In *Proc. of ASPLOS'00*, pages 139–149, 2000.

[41] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pages 13–22, 1995.

[42] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, 1973.

[43] K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, 1998.

[44] R. Leissa, M. Köster, and S. Hack. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 202–212, 2015.

[45] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, pages 121–141, 1979.

[46] X. Leroy. A formally verified compiler back-end. *JAR*, pages 363–446, 2009.

[47] W. M. McKeeman. Peephole optimization. *Commun. ACM*, pages 443–444, 1965.

[48] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 448–461, 2016.

[49] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, 1997.

[50] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, 2000.

[51] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[52] Go nilcheck elimination source code, 2017. https://github.com/golang/go/blob/release-branch.go1.9/src/cmd/compile/internal/ssa/nilcheck.go.

[53] M. Paleczny, C. Vick, and C. Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, 2001.

[54] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, 1998.

[55] L. Rideau, B.P. Serpette, and X. Leroy. Tilting at windmills with Coq : Formal verification of a compilation algorithm for parallel moves. *JAR*, pages 307–326, 2008.

[56] S. Rideau and X. Leroy. Validating register allocation and spilling. In *Compiler Construction (CC 2010)*, pages 224–243, 2010.

[57] J. Rosemann, S. Schneider, and S. Hack. Verified spilling and translation validation with repair. In *Interactive Theorem Proving*, pages 427–443, 2017.

[58] B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proc. of POPL'88*, pages 12–27, 1988.

[59] S. Schneider, G. Smolka, and S. Hack. A linear first-order functional intermediate language for verified compilers. In *Interactive Theorem Proving*, pages 344–358, 2015.

[60] J. Singer. Static program analysis based on virtual register renaming. Technical report, 2006.

[61] Raisonnement sémantique sur sea of nodes, 2018. https://www.irisa.fr/celtique/ext/sea-of-nodes/.

[62] V.C. Sreedhar, R. Ju, D.M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proc. of SAS'99*, pages 194–210, 1999.

[63] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for cakeml. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 60–73, 2016.

[64] Homepage of the GCC project, 1987. https://gcc.gnu.org/.

[65] Homepage of the Go project, 2012. https://golang.org/.

[66] Homepage of the LLVM project, 2003. https://llvm.org/.

[67] Static single assignment book, 2015. http://ssabook.gforge.inria.fr/latest/book.pdf.

[68] J. Tristan and X. Leroy. Formal verification of translation validators : A case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 17–27, 2008.

[69] J.B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proc. of PLDI'11*, pages 295–305, 2011.

[70] J.B. Tristan and X. Leroy. Verified validation of lazy code motion. In *Proc. of PLDI'09*, pages 316–326, 2009.

[71] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM TOPLAS*, 1991.

[72] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, 2011.

[73] J. Zhao. *Formalizing an SSA-based compiler for verified advanced program transformations*. PhD thesis, University of Pennsylvania, 2013.

[74] J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. of PLDI'13*, pages 175–186, 2013.

[75] J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. Formalizing the LLVM intermediate representation for verified program transformation. In *Proc. of POPL'12*, pages 427–440, 2012.