Boris Yakobowski

DEA Programmation

Étude sémantique d'un langage intermédiaire de type Static Single Assignment

sous la direction de Xavier Leroy

Stage effectué à l'INRIA Rocquencourt
de Avril à Septembre 2004

Table des matières

Introduction **4**
Contexte de ce travail 5

Notations **6**

1 Langage intermédiaire **7**
1.1 Syntaxe du langage 7
1.2 Sémantique opérationnelle 8
1.3 Chemin d'exécution 9
1.4 Fonctions correctes 11
1.5 Raisonnement sur les fonctions 12
1.5.1 Notion d'équivalence observationnelle 12
1.5.2 Principes de preuves 12

2 Sémantique de SSA **14**
2.1 Notions de SSA 14
2.2 Fonctions ϕ dans le langage RTL 15
2.3 Propriétés de la forme SSA 16

3 Passage en forme SSA **17**
3.1 Idées générales 17
3.2 Renommage d'un bloc 18
3.3 Insertion des fonctions ϕ 20
3.4 Commentaires 22
3.4.1 Restriction sur la définition des registres 22

4 Transformations sur la forme SSA **23**
4.1 Substitution textuelle 23
4.2 Propagation de constantes et de copies 24
4.3 Simplification d'instructions ϕ 25
4.4 Élimination de code mort 26
4.5 Algorithme général par liste de tâches 27
4.5.1 Algorithme général 27
4.5.2 Instanciation 28
Propagation de constantes ou de registres 28

Suppression de code mort . . . 28
Simplification des instructions ϕ . . . 28
Autres algorithmes . . . 29
"Mélange" des algorithmes . . . 29
4.6 Élimination des sous-expressions communes . . . 29
4.6.1 Ordre de parcours . . . 31
4.6.2 Remarques . . . 32
4.7 Discussion . . . 32
4.7.1 Séparation de variables . . . 32
4.7.2 Optimisations conditionnelles . . . 32

5 Sortie de la forme SSA **34**
5.1 Algorithme . . . 34
5.2 Remarques . . . 35

6 Discussion **36**
6.1 Travaux connexes . . . 36
6.2 Formalisation en Coq . . . 36
6.3 Conclusion . . . 37

Bibliographie **38**

Introduction

La forme SSA (*static single assignment*) est une forme intermédiaire de compilation, dans laquelle chaque variable d'une fonction ne peut être assignée statiquement qu'une seule fois.

En forme SSA, certaines informations sur une fonction peuvent être obtenues plus facilement, et prennent moins d'espace. C'est typiquement le cas des *reaching definitions*, qui lient les définitions d'une variable et les emplacements dans le code qu'elles atteignent. Dans une fonction "traditionnelle", ces informations prennent une place en $O(n^2)$, n étant la taille de la fonction ; en forme SSA, la place nécessaire (ainsi que le temps nécessaire pour les générer) sont en $O(n)$.

Par ailleurs, de nombreux algorithmes d'optimisations sont plus efficaces en forme SSA, au moins au niveau complexité. De plus, les optimisations sont également parfois plus efficaces.

Toutefois, bien que la forme SSA soit de plus en plus utilisée dans les compilateurs modernes[1], sa sémantique est mal comprise. Par exemple, après certaines transformations, les algorithmes utilisés pour sortir de la forme SSA généraient un code incorrect, qui n'avait pas le même comportement que le programme d'origine.

De façon générale, un programme en forme SSA doit vérifier certaines conditions, qui sont souvent insuffisamment mises en avant dans les textes de référence. De même, toutes les optimisations ne sont pas licites en forme SSA.

Dans ce document, on introduit une sémantique SSA pour un langage intermédiaire de type RTL (*register transfer langage*). On énonce les propriétés qu'une fonction doit vérifier pour être en forme SSA, et on propose :

- Un algorithme transformant une fonction RTL en une fonction SSA équivalente.
- Plusieurs algorithmes d'optimisation sur les fonctions en forme SSA (propagation de constantes, suppression de copies, suppression de code mort, élimination de sous-expressions communes).
- Un algorithme transformant une fonction en forme SSA en une fonction RTL équivalente.

Ces algorithmes étant à terme destinés à être incorporés dans un compilateur optimisant prouvé, nous nous sommes efforcés de choisir les solutions les plus adaptées à la preuve formelle (souvent les plus simples et les plus décomposables possibles) ;

[1]La version 3.5 de GCC, le compilateur libre du projet GNU, devrait utiliser la forme SSA pour certaines de ses optimisations.

toutefois nous pensons qu'aucun de nos choix n'est vraiment rédhibitoire comparé à des algorithmes plus évolués.

Le but du travail présenté ici était donc triple :

1. Trouver une sémantique acceptable pour SSA.
2. Trouver dans la littérature ou inventer des algorithmes travaillant sur des fonctions en forme SSA, et préservant la sémantique.
3. Prouver la correction de ces algorithmes, en gardant à l'esprit la nécessité de les prouver formellement ensuite.

Contexte de ce travail

Ce stage a été encadré par Xavier Leroy, dans le projet Cristal ; toutefois ce travail s'inscrit plus particulièrement dans le cadre de l'ARC Concert.

L'équipe Cristal travaille à la conception de langages de programmation et à leur formalisation, avec pour objectif principal d'accroître la robustesse des applications informatiques et la rapidité de leur développement. Ceux-ci sont accrus par l'utilisation de langages de programmation expressifs et sûrs, et les recherches menées au projet Cristal ont pour objectif de proposer de tels langages ainsi que d'étudier formellement leurs propriétés. L'équipe Cristal se compose actuellement de 7 chercheurs permanents, 3 chercheurs détachés, et 1 doctorant.

L'action de recherche coopérative Concert a pour objectif principal de déterminer s'il est faisable, dans l'état actuel des connaissances, de réaliser un compilateur réaliste qui soit certifié, c'est-à-dire accompagné d'une preuve Coq d'équivalence sémantique entre le code source et le code machine engendré. C'est une action concertée entre le CNAM-I.I.E., et les projets Cristal, Lemme, Mimosa, Oasis et Miró de l'INRIA.

Dans le cadre de l'action Concert, ce stage a pour but d'étudier d'éventuelles simplifications algorithmiques sur les optimisations faites par le compilateur. Ces optimisations sont actuellement effectuées par des algorithmes de data-flow, dont l'efficacité laisse à désirer ; la forme SSA, par les simplifications qu'elle génère, pourrait permettre de s'affranchir de ces problèmes.

Toutefois, un des objectifs du stage est également de voir si ces simplifications ne s'accompagnent pas d'une plus grande complexité lors de l'implémentation et de l'écriture des preuves de correction.

Une version préliminaire de ce travail a été présentée lors d'une réunion Concert ayant eu lieu au moins de Juin 2004.

Notations

Dans tout le document, certains noms de variables sont implicitement quantifiés sur un type ; quand c'est le cas, on s'autorise à ne pas quantifier explicitement dans les formules. Par exemple, la lettre n varie sur l'ensemble $\mathbb{N}$ des entiers naturels ; les autres conventions seront introduites au fur et à mesure.

On note $\textsc{Dom}(f)$ et $\textsc{CoDom}(f)$ le domaine et le codomaine d'une fonction, $\circ$ la composition de deux fonctions, et $f|_D$ la restriction de f au domaine D. La fonction valant f sauf en v où elle vaut e est notée $f[v := e]$. La fonction constante égale à e est notée $(_ \mapsto e)$, l'identité Id.

Étant donné un ensemble E, on note $\mathsf{List}(E)$ l'ensemble des listes (ordonnées) d'éléments de E. La liste vide est notée $[]$. La liste constituée de e puis des éléments de l est notée $[e; l]$; on abrège $[e_1; [...; [e_n; l]]]$ en $[e_1; ...; e_n; l]$. Étant donné une liste l de longueur $|l|$, son $n$ième élément ($1 \leq n \leq |l|$) est $l.(n)$. La liste $l[n := e]$ est égale à l, sauf pour l'élément n qui est e. La concaténation sur les listes est notée @.

On définit un prédicat[2] d'appartenance syntaxique $\Subset$ sur les éléments d'ensembles construits (par exemple les listes). Ainsi on écrira $e \Subset l$ si et seulement si e est un des éléments de l *ou* si e apparaît dans un des éléments de l.

De même, on définit une opération de substitution syntaxique, notée également $e[e' := e'']$, qui remplace toutes les occurrences de e' dans e par e''.

[2]Non typé.

Chapitre 1

Langage intermédiaire

Dans cette partie on décrit le langage intermédiaire sur lequel on va raisonner pour les preuves. Ce langage est un langage 3 adresses, ou RTL (*Register Transfer Langage*) pour un processeur RISC : toutes les opérations de calcul se font sur les registres, et on ajoute deux instructions *load* et *store* pour accéder à la mémoire.

Dans le cadre de l'étude de SSA, on peut ne pas définir tout le langage assembleur du processeur, qui n'a pas grand intérêt. Pour cela on se contente de supposer l'existence d'un ensemble d'opérateurs binaires, qui contient par exemple les opérations arithmétiques usuelles (addition, multiplication...).

1.1 Syntaxe du langage

Le langage est présenté Fig. 1.1. Une fonction RTL est constituée d'une fonction partielle de l'ensemble des étiquettes dans celui des blocs, et d'un nombre de paramètres. Par convention, les registres r_1 à r_n d'une fonction à n arguments sont les paramètres de la fonction.

On définit des fonctions d'accès aux éléments d'un bloc par $\textsc{Instrs}(I, j) = I$ et $\textsc{Branch}(I, j) = j$. Par ailleurs, étant donné $F = (B, n)$, on confond la plupart du temps F et B.

Une fonction est dite *cohérente* lorsque toutes les étiquettes apparaissant dans une instruction de branchement font partie du domaine de la fonction : $\mathsf{Coherent}(F) \equiv \forall l, l \Subset F \implies l \in \textsc{Dom}(F)$. Dans la suite on ne considère que des fonctions cohérentes.

Par convention, on suppose que l_{init} est toujours le premier bloc d'une fonction à être exécuté. On suppose également qu'aucune instruction de branchement n'a pour destination l_{init}. Enfin, on suppose l'existence d'une étiquette $l_{\neq}$ telle que quelle que soit F, $l_{\neq} \notin \textsc{Dom}(f)$.

Étant donné une fonction F, on désigne par $F.l.n$ la $n$ième instruction linéaire du bloc dont l'étiquette est l si cette dernière existe, et l'instruction de branchement sinon. Par homogénéité, on note $F.l.\omega$ l'instruction de saut du bloc dont l'étiquette est l. La variable τ parcourt l'ensemble Loc des valeurs de la forme $l.n$, avec $= n \in \mathbb{N} \cup \{\omega\}$;

$$
\begin{array}{llr}
i \in \mathsf{LinInstrs} & ::= () & (\textsc{NOp}) \\
 & \mid\ r \leftarrow \mu & (\textsc{Copy}) \\
 & \mid\ r \leftarrow \mu \star \mu & (\textsc{Op}) \\
 & \mid\ r \leftarrow F(\mu, .., \mu) & (\textsc{Call}) \\
 & \mid\ r \leftarrow [\mu] & (\textsc{Load}) \\
 & \mid\ [\mu] \leftarrow \mu & (\textsc{Store}) \\
j \in \mathsf{BranchInstrs} & ::= \rightarrow\!\mid \mu & (\textsc{Return}) \\
 & \mid\ \rightsquigarrow l & (\textsc{Jump}) \\
 & \mid\ \rightsquigarrow \mu^{?} : l, l & (\textsc{CondJump}) \\
b \in \mathsf{Blocks} & = \mathsf{List(LinInstrs)} \times \mathsf{BranchInstrs} & \\
F \in \mathsf{Functions} & = \mathsf{Blocks}^{\mathsf{Labels}} \times \mathbb{N} &
\end{array}
$$

$$
\begin{array}{l}
l \in \mathsf{Labels} \\
\star \in \mathsf{Ops} \\
r \in \mathsf{Regs} \\
v \in \mathsf{Vals} \\
\mu \in \mathsf{Regs} \cup \mathsf{Vals} \\
\iota \in \mathsf{Instrs} = \mathsf{LinInstrs} \cup \mathsf{BranchInstrs}
\end{array}
$$

Fig. 1.1 – Syntaxe du langage

on appelle ces valeurs *emplacements*. On note $\tau_\bullet = l_{\text{init}}.1$; et $\tau(F)$ l'ensemble des emplacements correspondants à des instructions de la fonction F.

On définit RDef et RUse (registres définis et registres utilisés) par :

$$\mathsf{RDef}(r \leftarrow ...) = \{r\} \quad \mathsf{RDef}([\mu] \leftarrow \ldots) = \emptyset \quad \mathsf{RDef}(j) = \emptyset$$

$$\mathsf{RUse}(r \leftarrow e) = \{ r \mid r \Subset e \} \quad \mathsf{RUse}([\mu_m] \leftarrow \mu) = \{ r \mid r \Subset \mu_m \} \cup \{ r \mid r \Subset \mu \}$$
$$\mathsf{RUse}(\rightarrow\!\mid \mu) = \{ r \mid r \Subset \mu \} \quad \mathsf{RUse}(\rightsquigarrow l) = \emptyset \quad \mathsf{RUse}(\rightsquigarrow \mu^{?} : l_T, l_F) = \{ r \mid r \Subset \mu \}$$

On définit également RDef et RUse en tant que sous-ensembles de Instrs par :

$$\mathsf{RDef}(r) = \{ \iota \mid r \in \mathsf{RDef}(\iota) \} \qquad \mathsf{RUse}(r) = \{ \iota \mid r \in \mathsf{RUse}(\iota) \}$$

1.2 Sémantique opérationnelle

On modélise l'état des registres et de la mémoire à un instant t de l'exécution par des fonctions totales, mais à image dans $\mathsf{Vals} \cup \{\textsc{Bad}\}$; on note $\mathcal{R}_{\textsc{Bad}}$ la fonction qui à tout registre associe Bad. Bad est une valeur d'erreur, qui correspond à l'arrêt prématuré d'une fonction (division par 0, accès à une zone mémoire non initialisée...). Les opérateurs sont donc vus comme des fonctions totales de $\mathsf{Vals} \times \mathsf{Vals}$ dans $\mathsf{Vals} \cup \{\textsc{Bad}\}$. On étend les registres aux éléments de Vals en posant $\mathcal{R}(v) = v$

La sémantique opérationnelle est donnée sous la forme de prédicats d'évaluations mutuellement récursifs, sur les instructions (Fig. 1.2) et les blocs de fonctions (Fig 1.3). Le résultat de l'exécution d'une fonction est un état mémoire et une valeur de retour ; l'état final des registres est ignoré.

$$\frac{}{\langle \mathcal{R}, \mathcal{M}, l_{-}, () \rangle \to \langle \mathcal{R}, \mathcal{M} \rangle}\ \textsc{SNOp} \qquad \frac{\mathcal{R}(\mu) = v}{\langle \mathcal{R}, \mathcal{M}, l_{-}, r \leftarrow \mu \rangle \to \langle \mathcal{R}[r := v], \mathcal{M} \rangle}\ \textsc{SCopy}$$

$$\frac{\mathcal{R}(\mu_1) = v_1 \qquad \mathcal{R}(\mu_2) = v_2 \qquad v_1 \star v_2 = v}{\langle \mathcal{R}, \mathcal{M}, l_{-}, r \leftarrow \mu_1 \star \mu_2 \rangle \to \langle \mathcal{R}[r := v], \mathcal{M} \rangle}\ \textsc{SOp}$$

$$\frac{\mathcal{R}(\mu) = v \qquad \mathcal{M}(v) = v'}{\langle \mathcal{R}, \mathcal{M}, l_{-}, r \leftarrow [\mu] \rangle \to \langle \mathcal{R}[r := v'], \mathcal{M} \rangle}\ \textsc{SLoad}$$

$$\frac{\mathcal{R}(\mu_m) = v_m \qquad \mathcal{R}(\mu) = v}{\langle \mathcal{R}, \mathcal{M}, l_{-}, [\mu_m] \leftarrow \mu \rangle \to \langle \mathcal{R}, \mathcal{M}[v_m := v] \rangle}\ \textsc{SStore}$$

$$\frac{\forall n,\ \mathcal{R}(\mu_n) = v_n \qquad \langle \mathcal{M}, (v_1, ..., v_n), F \rangle \Mapsto \langle \mathcal{M}', v \rangle}{\langle \mathcal{R}, \mathcal{M}, l_{-}, r \leftarrow F(\mu_1, ..., \mu_n) \rangle \to \langle \mathcal{R}[r := v], \mathcal{M}' \rangle}\ \textsc{SCall}$$

$$\frac{}{\langle \mathcal{R}, \mathcal{M}, l_{-}, [] \rangle \to^* \langle \mathcal{R}, \mathcal{M} \rangle}$$

$$\frac{\langle \mathcal{R}, \mathcal{M}, l_{-}, i \rangle \to \langle \mathcal{R}', \mathcal{M}' \rangle \qquad \langle \mathcal{R}', \mathcal{M}', l_{-}, L \rangle \to^* \langle \mathcal{R}'', \mathcal{M}'' \rangle}{\langle \mathcal{R}, \mathcal{M}, l_{-}, [i; L] \rangle \to^* \langle \mathcal{R}'', \mathcal{M}'' \rangle}$$

$$\frac{\mathcal{R}(\mu) = v \neq 0}{\langle \mathcal{R}, \rightsquigarrow \mu^? : l_T, l_F \rangle \rightsquigarrow l_T}\ \textsc{SCondJumpT} \qquad \frac{\mathcal{R}(\mu) = 0}{\langle \mathcal{R}, \rightsquigarrow \mu^? : l_T, l_F \rangle \rightsquigarrow l_F}\ \textsc{SCondJumpF}$$

$$\frac{}{\langle \mathcal{R}, \rightsquigarrow l \rangle \rightsquigarrow l}\ \textsc{SJump} \qquad \frac{\mathcal{R}(\mu) = v}{\langle \mathcal{R}, \rightarrow\!| \ \mu \rangle \rightarrow\!| \ v}\ \textsc{SReturn}$$

FIG. 1.2 – Sémantique opérationnelle pour les instructions.

Une différence notable par rapport aux sémantiques usuelles est qu'on mémorise l'étiquette du bloc duquel on “arrive”. Cette information n'est pas utile pour les instructions qu'on présente dans cette partie, mais sera utilisée par les fonctions en forme SSA.

1.3 Chemin d'exécution

On peut représenter statiquement l'exécution d'une fonction par la liste des instructions qui sont exécutées dynamiquement. Pour cela, étant donné une fonction F et un emplacement, on définit une notion d'emplacements successeurs, qui modélisent le flot

$$\frac{\langle \mathcal{R}, \mathcal{M}, F, l_-, \textsc{Instrs}(F(l)) \rightarrow^* \langle \mathcal{R}', \mathcal{M}' \rangle \qquad \langle \mathcal{R}', \textsc{Branch}(F(l)) \rangle \rightsquigarrow l'}{\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \mapsto \langle \mathcal{R}', \mathcal{M}', l' \rangle} \textsc{SBlockJump}$$

$$\frac{}{\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \mapsto^* \langle \mathcal{R}, \mathcal{M}, l_-, l \rangle}$$

$$\frac{\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \mapsto \langle \mathcal{R}', \mathcal{M}', l' \rangle \qquad \langle \mathcal{R}', \mathcal{M}', F, l, l' \rangle \mapsto^* \langle \mathcal{R}'', \mathcal{M}'', l'', l''' \rangle}{\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \mapsto^* \langle \mathcal{R}'', \mathcal{M}'', l'', l''' \rangle}$$

$$\frac{\langle \mathcal{R}, \mathcal{M}, l_-, \textsc{Instrs}(F(l)) \rangle \rightarrow^* \langle \mathcal{R}', \mathcal{M}' \rangle \qquad \langle \mathcal{R}', \textsc{Branch}(F(l)) \rangle \rightarrow\!| \; v}{\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \mapsto\!| \; \langle \mathcal{M}', v \rangle} \textsc{SBlockReturn}$$

$$\frac{\begin{array}{c}\langle \mathcal{R}_{\textsc{Bad}}[r_1 := v_1, ..., r_n := v_n], \mathcal{M}, F, l_{\neq}, l_{\text{init}} \rangle \mapsto^* \langle \mathcal{R}', \mathcal{M}', l, l' \rangle \\ \langle \mathcal{R}', \mathcal{M}', F, l, l' \rangle \mapsto\!| \; \langle \mathcal{M}'', v \rangle\end{array}}{\langle \mathcal{M}, (v_1, ..., v_n), F \rangle \Mapsto \langle \mathcal{M}'', v \rangle} \textsc{SExe}$$

Fig. 1.3 – Sémantique opérationnelle pour les blocs et les fonctions

de contrôle de la fonction.

$$s_F(l.n) = \begin{cases} \{ l.(n+1) \} & \text{Si } F.l.(n+1) \in \mathsf{LinInstrs} \\ \{ l'.1 \mid l' \in F.l.\omega \} & \text{Si } F.l.n \in \mathsf{BranchInstrs} \end{cases}$$

On dit que deux emplacements τ et τ' se *suivent* dans F si $\tau' \in s_F(\tau)$, et on appelle *chemin* dans F une liste de tels emplacement. La variable I parcourt l'ensemble des chemins ; Soit $\mathsf{Path}_{\tau_1,...,\tau_n}$ le sous-ensemble de Path constitué des chemins passant par $\tau_1, ..., \tau_n$.

Étant donné un élément I de $\mathsf{Path}_{\tau_\bullet}$, un entier n tel que $n \leq |I|$, un état initial des registres $\mathcal{R}$ et un état initial de la mémoire $\mathcal{M}$, on note $\mathcal{R}^F_{I,n^+}$ (resp. $\mathcal{R}^F_{I,n^-}$) l'état des registres après (resp. avant) l'exécution[1] des n premières instructions de I ; on définit similairement $\mathcal{M}^F_{I,n^+}$ et $\mathcal{M}^F_{I,n^-}$. On a bien sûr les égalités $\mathcal{R}^F_{I,1^-} = \mathcal{R}$ et $\mathcal{R}^F_{I,n^+} = \mathcal{R}^F_{I,n+1^-}$ (de même pour la mémoire).

On dit qu'un emplacement τ *domine* (resp. domine strictement) statiquement un emplacement τ' (noté $\tau \prec \tau'$, resp. $\tau \precneqq \tau'$) si, dans tout chemin partant de $\tau_\bullet$, une occurrence de τ' est nécessairement précédée (resp. précédée strictement) par une occurrence de τ :

$$\tau \prec \tau' \text{ (resp. } \tau \precneqq \tau') \equiv \forall I \in \mathsf{Path}_{\tau'}, I.(n') = \tau' \implies \exists n \leq n' \text{ (resp. } n < n') \mid I.(n) = \tau$$

[1] On étend $\rightarrow$ aux instructions de saut, en considérant que l'état de la mémoire et des registres est inchangé après l'exécution d'une instruction de branchement

D'après les conventions sur les fonctions considérées (section 1.1), on est sûr que $\tau_\bullet$ domine tous les emplacements de la fonction.

On généralise la relation de domination aux blocs, en disant qu'un bloc d'étiquette l domine un bloc d'étiquette l' (noté $l \prec l'$) si et seulement si la première instruction du bloc associé à l est toujours exécutée avant la première instruction du bloc associé à l'. On obtient alors l'équivalence

$$l.n \prec l'.n' \Leftrightarrow (l = l' \wedge n \leq n') \vee (l \precneqq l')$$

La relation de domination est clairement un ordre partiel. Elle vérifie également les propriétés suivantes :

Lemme 1 (Plus petit dominateur)

$$(\tau_1 \prec \tau \wedge \tau_2 \prec \tau) \implies (\tau_1 \prec \tau_2 \vee \tau_2 \prec \tau_1)$$

Démonstration. On suppose que $\tau_1 \nprec \tau_2$ et $\tau_2 \nprec \tau_1$. Donc il existe un chemin de $\tau_\bullet$ à τ_1 ne passant pas par τ_2. Par conséquent, tous les chemins allant de τ_1 à τ doivent contenir τ_2 (sinon $\tau_2 \nprec \tau$). Le raisonnement symétrique est également vrai, et tous les chemins de τ_2 à τ contiennent τ_1. En concaténant, on obtient un chemin allant à τ qui est en fait un cycle ne comprenant pas τ : contradiction. □

Lemme 2 (Séquentialité) *Si $\tau \prec \tau' \prec \tau''$, tout chemin partant de $\tau_\bullet$ et passant par τ à τ'' comprend une occurrence de τ' entre toute occurrence de τ et τ''.*

$$\forall \tau, \tau', \tau'', \forall I \in \mathsf{Path}_{\tau_\bullet}, \forall n, n',$$
$$\tau \prec \tau' \prec \tau'' \wedge I.(n) = \tau \wedge I.(n'') = \tau'' \implies \exists n' \,\middle|\, n \leq n' \leq n'' \wedge I.(n') = \tau'$$

Démonstration. Le cas où deux des trois emplacements sont confondus est trivial, on suppose donc que τ, τ' et τ'' sont 2 à 2 distincts. Soit I un chemin partant de $\tau_\bullet$, passant par τ et τ'', et tel que τ' n'apparaisse pas entre deux occurrences de τ et τ'' ; soit I' ce sous-chemin entre τ et τ''. Soit maintenant un chemin I'' partant de $\tau_\bullet$, passant par τ, et ne passant pas par τ' ; ce chemin existe, car sinon $\tau' \prec \tau$ et donc $\tau = \tau'$ par antisymétrie. En concaténant I'' et I' on a un chemin partant de $\tau_\bullet$, passant par τ'' et ne passant pas par τ' : contradiction avec $\tau' \prec \tau''$. □

1.4 Fonctions correctes

On ne considère que des fonctions correctes, au sens suivant :

1. Toute instruction accédant à un registre doit être précédée *statiquement* d'au moins une instruction ayant définie ce registre. Ceci ne requiert toutefois pas qu'une définition domine toutes les lectures du registre[2]. En particulier, on interdit les fonctions telles que celle présentée Fig. 1.4 (qui pourrait potentiellement s'exécuter correctement, en fonction de r_0).

[2] Par exemple un registre peut être défini pour la première fois dans les deux blocs issus d'un *if*.

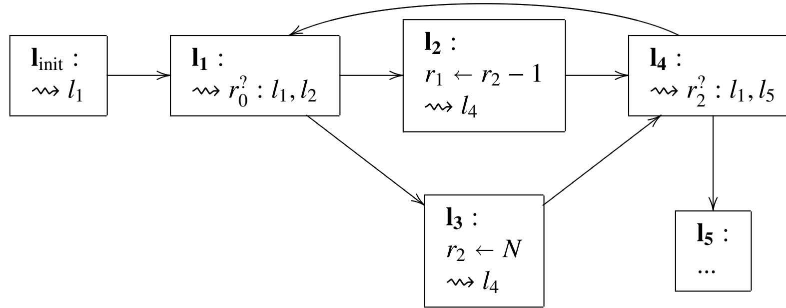


FIG. 1.4 – Fonction violant la propriété de définition des registres

2. Tout accès mémoire doit être correct dans la fonction source.

Avec ces conditions, la seule raison possible de l'apparition d'une valeur BAD est le caractère partiel des opérateurs (division par 0...).

1.5 Raisonnement sur les fonctions

1.5.1 Notion d'équivalence observationnelle

Étant donné une fonction, notre but est de transformer les instructions de cette fonction en une suite d'instructions plus efficace ; une définition exacte de "plus efficace" étant très complexe à expliciter (et dépend de nombreux facteurs, tels que le processeur cible...), on ne la formalisera pas ici. En revanche, on doit assurer que la fonction d'origine et la fonction "optimisée" se comportent de la même façon ; pour cela on définit une notion d'équivalence observationnelle.

L'équivalence choisie est que l'état mémoire et la valeur retournée par les deux fonctions doivent être les mêmes ; on la note $\sim$

$$
\begin{aligned}
F \sim F' \quad \equiv \quad & \forall \mathcal{M}, \forall v_1, ..., v_n, \\
& \langle \mathcal{M}, (v_1, ..., v_n), F \rangle \Mapsto \langle \mathcal{M}', v \rangle \Leftrightarrow \langle \mathcal{M}, (v_1, ..., v_n), F' \rangle \Mapsto \langle \mathcal{M}', v \rangle
\end{aligned}
$$

C'est une forme d'équivalence relativement faible. En particulier, elle ne précise rien sur les fonctions ne terminant pas. Des sémantiques plus fines, basées sur des raisonnements coinductifs (qui permettent de raisonner sur des fonctions ne terminant pas) ont été proposées [Gle04b].

1.5.2 Principes de preuves

Pour prouver un résultat sur la sémantique d'une fonction, il est nécessaire de raisonner simultanément sur les prédicats de la sémantique opérationnelle ($\rightarrow$, $\rightarrow^*$, $\rightsquigarrow$, $\rightarrow\!|$, $\Box\!\!\rightarrow$, $\Box\!\!\rightarrow^*$, $\Box\!\!\rightarrow\!|$, $\Mapsto$).

On considère le cas où l'on souhaite prouver que $\langle \mathcal{M}, (v_1, ..., v_n), F \rangle \Mapsto \langle \mathcal{M}', v \rangle$ implique $\langle \mathcal{M}, (v_1, ..., v_n), F' \rangle \Mapsto \langle \mathcal{M}', v \rangle$. Il suffit pour cela de prouver qu'il existe une relation R sur les registres[3] telle que :

1. $\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \boxminus\!\!\rightarrow^* \langle \mathcal{R}', \mathcal{M}', l', l'' \rangle \wedge \langle \mathcal{R}, \mathcal{R}'' \rangle \in R \implies$
   $\exists \mathcal{R}''' \mid \langle \mathcal{R}'', \mathcal{M}, F', l_-, l \rangle \boxminus\!\!\rightarrow^* \langle \mathcal{R}''', \mathcal{M}', l', l'' \rangle \wedge \langle \mathcal{R}', \mathcal{R}''' \rangle \in R$
2. $\langle \mathcal{R}, \mathcal{M}, F, l_-, l \rangle \boxminus\!\!\rightarrow\!\!| \langle \mathcal{M}', v \rangle \wedge \langle \mathcal{R}, \mathcal{R}'' \rangle \in R \implies \langle \mathcal{R}'', \mathcal{M}, F', l_-, l \rangle \boxminus\!\!\rightarrow\!\!| \langle \mathcal{M}', v \rangle$

Pour cela, on montre généralement que les instructions linéaires de chaque bloc s'exécutent de la même façon par F et F'. Pour les transformations "$1 - 1$" (qui transforment une instruction en une autre instruction), la preuve se fait en utilisant le principe d'induction obtenu directement sur $\rightarrow$ et $\rightarrow^*$. En revanche, pour les autres transformations, il est nécessaire d'utiliser des principes plus forts, complexes à obtenir en Coq.

[3]On peut également faire dépendre cette relation de l_- et de l, ce qui donne un principe de preuve plus puissant.

Chapitre 2

Sémantique de SSA

2.1 Notions de SSA

Comme mentionné dans l'introduction, la forme SSA permet principalement d'obtenir des compilateurs plus efficaces, car plus rapides et parfois "plus optimisants" [App98, Muc97, Mor98]. En forme SSA, chaque registre ne peut apparaître qu'une seule fois à gauche d'une affectation dans toute une fonction. La transformation d'un bloc est relativement aisée, puisqu'il suffit de renommer les registres (Fig 2.1).

Un problème se pose toutefois en présence d'un *if* qui définit un registre r dans chacune de ses branches, lorsque r est utilisée après ce *if*. Pour le résoudre, on introduit une fonction spéciale, notée ϕ, qui mémorise le bloc à partir duquel on arrive, et qui réalise une affectation conditionnelle (Fig 2.1).

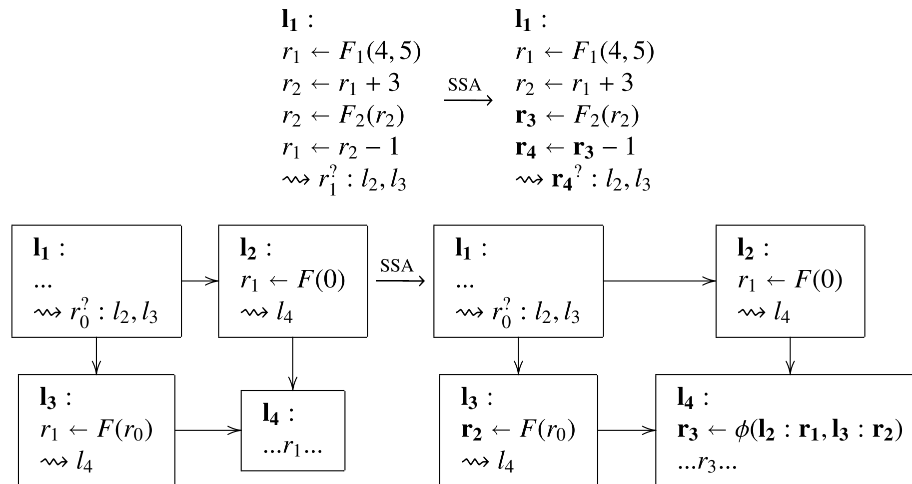


FIG. 2.1 – Exemples de fragments de code convertis en forme SSA

2.2 Fonctions ϕ dans le langage RTL

On présente (Fig. 2.2) la production qui ajoute à la grammaire des instructions linéaires les instructions ϕ ; on introduit également[1] une instruction i_ϕ générique. Enfin, on donne la sémantique des instructions ϕ.

$$i \in \mathsf{LinInstrs} \quad ::= \quad ...$$

$$\mid \quad \begin{pmatrix} r \\ \vdots \\ r \end{pmatrix} \leftarrow \phi \begin{matrix} l & ... & l \\ \begin{pmatrix} \mu & ... & \mu \\ \vdots & \ddots & \vdots \\ \mu & ... & \mu \end{pmatrix} \end{matrix} \qquad (\textsc{Phi})$$

$$i_\phi = \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \leftarrow \phi \begin{matrix} l_1 & ... & l_m \\ \begin{pmatrix} \mu_{1,1} & ... & \mu_{1,m} \\ \vdots & \ddots & \vdots \\ \mu_{n,1} & ... & \mu_{n,m} \end{pmatrix} \end{matrix}$$

$$\frac{\forall p,\ \mathcal{R}(\mu_{p,q}) = v_p}{\langle \mathcal{R}, \mathcal{M}, l_q, i_\phi \rangle \to \langle \mathcal{R}[r_1 := v_1, ..., r_n := v_n], \mathcal{M} \rangle}\ \textsc{SPhi}$$

Fig. 2.2 – Instructions ϕ dans le langage RTL

Les instruction ϕ ont des comportements différents en fonction du bloc du flot de contrôle par lequel on est arrivé à l'instruction. Pour préserver le déterminisme de l'exécution, on impose aux étiquettes apparaissant en entête de la matrice de droite d'être deux à deux distinctes. Elles se comportent en fait comme des instructions de "multi-copie" sur tous les registres présents à gauche de la flèche d'assignation. Cette présentation parallèle explicite diffère de la présentation traditionnelle, dans laquelle plusieurs fonctions ϕ se succèdent, et sont censées s'exécuter simultanément[2] :

$$\begin{gathered} r_1 \leftarrow \phi(\mu_{1,1}, ..., \mu_{1,m}) \\ \vdots \\ r_n \leftarrow \phi(\mu_{n,1}, ..., \mu_{n,m}) \end{gathered}$$

On étend RDef et RUse aux instructions ϕ (en les distinguant toutefois des autres) :

$$\mathsf{RDef}_\phi(i_\phi) = \{r_1, ..., r_n\} \qquad \mathsf{RDef}_\phi(r) = \{\, \iota \mid r \in \mathsf{RDef}_\phi(\iota) \,\}$$
$$\mathsf{RUse}_\phi(i_\phi, l_q) = \{\, r \mid \exists p \mid \mu_{p,q} = r \,\} \qquad \mathsf{RUse}_\phi(i_\phi) = \cup_{1 \le q \le m} \mathsf{RUse}_\phi(i_\phi, l_q)$$
$$\mathsf{RUse}_\phi(r) = \{\, \iota \mid r \in \mathsf{RUse}_\phi(\iota) \,\}$$

[1] Afin de réduire la taille des énoncés, et de faciliter leur lecture.
[2] Hypothèse qui a souvent été oubliée en pratique.

2.3 Propriétés de la forme SSA

Pour être en forme SSA, une fonction doit vérifier les propriétés suivantes :

Propriété 1 *Une instruction ϕ apparaît uniquement en début de bloc. De plus, le bloc initial ne doit pas contenir de ϕ.*

$$\forall l, n, F.l.n = i_\phi \implies (n = 1) \wedge (l \neq l_{\text{init}})$$

Propriété 2 (Unicité de la définition) *Un registre ne peut être défini statiquement qu'une seule fois dans toute la fonction (i.e. la fonction ne comprend qu'une seule instruction de type $r \leftarrow ...$ pour un registre r donné).*

$$\forall r, \forall \tau_1, \tau_2, \{F.\tau_1, F.\tau_2\} \subset \mathsf{RDef}(r) \cup \mathsf{RDef}_\phi(r) \implies \tau_1 = \tau_2$$

Afin de ne pas séparer les paramètres $r_1, ..., r_n$ de la fonction des autres, on introduit un emplacement fictif $l_{\neq}.0$ tel que $l_{\neq}.0 \lneqq \tau_\bullet$ et $\forall i, 1 \leq i \leq n, F.l_{\neq}.0 \in \mathsf{RDef}(r_i)$.

Étant donné que tout registre utilisé doit être défini, pour tout registre r présent dans une fonction, il existe une et une seul définition correspondante, dont l'emplacement est désigné par $\text{Def}(r)$.

Propriété 3 (Domination) *Tout emploi d'un registre dans une instruction qui n'est pas une instruction ϕ doit être dominé strictement par la définition de ce registre.*

Si un registre apparaît dans une instruction ϕ, dans une colonne étiquetée par l, la définition de ce registre doit dominer la dernière instruction du bloc étiqueté par l.

$$F.\tau \in \mathsf{RUse}(r) \implies \text{Def}(r) \lneqq \tau \wedge \forall l, F.\tau \in \mathsf{RUse}_\phi(r, l) \implies \text{Def}(r) \prec l.\omega$$

La propriété de domination interdit l'existence d'instructions du type $r \leftarrow ...r...$ ailleurs que dans une instruction ϕ. Pour les instructions ϕ, on a le lemme suivant :

Lemme 3 *Pour tout indice p d'une ligne de i_ϕ, il existe au moins un argument de cette ligne de i_ϕ qui ne soit pas un registre assigné par i_ϕ :*

$$\forall p, \left\{ r \mid \exists q \mid \mu_{p,q} = r \right\} \not\subset \{r_1, ..., r_n\}$$

Intuitivement, cet indice correspond à un des blocs par lesquels on arrive pour la première fois dans le bloc de l'instruction ϕ.

Démonstration. Par l'absurde, supposons qu'il existe p tel que $\{ r \mid \exists q \mid \mu_{p,q} = r \} \subset \{r_1, ..., r_n\}$. Soit l l'étiquette du bloc dans lequel apparaît i_ϕ. Pour toute étiquette l' d'un bloc prédécesseur de celui étiqueté par l, on sait que $l.1 \prec l'.\omega$, et donc que $l \prec l'$. En particulier, cela veut dire que pour arriver à l, il faut passer par l préalablement : contradiction. □

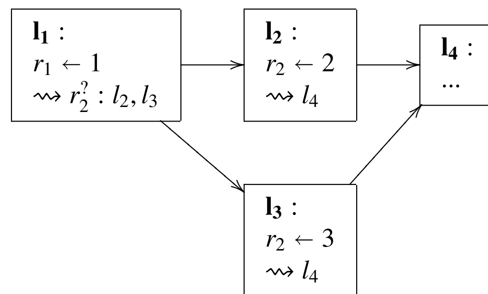
Chapitre 3

Passage en forme SSA

3.1 Idées générales

La transformation d'une fonction RTL en une fonction en forme SSA nécessite un renommage de registres (pour éviter toute réutilisation), et une insertion de fonctions ϕ, afin de fusionner des valeurs de registres venant de deux branches distinctes du code.

Les techniques traditionnelles [CFR+91, BP03] de passage en SSA n'insèrent qu'un nombre de ϕ minimal ; un exemple est présenté Fig. 3.1.



Il n'est pas nécessaire de mettre de fonction ϕ pour le registre r_1 dans l_4, étant donné que c'est toujours la définition de r_1 dans l_1 qui atteint l_4.

FIG. 3.1 – Lien entre structure du flot de contrôle et emplacement des fonctions ϕ

Cette approche a toutefois l'inconvénient de demander des informations complexes à obtenir[1] sur la fonction considérée. Par ailleurs, montrer sa correction ne nous parait pas évident.

Nous adoptons le passage vers une forme SSA *maximale*, où l'on insère au début de chaque bloc une fonction ϕ pour chaque registre de la fonction défini avant ce bloc[2].

[1]En l'occurrence la frontière de domination de chaque emplacement, qui est, pour un τ donné, l'ensemble des emplacements τ' tels que τ domine un prédécesseur de τ', mais ne domine pas strictement τ'.

[2]Cette restriction a son importance. Elle sera développée Section 3.4.1.

Les instructions ϕ inutiles sont ensuite simplifiées (voire supprimées) par un algorithme idoine (Section 4.3). Il a été prouvé [AH00] que, pour des graphes de flot de contrôle réductibles[3], les simplifications menaient à la forme SSA minimale obtenue par les algorithmes traditionnels.

Contrairement à tous les algorithmes présentés jusque là, nous effectuons le renommage en parallèle de l'insertion des fonctions ϕ. Cette optimisation, à première vue uniquement algorithmique, simplifie également la preuve de correction.

3.2 Renommage d'un bloc

On considère un ensemble R de registres. On se donne une fonction Fresh qui génère des noms de nouveaux registres (c'est donc une fonction à effets de bord). Fresh prend un argument un registre, et s'en sert pour renvoyer le nouveau registre ; en particulier, on suppose aussi l'existence d'une fonction Old telle que $\textsc{Old}(\textsc{Fresh}(r)) = r$.

On note f un renommage sur R, et on l'étend aux valeurs en posant $f(v) = v$.

On définit successivement 3 fonctions, pour renommer les registres d'une instruction linéaire (Fig. 3.2), d'une liste d'instructions linéaires (Fig. 3.3), et d'une instruction de branchement (Fig. 3.4).

SSAInstr(i, f)
1 **match** i **with**
2 | () ⇒ **return** ((), f)
3 | $r \leftarrow \mu$ ⇒
4 $r' \leftarrow f(r)$; $f' \leftarrow f[r := \textsc{Fresh}(r)]$; $\mu' \leftarrow f'(\mu)$
5 **return** $(r' \leftarrow \mu',\ f')$
6 | $r \leftarrow [\mu]$ ⇒
7 $r' \leftarrow f(r)$; $f' \leftarrow f[r := \textsc{Fresh}(r)]$; $\mu' \leftarrow f'(\mu)$
8 **return** $(r' \leftarrow [\mu'],\ f')$
9 | $[\mu_m] \leftarrow \mu$ ⇒
10 $\mu'_m \leftarrow f(\mu_m)$; $\mu' \leftarrow f(\mu)$
11 **return** $([\mu'_m] \leftarrow \mu',\ f)$
12 | $r \leftarrow \mu_1 \star \mu_2$ ⇒
13 $r' \leftarrow f(r)$; $f' \leftarrow f[r := \textsc{Fresh}(r)]$
14 $\mu'_1 \leftarrow f'(\mu_1)$; $\mu'_2 \leftarrow f'(\mu_2)$
15 **return** $(r' \leftarrow \mu'_1 \star \mu'_2,\ f')$
16 | $r \leftarrow F(\mu_1, ..., \mu_n)$ ⇒
17 $r' \leftarrow f(r)$; $f' \leftarrow f[r := \textsc{Fresh}(r)]$
18 $\mu'_1 \leftarrow f'(\mu_1)$; ... ; $\mu'_n \leftarrow f'(\mu_n)$
19 **return** $(r' \leftarrow F(\mu'_1, ..., \mu'_n),\ f')$

Fig. 3.2 – Renommage des registres d'une instruction linéaire

[3] C'est à dire des graphes dans lesquels on ne peut entrer dans une boucle que d'une seule façon. C'est le cas dans la plupart des langages structurés, y compris celui utilisé dans Concert.

Lemme 4 *Soit i une instruction, f un renommage des registres. Soient* $(i', f') = \text{SSAI}\textsc{nstr}(i, f)$. *Alors :*

1. $\langle \mathcal{R}, \mathcal{M}, l_-, i\rangle \to \langle \mathcal{R}', \mathcal{M}'\rangle \implies \forall \mathcal{R}'' \mid \mathcal{R}'' \circ f' = \mathcal{R}|_R$,
   $\exists \mathcal{R}''' \mid \mathcal{R}''' \circ f = \mathcal{R}'|_R \wedge \langle \mathcal{R}'', \mathcal{M}, l_-, i'\rangle \to \langle \mathcal{R}''' \mathcal{M}'\rangle$
2. $\langle \mathcal{R}'', \mathcal{M}, l_-, i'\rangle \to \langle \mathcal{R}''', \mathcal{M}'\rangle \implies$
   $\langle \mathcal{R} = \mathcal{R}'' \circ \text{O}\textsc{ld}, \mathcal{M}, l_-, i\rangle \to \langle \mathcal{R}' = \mathcal{R}''' \circ \text{O}\textsc{ld}, \mathcal{M}'\rangle$

On remarque que le renommage est fait “à l'envers” : c'est le *résultat* de l'évaluation de i qui est renommé selon f, pas ses arguments.

Démonstration. Le point 2 est aisé ; en effet, soit i'' l'instruction égale à i', sauf pour les registres r qui sont remplacés par $\text{O}\textsc{ld}(r)$. Par construction de SSAINSTR, on a $i'' = i$, d'où la conclusion.

Pour le point 1, les cas de NOP et STORE sont triviaux, puisque $f = f'$. On suppose alors que $\iota = r_d \leftarrow \mu$, les cas restants se traitant de façon similaire. Soit $r''_d = f'(r_d)$; on a $f' = f[r_d := r''_d]$, $i' = f(r_d) \leftarrow f'(\mu)$, $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r_d := \mathcal{R}(\mu)]$. On considère l'évaluation de $f'(\mu)$ dans $\mathcal{R}''$; si μ est une valeur, $\mathcal{R}''(f'(\mu)) = \mu$. Sinon, soit $r_\mu = \mu$, $\mathcal{R}''(f'(\mu)) = \mathcal{R}''(f'(r_\mu)) = \mathcal{R}(r_\mu)$ par hypothèse. Dans les deux cas $\mathcal{R}''(f'(\mu)) = \mathcal{R}(\mu)$, et $\mathcal{R}''' = \mathcal{R}''[r'_d := \mathcal{R}\mu]$.

Ensuite, si $r = r_d$, alors $f(r_d) = r'_d$ et $\mathcal{R}'''(r'_d) = \mathcal{R}(r_\mu) = \mathcal{R}'(r_d)$. Sinon, si $r \neq r_d$, $f(r) \neq r'_d$ donc $\mathcal{R}'''(f(r)) = \mathcal{R}''(f(r))$. Ensuite $f(r) = f'(r)$ et $\mathcal{R}(r) = \mathcal{R}'(r)$, et $\mathcal{R}'''(f(r)) = \mathcal{R}''(f'(r)) = \mathcal{R}(r) = \mathcal{R}'(r)$. Dans les deux cas, on a l'égalité voulue. □

```
SSAInstrs(L, f)
1  match L with
2    | []  ⇒ return ([], f)
3    | i :: L'  ⇒
4        (L'', f') ← SSAInstrs(L', f)
5        (i', f'') ← SSAInstr(i, f')
6        return ([i'; L''], f'')
```

FIG. 3.3 – Renommage des registres d'une suite d'instructions linéaires

On montre immédiatement (par induction sur la longueur de L), le résultat suivant :

Lemme 5 *Soient L une liste d'instructions linéaires, f un renommage des registres. Soient* $(L', f') = \text{SSAI}\textsc{nstrs}(L', f)$. *Alors :*

1. $\langle \mathcal{R}, \mathcal{M}, l_-, L\rangle \to^* \langle \mathcal{R}', \mathcal{M}'\rangle \implies \forall \mathcal{R}'' \mid \mathcal{R}'' \circ f' = \mathcal{R}|_R$,
   $\exists \mathcal{R}''' \mid \mathcal{R}''' \circ f = \mathcal{R}'|_R \wedge \langle \mathcal{R}'', \mathcal{M}, l_-, L'\rangle \to^* \langle \mathcal{R}''' \mathcal{M}'\rangle$
2. $\langle \mathcal{R}'', \mathcal{M}, l_-, L'\rangle \to^* \langle \mathcal{R}''', \mathcal{M}'\rangle \implies$
   $\langle \mathcal{R} = \mathcal{R}'' \circ \text{O}\textsc{ld}, \mathcal{M}, l_-, L\rangle \to^* \langle \mathcal{R}' = \mathcal{R}''' \circ \text{O}\textsc{ld}, \mathcal{M}'\rangle$

Enfin, le résultat suivant est immédiat :

SSABINSTR(j, f)
1 **match** j **with**
2 $\quad | \rightarrow\!| \; \mu \;\; \Rightarrow$ **return** $\rightarrow\!| \; f(\mu)$
3 $\quad | \rightsquigarrow l \;\; \Rightarrow$ **return** $\rightsquigarrow l$
4 $\quad | \rightsquigarrow \mu^? : l_T, l_F \;\; \Rightarrow$ **return** $\rightsquigarrow f(\mu)^? : l_T, l_F$

FIG. 3.4 – Renommage des registres d'une instruction de branchement

Lemme 6 *Soit j une instruction de branchement, f une fonction de renommage.*
- $\langle \mathcal{R}, j \rangle \rightsquigarrow l \Leftrightarrow \langle \mathcal{R}' \circ f, \text{SSABINSTR}(j, f) \rangle \rightsquigarrow l$
- $\langle \mathcal{R}, j \rangle \rightarrow\!| \; v \Leftrightarrow \langle \mathcal{R}' \circ f, \text{SSABINSTR}(j, f) \rangle \rightarrow\!| \; v$

3.3 Insertion des fonctions ϕ

Une fois un bloc renommé correctement (en se basant sur un renommage des registres que l'on va expliciter), il reste à insérer des instructions ϕ pour passer de l'espace de noms utilisé dans un bloc à un autre.

Pour cela, étant donné une fonction F utilisant un ensemble R de registres, on se donne, pour chaque étiquette $l \in \text{DOM}(f)$:
- Le plus grand sous-ensemble R_l de R tel que chaque registre de $R(l)$ soit défini statiquement dans F avant le bloc étiqueté par l.
- Un renommage f_l des registres de R_l, tel que pour $l \neq l'$, $f_l \cap f_{l'} = \emptyset$ et $f_l \cap R = \emptyset$.

On définit un modèle d'instruction ϕ par :

$$\phi_f(l_1, ..., l_m, , r_1, ..., r_n, r'_1, ..., r'_n) = \begin{pmatrix} r'_1 \\ \vdots \\ r'_n \end{pmatrix} \leftarrow \phi \begin{matrix} l_1 & ... & l_m \\ \end{matrix} \begin{pmatrix} f_{l_1}(r_1) & ... & f_{l_m}(r_1) \\ \vdots & \ddots & \vdots \\ f_{l_1}(r_n) & ... & f_{l_m}(r_n) \end{pmatrix}$$

Théorème 1 (Correction syntaxique du passage en forme SSA)
Étant donné une fonction F, SSA(F) est une fonction en forme SSA.

Démonstration. Les fonctions ϕ sont bien insérées uniquement en début de bloc. De plus, il n'y a pas de ϕ dans le bloc étiqueté par l_{init}.

Pour la propriété d'assignation unique, on observe que les noms de registres utilisés pour une assignation par SSAINSTRS(L, f) sont soit dans le codomaine de f, soit générés par FRESH ; donc si f et f' sont de codomaine disjoints, et si L et L' sont deux listes d'instructions linéaires, alors SSAINSTRS(L, f) et SSAINSTRS(L', f') considérés simultanément sont à assignation unique.

En remarquant de plus que le codomaine de f', lorsque f' est le renommage retourné par un appel à SSAINSTRS, est un ensemble de registres jamais définis auparavant, on en conclut que les assignations ayant lieu dans les instructions ϕ ou au début de l_{init} sont uniques.

SSA(F)
1 $r_1, ..., r_n \leftarrow$ ARGS(F)
2 $f_1, ..., f_m \leftarrow$ FRESHBLOCKSNAMES(F)
3 $F' \leftarrow F$
4 **for each** l **in** DOM(f)
5 **do** $\{r_{d_1}, ..., r_{d_k}\} \leftarrow R_l$
6 $\{l_1, .., l_p\} \leftarrow \{ l' \mid \exists \mathcal{R} \mid \langle \mathcal{R}, \text{BRANCH}(F(l')) \rangle \rightsquigarrow l \}$
7 $(L', f') \leftarrow$ SSAINSTRS(INSTRS($F(l)$), f_l)
8 $j' \leftarrow$ SSABINSTR(BRANCH($F(l)$), f_l)
9 **if** $l \neq l_{\text{init}}$
10 **then** $\phi \leftarrow \phi_f(l_1, ..., l_p, r_{d_1}, ..., r_{d_k}, f'(r_{d_1}), ..., f'(r_{d_k}))$
11 $F'(l) \leftarrow ([\phi; L'], j')$
12 **else** $L'' \leftarrow [f'(r_1) \leftarrow r_1, ..., f'(r_n) \leftarrow r_n]$
13 $F'(l) \leftarrow (L''@L'; j')$
14 **return** F'

FIG. 3.5 – Transformation d'une fonction RTL en une fonction SSA

Soient SSAINSTRS(L, f) = (L', f'). Pour la propriété de domination, on remarque que tous les registres utilisés en argument dans L' sont :
- Soit définis dans une instruction de L' dominant strictement l'instruction les utilisant.
- Soit dans CODOM(f'). Dans ce cas, ils sont définis dans l'instruction ϕ du bloc correspondant à L (sauf pour l_{init} : dans ce cas, ce sont nécessairement les paramètres de la fonction).

La propriété de domination est donc vérifiée pour les registres utilisés dans des instructions non-ϕ.

Pour les autres, on observe que tous les registres appartenant à CODOM(f_l) sont assignés dans le bloc étiqueté par l (au "pire" dans l'instruction ϕ). En conséquence la propriété de domination est vraie également pour les arguments des fonctions ϕ. □

Théorème 2 (Correction sémantique du passage en forme SSA)
Soit F une fonction ; $F \sim$ SSA(F).

Démonstration. Soient $f_1, ..., f_k$ les renommages introduits à la ligne 2 de SSA, $f'_1, ..., f'_k$ ceux obtenus à la ligne 5. On montre par induction structurelle les lemmes suivants, puis on conclut en utilisant le principe présenté section 1.5.2.

1. $\langle \mathcal{R}, \mathcal{M}, l_-, \text{INSTRS}(F(l)) \rangle \rightarrow^* (\mathcal{R}', \mathcal{M}') \implies \forall \mathcal{R}'' \mid \mathcal{R}'' \circ f_{l_-} = \mathcal{R}|_R,$
   $\exists \mathcal{R}''' \mid \mathcal{R}''' \circ f_l = \mathcal{R}'|_R \wedge \langle \mathcal{R}'', \mathcal{M}, l_-, \text{INSTRS}(\text{SSA}(F)(l)) \rangle \rightarrow^* \langle \mathcal{R}''' \mathcal{M}' \rangle$
2. $\langle \mathcal{R}'', \mathcal{M}, l_-, \text{INSTRS}(\text{SSA}(F)(l)) \rangle \rightarrow^* \langle \mathcal{R}''', \mathcal{M}' \rangle \implies$
   $\langle \mathcal{R} = \mathcal{R}'' \circ \text{OLD}, \mathcal{M}, l_-, \text{INSTRS}(F(l)) \rangle \rightarrow^* \langle \mathcal{R}' = \mathcal{R}''' \circ \text{OLD}, \mathcal{M}' \rangle$

Comme d'habitude, le point 2 est facile : en appliquant OLD aux fonctions ϕ_f, on obtient uniquement des lignes de la forme $r \leftarrow \phi(r, ..., r)$; la conclusion est alors directe en utilisant les lemmes précédents, et la transitivité de $\rightarrow^*$.

Le point 1 est à peine plus compliqué. Soit un $\mathcal{R}''$ vérifiant l'hypothèse. Par construction, $\langle \mathcal{R}'', \mathcal{M}, l_{-}, \mathrm{SSA}(F).l.1\rangle \to \langle \mathcal{R}'''', \mathcal{M}\rangle$, avec $\mathcal{R}'''' \circ f'_l = \mathcal{R}|_R$. On conclut directement en utilisant le lemme 5. □

3.4 Commentaires

Cette approche pour le passage en forme SSA introduit de nombreuses instructions ϕ qui sont superflues. Nous la choisissons néanmoins, pour 3 raisons :

- Après simplification, le nombre de fonctions ϕ est équivalent à celui obtenu par la méthode basée sur la frontière de domination [AH00].
- Les performances sont globalement bonnes [AH00].
- La preuve formelle est beaucoup plus facile.

De plus, il nous semble important de séparer la simplification des ϕ de l'algorithme de passage en forme SSA, la première étape pouvant être réutilisées ensuite.

Un autre point est important : notre approche par renommage inversé permet de connaître par avance les paramètres des instructions ϕ, et d'effectuer toute la conversion en SSA en une seule passe sur le code. Cette optimisation avait déjà été mentionnée comme étant envisageable, sans qu'une implémentation ait été proposée [AH00]. Étant donné que toutes les autres approches nécessitent deux passes (insertion des ϕ, *puis* renommage), il est probable que les performances de notre algorithme soient meilleures que celles précédemment reportées.

3.4.1 Restriction sur la définition des registres

La restriction syntaxique que nous avons introduite sur la définition des registres dans la section 1.4 prend tout son sens ici. En effet, elle permet d'assurer la correction de notre forme SSA (tout registre utilisé vérifie bien la propriété de domination), tout en autorisant à ne pas initialiser tous les registres dans le bloc initial. Cette approche, qui est celle proposée par les algorithmes de passage en SSA qui traitent la question[4], est insatisfaisante pour le programmeur (de nombreux programmes parfaitement licites ne la respectent pas), et peu exploitable dans le cadre de la preuve formelle. En effet, le compilateur peut difficilement trouver des valeurs par défaut acceptables pour les registres de la fonction qui ne sont pas initialisés dans l_{init} : ce faisant, il risquerait en effet de créer une fonction "plus définie" que la fonction d'origine, résultat que l'on souhaite éviter.

Insistons sur le fait que cette restriction est très peu contraignante : lire un registre alors qu'il peut ne pas avoir été initialisé est un très mauvais style de programmation. À ce titre, la figure 1.4 est un bon exemple, complexe à interpréter : il n'est pas immédiat de savoir sous quelles conditions l'évaluation de r_2 a une chance de réussir.

[4]Les autres la passant sous silence.

Chapitre 4

Transformations sur la forme SSA

Dans cette section, on s'intéresse aux transformations/optimisations valides en forme SSA. Les optimisations suivantes sont considérées :

- Propagation de constantes
- Propagations de copies
- Élimination de code mort
- Élimination des sous-expressions communes
- Séparation des registres

4.1 Substitution textuelle

En forme SSA on peut remplacer, sous certaines conditions, un registre utilisé dans une instruction par sa définition.

Soit F une fonction, $r \Subset F$ tel que $F.\mathrm{Def}(r) = r \leftarrow \mu$. Soit τ tel que $F.\tau \in \mathsf{RUse}(r)$. Soit $F' = F[\tau := (F.\tau)[r := \mu]]$.

Lemme 7 (Correction syntaxique) *F' est forme SSA.*

Démonstration. La propriété d'unicité de la définition est triviale : nécessairement $\tau \neq \mathrm{Def}(r)$ (par propriété de domination) et donc aucun membre gauche d'une instruction n'est changé.

Pour la propriété de domination, le résultat est trivial si μ. est une constante. Sinon, soit $r' = \mu$. On suppose dans un premier temps que $F.\tau$ n'est pas une instruction ϕ ; il suffit alors de montrer que $\mathrm{Def}(r') \lneqq \tau$. On a par hypothèse $\mathrm{Def}(r') \lneqq \mathrm{Def}(r)$ et $\mathrm{Def}(r) \lneqq \tau$, et on conclut par transitivité de $\prec$. Si $F.\tau$ est une instruction ϕ, soit l tel que r apparaisse dans la colonne étiquetée par l de l'instruction (l n'est pas nécessairement unique). Il faut montrer que $\mathrm{Def}(r') \prec l.\omega$. Or $\mathrm{Def}(r') \lneqq \mathrm{Def}(r)$ et $\mathrm{Def}(r) \prec l.\omega$, d'où encore une fois conclusion par transitivité. □

Lemme 8 (Correction sémantique) *F et F' sont équivalentes : $F \sim F'$*

Démonstration. On montre simultanément que F et F' se simulent exactement l'une l'autre. Soit I un chemin d'exécution partant de $\mathcal{R}$ et $\mathcal{M}$. Soit k tel que $I.n = \tau$.

Si μ est une constante, alors l'évaluation de $F.\tau$ et $F'.\tau$ est trivialement identique. Si μ n'est pas une constante, soit $r' = \mu$. On sait que $\mathrm{Def}(r') \lneqq \mathrm{Def}(r)$. Soit n_r et $n_{r'}$ les plus grands entiers tels que $n_r < k$, $n_{r'} < n$, $I.(n_r) = \mathrm{Def}(r)$ et $I.(n_{r'}) = \mathrm{Def}(r')$. Nécessairement $n_{r'} < n_r$ (sinon contradiction avec le lemme 2). Pour tous les indices n entre $n_{r'}$ et $k-1$, $\mathcal{R}^+_{I,n}(r')$ est constant (et égal pour F et F') ; de même pour $\mathcal{R}^+_{I,n}(r)$ entre n_r et k. En particulier, $\mathcal{R}^-_{I,r}(r') = \mathcal{R}^-_{I,k}(r')$, et donc $F.I.(k)$ et $F'.I.(k)$ s'évaluent de la même façon. □

4.2 Propagation de constantes et de copies

La propagation de constantes est une transformation qui remplace l'utilisation d'un registre par la constante qui lui correspond le cas échéant. Ceci permet par suite d'autres optimisations, telles que les optimisations sur les opérateurs. Par exemple, en notant << l'opérateur de décalage logique (qui est généralement exécuté plus rapidement par le processeur qu'une addition) :

$$\begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow 2 \\ r_3 \leftarrow r_1 * r_2 \end{array} \xrightarrow{\text{Propagation de constantes}} \begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow 2 \\ r_3 \leftarrow r_1 * 2 \end{array} \xrightarrow{\text{Substitution d'opérateur}} \begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow 2 \\ r_3 \leftarrow r_1 << 1 \end{array}$$

La propagation de copies est une transformation qui, étant donné une instruction $r_1 \leftarrow r_2$, remplace, tant que les valeurs de r_1 et r_2 n'ont pas changé, les occurrences de r_1 par r_2. Une telle transformation peut avoir plusieurs avantages :

1. Une utilisation moindre de registres si la copie peut être complètement éliminée (par suppression de code mort), ce qui peut faciliter le travail de l'allocateur de registres[1]
2. Possibilité de simplifier des expressions ensuite, par exemple par substitution d'opérateur ou élimination des sous-expressions communes.

$$\begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow r_1 \\ r_3 \leftarrow r_1 + r_2 \end{array} \xrightarrow{\text{Propagation de copies}} \begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow r_1 \\ r_3 \leftarrow r_1 + r_1 \end{array} \xrightarrow{\text{Substitution d'opérateur}} \begin{array}{l} r_1 \leftarrow F(...) \\ r_2 \leftarrow r_1 \\ r_3 \leftarrow r_1 << 1 \end{array}$$

Dans un fonction RTL standard, la propagation de constantes ou de copies nécessite de détecter que la valeur du registre constant (pour la propagation de constantes) et de r_1 ou r_2 (pour la propagation de copies) change ; cette opération est conceptuellement simple, mais plus difficile à mettre en œuvre puisqu'elle nécessite une analyse de data-flow.

En forme SSA, la valeur d'un registre assigné par copie ou à une constante vaut en tout point de son domaine d'existence la valeur du registre d'origine (pour une copie), ou de la constante (Lemme 8). En conséquence, les deux formes de propagation

[1]Tout particulièrement lorsque celui-ci ne prend pas en compte de telles copies.

se ramènent à une simple répétition d'opérations de substitutions textuelles. Un algorithme général pour ce type de transformation sera présenté section 4.5.

4.3 Simplification d'instructions ϕ

On montre qu'il est possible de séparer, sous certaines conditions, une instruction ϕ en une autre instruction ϕ plus "petite" et une instruction de copie. Cette optimisation a été remarquée lors de l'étude du passage en forme SSA "maximale" [AH00].

Définition 1 (Séparation de ϕ) *Étant donné une fonction F, soit q tel qu'on ait $F.l_q.1 = i_\phi$. Soit un indice de ligne p ; on suppose que tous les arguments apparaissant à droite de l'assignation sur la ligne étiquetée par p font partie de l'ensemble $\{ r_p, \mu_p \}$, où μ_p est une valeur quelconque. Soit alors $i_{\phi-p}$ l'instruction correspondant à i_ϕ sauf pour la ligne p que l'on supprime ; on peut remplacer i_ϕ par la suite d'instruction $[i_{\phi-p}; r_p \leftarrow \mu_p]$.*

Lemme 9 *Si μ_p est un registre r'_p, alors $\mathrm{Def}(r'_p) \precneqq l_q.1$ (et donc $r'_p \notin \mathsf{RDef}_\phi(i_\phi)$).*

Démonstration. Soit $l_{q'}$ l'étiquette d'un bloc dominant strictement le bloc étiqueté par l_q (qui existe forcément). Nécessairement l'argument de i_ϕ sur la ligne p et la colonne $l_{q'}$ est r'_p : sinon ce serait r_p, et on aurait $l_q.1 \prec l_{q'}.\omega$, ce qui serait contradictoire avec $l_{q'} \precneqq l_q$. On a donc $\mathrm{Def}(r'_p) \prec l_{q'}.\omega \precneqq l_q.1$, d'où conclusion par transitivité. □

Théorème 3 (Correction syntaxique) *Une fonction en forme SSA sur laquelle on effectue une séparation de ϕ reste en forme SSA.*

Démonstration. La transformation préserve l'emplacement des instructions ϕ, qui restent en tête de bloc. Par ailleurs on est toujours en assignation unique, le seul changement intervenant pour r_p qui n'est plus assigné dans $i_{\phi-p}$ mais dans l'instruction suivante.

La propriété de domination est moins évidente. On commence par montrer que la définition de r_p domine toujours ses utilisations.

Si r_p était utilisé dans une instruction autre que i_ϕ, la nouvelle définition domine toujours l'utilisation, puisqu'elle a été insérée juste après i_ϕ. Sinon, soit $l_{q'}$ l'étiquette d'une colonne dans laquelle r_n apparaît dans $i_{\phi-p}$. Par hypothèse on $\mathrm{Def}(r_n) = l_q.1 \prec l_{q'}.\omega$ (dans la fonction d'origine) ; on doit montrer $\mathrm{Def}(r_n) = l_q.2 \prec l_{q'}.\omega$ (dans la fonction modifiée), ce qui est trivial.

On montre finalement que si μ_p est un registre (par exemple r'_p), alors la définition de r'_p domine strictement $l_q.2$. D'après le lemme 9, $\mathrm{Def}(r'_p) \prec l_q.1$, et donc $l_q.2$ par transitivité. □

Lemme 10 *Les suites d'instructions $[i_\phi]$ et $[i_{\phi-p}; r_p \leftarrow \mu]$ sont équivalentes :*

$$\forall p,\ \exists \mu_p \,|\, \{ r_p, \mu_{p,1}, ..., \mu_{p,m} \} = \{ r_p, \mu_p \} \implies \\ \langle \mathcal{R}, \mathcal{M}, l_-, [i_\phi] \rangle \to^* \langle \mathcal{R}', \mathcal{M}' \rangle \Leftrightarrow \langle \mathcal{R}, \mathcal{M}, l_-, [i_{\phi-p}; r_p \leftarrow \mu] \rangle \to^* \langle \mathcal{R}', \mathcal{M}' \rangle$$

Démonstration. On montre le résultat par cas sur le bloc d'arrivée. On considère tout d'abord le cas des colonnes où l'argument de l'instruction ϕ était μ_p. Les instructions ϕ réalisant des copies, le résultat est quasiment direct. Le seul cas problématique est lorsque μ_p est modifié dans $i_{\phi-p}$; on a vu plus haut que c'est impossible (lemme 9). Pour les autres cas, on n'aurait théoriquement besoin d'aucune instruction (puisque l'instruction ϕ ne fait que recopier le contenu de r_n dans lui-même). Mais par hypothèse de domination on avait $\mathcal{R}(r_n) = \mathcal{R}(\mu_p)$, donc la nouvelle instruction de copie ne fait en fait rien ; en particulier, il est intéressant en terme de performances de l'éliminer[2] par propagation de copie, puis suppression de code mort. □

Le résultat suivant est une simple conséquence :

Théorème 4 (Correction sémantique) *Pour toute fonction F dans laquelle on effectue une séparation de ϕ donnant une fonction F', F et F' sont sémantiquement équivalentes.*

4.4 Élimination de code mort

En forme SSA, une forme simple d'élimination de code mort devient triviale. En effet, considérons une instruction[3] $r \leftarrow r'$, $r \leftarrow \mu_1 \star \mu_2$, $r \leftarrow [\mu]$ ou $r \leftarrow \phi(...)$ (y compris si r n'est pas seul dans l'instruction ϕ). Si r n'apparaît nulle part ailleurs dans la fonction, il est clair que l'instruction est superflue, et peut être remplacée par ().

Bien évidemment, cette propriété est également vraie pour les fonctions qui ne sont pas en forme SSA. Toutefois, elle est rarement vérifiée car les variables sont réutilisées.

Définition 2 (Suppression d'assignation morte) *Soit F une fonction, τ un emplacement tel que $F.\tau \in \mathsf{RDef}(r) \cup \mathsf{RDef}_\phi(r) \wedge F.\tau \notin \textsc{Call} \wedge \forall \tau', F.\tau' \notin \mathsf{RUse}(r) \cup \mathsf{RUse}_\phi(r)$.*

Si $F.\tau \in \mathsf{RDef}(r)$, soit $i = ()$; sinon, soit i l'instruction ϕ correspondant à $F.\tau$ sauf pour la ligne correspondant à r que l'on supprime (si il ne reste plus aucune ligne, on considère que i est également ()).

Soit $F' = F[\tau := i]$.

Théorème 5 *F' est en forme SSA, et $F \sim F'$.*

Démonstration. La bonne formation est évidente. La seule propriété importante est la propriété de domination, qui est triviale étant donné que r n'est jamais utilisé dans F.

Pour la correction sémantique, on montre par induction structurelle que F et F' se simulent exactement, sauf pour la valeur de $\mathcal{R}(r)$. □

La suppression d'assignation morte n'est toutefois pas une optimisation à faire systématiquement ; en effet, elle peut contrarier d'autres optimisations telles que l'élimination des sous-expressions communes, en réduisant l'ensemble des expressions disponibles dans le programme. La meilleure approche consiste en fait à ne l'exécuter qu'avant de sortir de la forme SSA.

[2] L'article d'origine [AH00] propose d'ailleurs une substitution textuelle de r'_p par r_p.

[3] On ne considère pas $r \leftarrow F(...)$ à cause des cas où F fait un effet de bord.

Encore une fois, le résultat présenté ici ne supprime qu'une seule variable inutilisée. Un algorithme supprimant l'intégralité des variables mortes peut être écrit en utilisant une liste de tâches, qu'on présente en section 4.5.

Notons qu'un autre type d'élimination de code mort[4] est aisé à mettre en œuvre, non grâce à la forme SSA, mais grâce à notre langage RTL. Soit en effet $l \in \textsc{Dom}(F)$ telle que l ne soit destination d'aucune instruction de branchement de F. Il est clair que le bloc étiqueté par l ne sera jamais exécuté, et que F et $F|_{\textsc{Dom}(F)-l}$ sont équivalentes.

Dans un langage RTL non structuré (où toutes les instructions se suivent, et où les sauts conditionnels mènent soit à l'instruction suivante, soit à une étiquette prédéfinie), ce type de transformation est coûteux et complexe à prouver.

4.5 Algorithme général par liste de tâches

4.5.1 Algorithme général

On suppose l'existence d'un prédicat P sur $\mathsf{Functions} \times \mathsf{Loc}$, d'une fonction de transformation T prenant en argument une fonction F et un emplacement $\tau \mid P(F, \tau)$, et renvoyant une nouvelle fonction, et d'une fonction de changement C prenant en argument une fonction F, un emplacement $\tau \mid P(F, \tau)$, et renvoyant un ensemble d'emplacements de F. L'algorithme générique[5] est présenté Fig. 4.1.

```
WorkList(F)
1  F' ← F
2  W ← τ(F)
3  while W ≠ ∅
4  do {τ} ∪ W' ← W
5      if P(F', τ)
6         then F' ← T(F, τ)
7              W ← W' ∪ C(F, τ)
8         else W ← W'
9  return F'
```

Fig. 4.1 – Algorithme générique par liste de tâches

On demande l'hypothèse suivante sur P et T :

Définition 3 (Correction sequentielle) *Soit F une fonction, τ un emplacement tel que $P(F, \tau)$. Soit $F' = T(F, \tau)$. Alors $F \sim F'$.*

Le résultat suivant est alors immédiat par transitivité de $\sim$.

[4] L'algorithme proposé pour la suppression d'assignations mortes effectue la suppression de code *inutile* : le code est exécuté, mais son résulta n'est pas utilisé. Dans ce paragraphe, nous traitons la suppression de code *jamais exécuté* ; la terminologie sur ce point est malheureusement ambiguë.

[5] Inspiré par Appel [App98].

Théorème 6 (Correction) *Soit F une fonction quelconque. Si* WorkList(F) *termine,* $F \sim$ WorkList(F).

Pour la terminaison, on se donne une relation M bien fondée, qui décroît strictement lorsque on applique T ; la preuve se fait alors par un ordre lexicographique sur $(M, |W|)$. En effet, soit M décroît strictement (si P est vérifié), soit la taille de W décroît strictement ; étant donné que M est supposée bien fondée, le produit lexicographique est lui-même bien fondé.

4.5.2 Instanciation

On précise dans cette partie les optimisations qui peuvent être traitées par l'algorithme par liste de tâches.

Propagation de constantes ou de registres

On utilise les résultats de la section 4.2. $P(F, \tau)$ est la propriété "$F.\tau = r \leftarrow \mu \wedge \exists \tau' \mid r \Subset F.\tau'$". Si $F.\tau = r \leftarrow \mu$, $T(F, \tau) = F[r := \mu]$ et $C(F, \tau) = \{\tau' \mid r \Subset F.\tau'\}$.

La correction sémantique découle de la transitivité de $\sim$, et du Lemme 8.

Comme mesure, on définit les registres "définis mais pas utilisés" comme étant l'ensemble des registres $r$de F tels que :

- $\exists \tau \mid F.\tau \in \mathsf{RDef}(r) \cup \mathsf{RDef}_\phi(r)$
- $\forall \tau, F.\tau' \notin \mathsf{RUse}(r) \cup \mathsf{RUse}_\phi(r)$

Cet ensemble est borné par l'ensemble des registres présents dans la fonction. En conséquence, la taille de cet ensemble est une mesure bien fondée, et elle augmente bien après chaque propagation.

Suppression de code mort

On utilise les résultats de la section 4.4. $P(F, \tau)$ est la propriété "$F.\tau \in \mathsf{RDef}(r) \cup \mathsf{RDef}_\phi(r) \wedge F.\tau \notin \textsc{Call} \wedge \forall \tau', F.\tau' \notin \mathsf{RUse}(r) \cup \mathsf{RUse}_\phi(r)$". $T(F, \tau) = F[\tau := ()]$ (si $F.\tau$ n'est pas une instruction ϕ ; sinon c'est l'instruction ϕ privée de la ligne correspondant à r). $C(F, \tau) = \{\tau' \mid r' \in F.\tau', r' \Subset F.\tau\}$ (après chaque suppression d'expression, on reconsidère l'utilité de tous les registres apparaissant dans l'expression).

La correction séquentielle a été prouvé dans le théorème 5.

Comme mesure bien fondée, on prend simplement la taille de l'ensemble des registres présents dans la fonction (bornée par 0).

Simplification des instructions ϕ

On réutilise les résultats de la section 4.3. $P(F, \tau)$ est la propriété "$F.\tau$ est une instruction ϕ, dont la ligne p est de la forme $r \leftarrow \phi(\mu_1, ..., \mu_n)$ avec $\{r, \mu_1, ..., \mu_n\} = \{r, \mu\}$". Soit de nouveau $i_{\phi-p}$ l'instruction correspondant à p, sauf pour la $p$ième ligne. Alors $T(F, \tau) = F[\tau := i_{\phi-p}][r := \mu]$, et $C(F, \tau) = \{\tau\} \cup \{\tau' \mid \mu \Subset F.\tau'\}$ (si μ est un registre). Plutôt que d'insérer une copie inutile, on substitue directement la valeur extraite de l'instruction ϕ, et on élimine l'assignation.

La correction séquentielle découle des résultats sur la séparation des instructions ϕ (Théorème 4), la propagation de registres, et la suppression de code mort.

Comme mesure bien fondée, on prend la taille de l'ensemble des registres définis par une instruction ϕ dans F.

Dans le cas précis de cet algorithme, il est bien évidemment plus intéressant de n'initialiser la liste de tâches qu'avec les emplacements des fonctions ϕ.

Autres algorithmes

D'autres optimisations peuvent être traitées par cet algorithme, par exemple la simplification d'opérateurs donc certains des arguments sont constants, ou l'évaluation de sauts conditionnels (on remplace par exemple $\rightsquigarrow 2^? : l_1, l_2$ par l_1). Étant donné qu'ils ne sont pas spécifiques à la forme SSA, on ne les présente pas ici.

"Mélange" des algorithmes

Il est bien sûr plus intéressant d'exécuter toutes ces optimisations en parallèle. Le produit lexicographique des trois mesures utilisées est toujours une mesure bien fondée, et permet de garantir la terminaison de l'algorithme général.

4.6 Élimination des sous-expressions communes

L'élimination des sous-expressions communes a pour but de supprimer des calculs redondants :

$$\begin{array}{l} r_2 \leftarrow r_0 + r_1 \\ r_3 \leftarrow r_0 \\ r_3 \leftarrow r_3 + r_1 \end{array} \xrightarrow{\text{Élimination des sous-expressions communes}} \begin{array}{l} r_2 \leftarrow r_0 + r_1 \\ r_3 \leftarrow r_0 \\ r_3 \leftarrow r_2 \end{array}$$

Nous utilisons une technique connue sous le nom de value numbering, où chaque expression calculée est enregistrée (sous un nouveau nom, d'où le "value"). On veut éviter deux types de calculs inutiles : ceux utilisant les opérateurs $\star$, et les accès mémoire. Notre analyse sera néanmoins relativement peu efficace pour les accès mémoire, puisque nous considérons que tout Store ou Call invalide l'intégralité de nos connaissances sur la mémoire.

On note C un contexte, c'est à dire une fonction prenant un argument de type $\mu \star \mu$ ou $[\mu]$ et renvoyant un registre. On note $C[[_] :=]$ un tel contexte, dans lequel tous les renseignements sur la mémoire ont été effacés. On considère que C renvoie $\perp$ partout où il n'est pas défini.

On note f un renommage sur les variables, et f^* son itéré (c'est à dire la fonction telle que $f^*(r) = f(...(f(r)))$, avec la condition $f(f^*(r)) = f^*(r)$. Pour que cette définition ait un sens, on demande que le renommage ne comprenne pas de cycle. Ce sera le cas dans une fonction en forme SSA, où les registres sont ordonnés par l'ordre de domination sur l'emplacement de leur définition. On étend comme d'habitude le renommage aux instructions.

On définit trois fonctions, travaillant respectivement sur une instruction linéaire (Fig. 4.2), une liste de telles instructions, (Fig. 4.3), et une fonction entière (Fig. 4.4).

CSEINSTR(i, C, f)
1 **match** i **with**
2 $\quad | \ r \leftarrow \mu_1 \star \mu_2 \quad \Rightarrow$
3 $\qquad e \leftarrow f^*(\mu_1 \star \mu_2)$
4 $\qquad$ **return if** $C(e) = r'$
5 $\qquad\qquad$ **then** $(r \leftarrow r', C, f[r' := r])$
6 $\qquad\qquad$ **else** $(i, C[e := r], f)$
7 $\quad | \ r \leftarrow [\mu] \quad \Rightarrow$
8 $\qquad e \leftarrow f^*([\mu])$
9 $\qquad$ **return if** $C(e) = r'$
10 $\qquad\qquad$ **then** $(r \leftarrow r', C, f[r' := r])$
11 $\qquad\qquad$ **else** $(i, C[e := r], f)$
12 $\quad | \ [\mu_m] \leftarrow [\mu] \ | \ r \leftarrow F(...) \quad \Rightarrow$
13 $\qquad$ **return** $(i, C[[_] :=], f)$
14 $\quad | \ r \leftarrow r' \quad \Rightarrow$
15 $\qquad$ **return** $(i, C, f[r' := r])$
16 $\quad | \ () \ | \ i_\phi \quad \Rightarrow$
17 $\qquad$ **return** (i, C, f)

FIG. 4.2 – Élimination des sous-expressions communes dans une instruction linéaire

---

Soit i une instruction, C et f un contexte et un renommage. Soit un éventuel registre r tel que $i \in \mathsf{RDef}(r)$; alors on demande que r n'apparaisse pas déjà dans C ou f. Si cette condition est vérifiée, et que C est valide, (c'est à dire que ses valeurs sont bien contenues à l'exécution dans les registres correspondants), l'éventuelle substitution effectuée par CSEINSTR est valide[6] (au sens que la nouvelle instruction et l'ancienne s'évaluent de la même façon), et le nouveau contexte et le nouveau renommage sont valides. La condition de non apparition de r est importante pour éviter tout cycle.

CSEINSTRS(L, C, f)
1 **match** L **with**
2 $\quad | \ [] \quad \Rightarrow$ **return** $([], \ C, f)$
3 $\quad | \ i :: L' \quad \Rightarrow$
4 $\qquad (i', C', f') \leftarrow$ CSEINSTR(i, C, f)
5 $\qquad (L'', C'', f'') \leftarrow$ SSAINSTRS(L', C', f')
6 $\qquad$ **return** $([i'; L''], C'', f'')$

FIG. 4.3 – Élimination des sous-expressions communes dans une liste d'instructions linéaires

[6] Les preuves de cette partie n'ont pu être terminées, aussi nous ne donnerons pas de résultats formels.

Soit maintenant une liste d'instructions linéaires L en assignation unique, un contexte C et un renommage f ; on demande à nouveau que tous les registres assignés dans L ailleurs que dans une instruction ϕ ne soient pas déjà présents dans C ou f.

Alors le résultat de CSEInstrs(L, C, f) est correcte : on obtient une liste d'instructions sémantiquement équivalente à L, et un contexte et un renommage corrects. Cette propriété se montre par induction sur la longueur de L, en utilisant la correction de CSEInstr.

CSE(F)
1 **for each** l **in** Dom$(F) \cup \{l_{\neq}\}$
2 **do** $D_l \leftarrow ((_ \mapsto \bot), \mathrm{Id})$
3 $F \leftarrow F'$
4 **for each** l **in** Dom(F)
5 **do** $(C, f) \leftarrow D_{\mathrm{PredDom}(l)}$
6 $(L', C', f') \leftarrow$ CSEInstrs(Instrs$(F(l)), C, f)$
7 $F' \leftarrow F'[l := (L', \mathrm{Branch}(F(l)))]$
8 $D_l \leftarrow (C', f')$
9 **return** F'

Fig. 4.4 – Élimination des sous-expressions communes dans une fonction

---

On note PredDom(l) le bloc dominant immédiatement le bloc étiqueté par l (on considère que PredDom$(l_{\mathrm{init}}) = l_{\neq}$; on a prouvé l'unicité de PredDom (Lemme 1), et son existence est assurée par le fait que $l_{\neq}$ domine toutes les autres étiquettes).

Étant donné une fonction F, la fonction CSE itère sur les étiquettes du domaine de F en optimisant un bloc, puis en enregistrant les valeurs disponibles à la fin de ce bloc (dans un tableau D, créé à cet effet). Tout au long de l'exécution, on a les invariants suivants :

– D_l contient un sous-ensemble des expressions disponibles à la sortie du bloc l
– F' est équivalente F

La correction de l'algorithme est assurée par la correction de CSEInstrs, par la propriété de domination des fonctions en forme SSA (qui assure que les expressions sont bien disponibles dans les blocs successeurs), et par la propriété d'assignation unique.

4.6.1 Ordre de parcours

Par définition, tous les ordres de parcours des étiquettes sont corrects, puisque les contextes sont toujours corrects. En revanche, certains sont moins intéressants que d'autres : ce sont ceux qui calculent les sous-expressions d'un bloc avant d'avoir calculées celles de son plus petit dominateur. Le meilleur ordre de parcours est en fait un parcours préfixe de l'arbre de domination des blocs.

Notre approche est toutefois intéressante dans le cadre de la preuve formelle ; en effet, elle évite de dépendre d'un ordre de parcours (qui peut être difficile à obtenir si les fonctions de création de l'arbre de domination n'ont pas été écrites).

4.6.2 Remarques

L'élimination des sous-expressions communes doit être effectuée avant la suppression de code mort, qui risque de supprimer certains calculs pouvant être réutilisés. En revanche elle ne nécessite pas de pré-traitement par propagation de copies ou de registres, puisque ces opérations sont simulées en interne.

Dans les algorithmes d'élimination des sous-expressions communes en forme non-SSA, on ne peut pas maintenir de renommage comme nous l'avons fait, étant donné que les variables ne sont pas ordonnées. Il faut alors utiliser des structures de données complexes, qui permettent des congruences entre expressions ; plus important (et plus difficile), on doit pouvoir "oublier" ces congruences lorsqu'un registre est réaffecté. Ces difficultés algorithmiques disparaissent en forme SSA.

4.7 Discussion

4.7.1 Séparation de variables

La séparation de variables a pour but d'éviter, autant que possible, la réutilisation de variables dans un programme. L'objectif est de faciliter le travail de l'allocateur de registres : il est plus intéressant d'avoir deux variables avec des durées de vie ne se chevauchant pas, plutôt qu'une seule variable avec une grande durée de vie (qui risque de mobiliser un registre). La séparation de variables est bien évidemment moins efficace que SSA (en termes de nombres de variables séparées), puisqu'elle n'insère pas de fonctions ϕ.

Afin de réaliser la séparation des variables, le compilateur doit calculer les reaching definitions, par une (coûteuse) analyse de data-flow ; en comparaison, en forme SSA, la séparation est faite automatiquement, et n'entraîne donc aucun surcoût.

4.7.2 Optimisations conditionnelles

La propagation de constantes que nous avons présentée ici pourrait être améliorée. Considérons le code C suivant, qui est donné en forme SSA simplifiée Fig. 4.5. Bien que la valeur de x soit toujours égale à 6, et que la fonction retourne toujours[7] 6, les optimisations effectuées ne permettent pas de le détecter.

```
x = 6;
while(y) { if (x=6)  y = y-1; else  x = x+1; }
return x;
```

Ceci est dû au fait que nos algorithmes ne tentent pas de supposer que r_{x_4} vaut toujours 6 ; implicitement, ils supposent que le bloc 3 va être exécuté, ce qui entraîne l'apparition de r_{x_4} et l'impossibilité de simplifier. Des algorithmes capables de faire cette hypothèse et de l'exploiter existent [WZ91].

[7] En arithmétique modulo, y sera à un certain moment égal à 0.

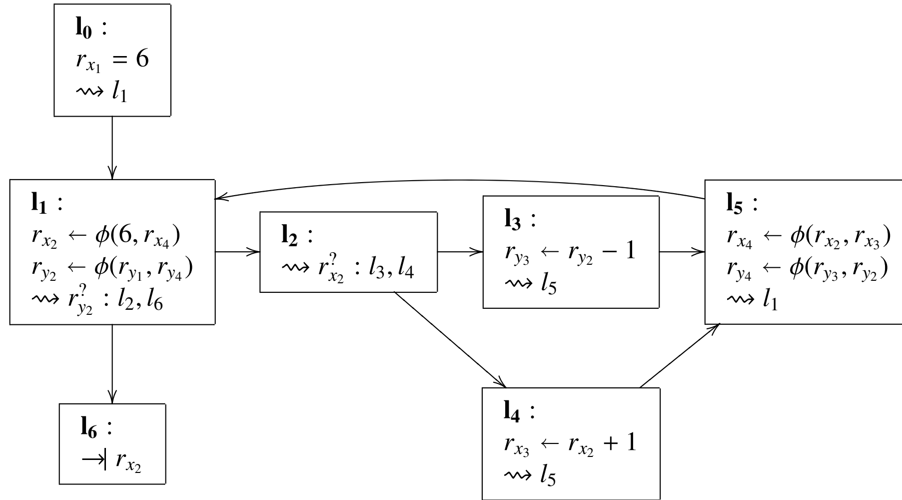


FIG. 4.5 – Propagation conditionnelles de registres

De fait, il existe 4 familles de propagation de constantes : conditionnel ou non, en forme SSA ou non. Généralement, les algorithmes en forme SSA correspondent, aux optimisations algorithmiques près, à ceux en forme non-SSA, avec des cas supplémentaires pour traiter les instructions ϕ.

Nous pensons que ce raisonnement se généralise à de nombreuses optimisations[8]. Prouver la correction de ces algorithmes devient alors plus simple, puisqu'il suffit de "réutiliser" les preuves sur les algorithmes standard.

Il existe également des algorithmes d'élimination des sous-expressions communes beaucoup plus puissants, qui sont capables de faire des hypothèses d'égalité similaires [CCK+97, Cli95].

[8] Toutes celles qui respectent les propriétés de la forme SSA, notamment l'hypothèse de domination.

Chapitre 5

Sortie de la forme SSA

5.1 Algorithme

Étant donné notre sémantique, il est conceptuellement très simple de sortir de la forme SSA : il suffit de remplacer les instructions ϕ par des instructions de copies. Deux problèmes potentiels se posent toutefois :

1. Conserver le parallélisme implicite des instructions ϕ
2. Trouver un endroit où insérer les copies

Pour résoudre le point 1, on suppose l'existence d'une fonction MultiCopie telle que MultiCopie$((r_1, ..., r_n), (\mu_1, ..., \mu_n))$ renvoie une suite d'instructions linéaires copiant *simultanément* les valeurs μ_m dans r_m ; une telle fonction utilisant au maximum un registre temporaire a déjà été prouvé dans l'action Concert.

Pour le point 2, il est intéressant d'introduire la notion *d'arêtes critiques* dans le graphe de flot de contrôle. Une arête est dite critique lorsqu'elle part d'un bloc ayant plusieurs successeurs pour aller à un bloc ayant plusieurs prédécesseurs ; c'est le cas de toutes les arêtes de "retour" d'une boucle, ou de l'arête 2 de la Fig. 5.1.

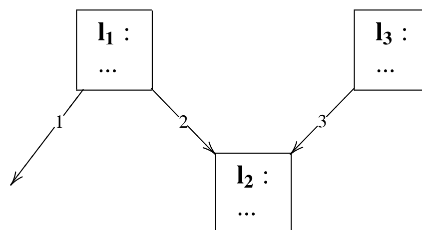


Fig. 5.1 – Un graphe contenant une arête critique

Les arêtes critiquent empêchent de placer correctement les instructions de copie. Prenons l'exemple de la Fig. 5.1. Supposons que le bloc l_2 commence par une instruc-

tion ϕ ; si on place les instructions de copie correspondant au passage de l_1 à l_2 à la fin de l_1, elles seront exécutées même lorsqu'on suit l'arête 1 pour quitter l_1. Par ailleurs, si on les place au début de l_2, elles seront exécutées même lorsqu'on arrive de l_2.

Pour résoudre ce problème, il est habituel de demander la séparation des arêtes critiques (en créant un bloc vide "sur" l'arête). Cette transformation peut être nécessaire également à des transformations avancées, telles que la factorisation des invariants de boucle.

Ici nous choisissons de créer un nouveau bloc pour chaque arête concernée par une instruction ϕ ; cela simplifie l'invariant utilisé pour prouver la fonction. Les blocs placés sur des arêtes non critiques disparaîtront de toute façon lors de la linéarisation du code. On suppose l'existence de nouveaux noms générés à partir des étiquettes existantes de $l : \forall l, l' \in \text{Dom}(f), l_{l,l'} \notin \text{Dom}(f)$. On utilise $\bullet$ pour concaténer des colonnes dans une matrice.

LeaveSSA(F)
1 $F' \leftarrow F$
2 **for each** l **in** $\text{Dom}(f)$
3 **do if** $\exists V_d, l_1, ..., l_m, V_1, ..., V_m \mid F.l.1 = V_d \leftarrow (l_1 : V_1 \bullet \ldots \bullet l_m : V_m)$
4 **then for each** p **in** $\{1, ..., m\}$
5 **do** $B \leftarrow (\text{Instrs}(F.l_p), \text{Branch}(F.l_p)[l := l_{l_p,l}])$
6 $B' \leftarrow (\text{MultiCopie}(V_d, V_p), \rightsquigarrow l)$
7 $F' \leftarrow F'[l_p := B][l_{l_p,l} := B']$
8 $B'' \leftarrow (\text{Instrs}(F.l)[1 := ()], \text{Branch}(F.l))$
9 $F' \leftarrow F'[l := B'']$
10 **return** F'

Fig. 5.2 – Sortie de la forme SSA

5.2 Remarques

La sortie de la forme SSA est une opération qui a longtemps été faite incorrectement [CFR+91, App98], à cause de la non prise en cause des deux difficultés mentionnées plus haut. En particulier, en écrivant une instruction ϕ par registre, il est facile d'oublier que celles-ci doivent s'exécuter en parallèle. C'est la raison pour laquelle nous avons choisi l'emploi d'une unique instruction multi-registres.

Des algorithmes de sortie de la phase SSA corrects ont été proposé [BCHS98, SJGS99, RdG04] ; tous sont par ailleurs relativement complexes. Certains tentent de minimiser le nombre de registres en sortie ; il reste à voir si ce travail est vraiment nécessaire avec un bon allocateur de registres.

Chapitre 6

Discussion

6.1 Travaux connexes

Les essais de formalisation de SSA sont rares. Le travail le plus proche est la formalisation d'une sémantique utilisant une machine à états abstraits [Gle04a] ; un générateur de code simple partant de la forme SSA est également prouvé. L'article n'aborde pas la transformation d'un langage non-SSA en un langage SSA, ni de possibles optimisations.

Dans un domaine connexe, une machine abstraite pour interpréter un langage en forme SSA a été proposé [vRWF04]. L'exécution nécessairement parallèle des instructions ϕ est effectuée en sauvegardant l'état des registres avant de commencer leur évaluation, et en indiquant la fin des instructions ϕ avec une instruction explicite.

6.2 Formalisation en Coq

Les preuves du compilateur développé dans l'action Concert sont faites dans l'assistant de preuve Coq [Coq04]. Traduire le langage RTL, ainsi que la sémantique, en Coq ne pose pas de problèmes. En revanche les preuves d'équivalence sémantique entre deux fonctions sont fastidieuses et complexes dès que les instructions des deux fonctions ne sont pas en relation 1-1. Cela risque de poser de nombreux problèmes, notamment lors de la preuve formelle de l'insertion des fonctions ϕ.

Une autre difficulté, plus insidieuse, est que la notion de domination n'a pas de rapport direct avec la sémantique. En conséquence, il est très difficile de l'utiliser comme hypothèse dans les preuves formelles ; elle est malheureusement cruciale pour les preuves telles que celle du lemme 8.

Pour résoudre ce problème, deux solutions sont envisageables :

- Trouver une propriété équivalente de domination, plus “compatible” avec la sémantique. Ce n'est a priori pas évident.
- Utiliser des preuves mettant en jeu des équivalences de chemins d'exécution, comme pour le lemme 8. L'inconvénient de cette approche est que ce type de

raisonnement n'est pas un principe inductif, et n'est pas généré automatiquement par Coq.

6.3 Conclusion

La sémantique SSA que nous avons proposé permet de prouver les optimisations les plus courantes faites en forme SSA. Comme prévu, ces optimisations sont très nettement algorithmiquement plus efficaces que celles faites sur les fonctions RTL classiques. Nous avons présenté un algorithme pour transformer une fonction RTL en une fonction en forme SSA pour lequel la classe des fonctions acceptée est extrêmement large : contrairement aux approches habituelles, nous ne demandons *pas* que les variables de la fonction soient initialisées dans le bloc initial. Par ailleurs, notre algorithme pour quitter la forme SSA est non seulement correct, mais également beaucoup plus simple que ceux habituellement présentés dans la littérature.

Ce travail demande a être validé par une implémentation complète en Coq. Par ailleurs, il est possible d'envisager la preuve de correction de nombreux autres algorithmes : algorithmes "standard" pour passer en (et sortir de) la forme SSA, algorithmes d'optimisations plus performants.

Bibliographie

[AH00] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *Proceedings of the 9th Internation Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–124. Springer Verlag, March 2000.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8) :859–881, 1998.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3) :375–425, 2003.

[CCK+97] Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 32, pages 273–286. ACM Press, 1997.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, October 1991.

[Cli95] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 246–257. ACM Press, 1995.

[Coq04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[Gle04a] Sabine Glesner. An ASM semantics for SSA intermediate representations. In *Proceedings of the 11th International Workshop on Abstract State Machines*. Springer Verlag, Lecture Notes in Computer Science, Mai 2004.

[Gle04b] Sabine Glesner. A proof calculus for natural semantics based on greatest fixed point semantics. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification), 7th European Conferences on Theory and Practice of Software (ETAPS 2004)*. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS), April 2004.

[Mor98] Robert Morgan. *Building an optimizing compiler*. Digital Press, 1998.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[RdG04] Fabrice Rastello, Francois de Ferrière, and Christophe Guillon. Optimizing translation out of ssa using renaming constraints. In *International Symposium on Code Generation and Optimization (CGO-04)*. IEEE Computer Society Press, March 2004.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210. Springer-Verlag, 1999.

[vRWF04] J. von Ronne, N. Wang, and M. Franz. Interpreting programs in static single assignment form. In *ACM SIGPLAN 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME'04)*, Washington, D.C., June 2004. (to be published).

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2) :181–210, 1991.