Rapport de stage du Master de recherche en informatique MPRI

*C*ertification d'un validateur de transformation

Jean-Baptiste Tristan

Stage dirigé par *Xavier Leroy* et effectué à l'INRIA

Résumé

Nous détaillons l'utilisation de la technique dite *validation de transformation* dans le cadre d'un compilateur optimisant certifié. La transformation étudiée est l'ordonnancement d'instructions par blocs pour un langage proche de l'assembleur. L'algorithme de validation est implémenté et certifié correct en utilisant l'assistant de preuve Coq.

Remerciements

Mes remerciements s'adressent, en premier lieu, à Xavier Leroy qui m'a proposé mon stage et accepte d'encadrer ma thèse.
Alain Frisch et Boris Yakobowski m'ont beaucoup aidé à mieux présenter mon travail et je leur en suis très reconnaissant.
Yann Régis-Gianas, Benoît Razet et Zaynah Dargaye qui ont passé du temps à corriger les innombrables fautes qui parsemaient les premières versions de ce rapport.

Table des matières

Chapitre 1

Introduction

Les exemples de catastrophes dues à des dysfonctionnements logiciels ne manquent pas : en 1996, l'explosion de la fusée Arianne à coûté un milliard de dollars à l'agence spatiale européenne ; en 1994, un avion de la Royal Air Force s'écrase tuant 29 personnes ; 5 patients sont mort suite à une erreur dans le logiciel de la machine à radiothérapie Therac 25... De même que les logiciels sont de plus en plus présents, en particulier dans les systèmes embarqués, les défaillances peuvent être de plus en plus coûteuses. Les méthodes formelles de validation de logiciels font donc depuis plusieurs années l'objet d'une attention toute particulière. Toutefois, cette attention se porte généralement sur le programme source, avant la transformation en un programme exécutable sur une machine. Or cette transformation (compilation) est un processus complexe, surtout si on souhaite un programme efficace, et des bogues peuvent être introduits par le compilateur dans le programme. Il est donc important de s'assurer que les propriétés établies sur un programme avant la compilation le sont toujours après : c'est le but de la *certification formelle de compilateurs*.

Un compilateur certifié est un compilateur accompagné d'une preuve formelle que la sémantique d'un programme est préservée tout au long de la compilation. Un tel compilateur est en cours de développement au sein du projet `Compcert` [2]. La technique utilisée est la suivante : les différentes passes du compilateur sont écrites dans l'assistant de preuve `Coq` [3, 5] puis prouvées correctes. Plus précisément, pour une passe T, on montre que $\forall p \forall t_p, t_p = T\ p \rightarrow [\![p]\!] = [\![t_p]\!]$, c'est à dire que les programmes p et t_p sont observationnellement équivalents si t_p est obtenu par transformation T de p. Ce compilateur est en cours de développement et produit de l'assembleur modérément optimisé pour processeur `PowerPC`, un des processeur les plus utilisés dans les systèmes embarqués. Deux langages peuvent actuellement être compilés : un sous-ensemble de C et un sous ensemble de langage fonctionnel.

Néanmoins, la complexité des optimisations est limitée par la complexité des preuves et une autre approche de la certification semble prometteuse : la validation de transformation. Selon cette approche, une passe du compilateur peut être développée dans un langage quelconque ; on programme alors en `Coq` un *validateur* qui analyse le programme initial et celui issu de la transformation et valide la transformation comme ayant préservée la sémantique ou la rejette. Ce validateur doit être prouvé correct : en comparaison avec la formule précédente, cela donne : $\forall p \forall t_p, valid(p, t_p) = true \rightarrow [\![p]\!] = [\![t_p]\!]$. On espère selon cette approche que, pour un certain type de transformation, *la validation soit plus simple à certifier que la transformation elle même*. Ceci est un des postulats que ce stage tente d'éclaircir.

La validation de transformation est un concept récent apparu dans les travaux de Pnueli & *al* [14] et qui s'applique dans leur travail à un compilateur de langage synchrone. Ce travail définit notamment le *cadre de validation* qui détaille les ingrédients nécessaires à une validation :

- La sémantique du langage source ainsi que celle du langage cible doivent être définies.
- La propriété d'équivalence entre un programme source et un programme cible doit être précisée.
- Il faut un algorithme pour décider de l'équivalence des deux programmes.

Le concept a ensuite été appliqué à des compilateurs pour langages usuels, formalisé et généralisé [11, 12, 8, 16, 17, 13].

Le cadre sémantique est déjà posé au sein du projet `Compcert`. Les langages intermédiaires du compilateur sont formellement spécifiés, syntaxe et sémantique, et la propriété à prouver est la suivante : la sémantique d'un programme doit être préservée par une transformation [7]. Il reste alors deux problèmes à résoudre. D'une part, il faut concevoir des algorithmes de validation pour les nombreuses optimisations que l'on pourrait vouloir implémenter [15, 9, 4] : malgré les nombreuses contributions dans la littérature sur la validation, beaucoup d'optimisations n'ont pas encore été étudiées sous cet angle et le problème reste largement ouvert. D'autre part, la certification de tels validateurs est un sujet qui n'a pas encore été abordé, ce qui pose la question de la *faisabilité* de ce processus et aussi de son *ingénierie* : *un validateur devant être certifié ne s'écrit pas comme un validateur usuel*.

Ce travail répond à quelques interrogations. D'une part, il montre que l'approche est faisable pour une certaine classe d'optimisations, l'ordonnancement d'instructions (*list scheduling* [15]), et qu'effectivement l'approche est tout à fait abordable, ce qui va dans le sens de notre postulat. D'autre part, il détaille ce processus de certification dans l'espoir de dégager des méthodes et des idées réutilisables lors de prochaines réalisations et révèle en particulier quelques détails importants tels que le choix du langage intermédiaire ou les différences entre établir une préservation faible de la sémantique (les deux programmes, s'ils s'exécutent sans erreurs, ont la même sémantique) et une préservation forte (les deux programmes ont la même sémantique).

L'exposé se déroule de la façon suivante : dans le chapitre 2 j'introduis le plus précisément possible le cadre sémantique par la présentation du langage d'étude, le langage *Mach*, les propriétés que l'on souhaite établir et la transformation de programmes *Mach* étudiée. Dans le chapitre 3, j'expose un algorithme de validation basé sur le concept d' *évaluation symbolique* et sa preuve de correction. Enfin, dans le chapitre 4, j'aborde la question des autres optimisations de la famille de l'ordonnancement et expose un algorithme original pour valider l'algorithme dit de *trace scheduling*.

Chapitre 2

Le langage Mach, définition et transformation de programmes

Notre langage d'étude est le langage Mach. Il s'agit d'un des langages intermédiaires utilisé dans le processus de compilation du compilateur certifié. Mach est un langage de bas niveau, proche de l'assembleur. L'intérêt de ce langage est triple. Premièrement, on peut y opérer des transformations de bas niveau efficacement telles que celles envisagées par la suite. Deuxièmement, le concept de bloc, central dans le processus de certification, y apparaît sans trop de difficultés. Troisièmement, il est un peu plus concis que le langage d'assemblage.

Dans la suite de ce chapitre, nous précisons les points un et deux du cadre de validation c'est-à-dire la sémantique des programmes ainsi que le critère d'équivalence envisagé. Pour cela, il faut commencer par décrire la syntaxe. Cette présentation se veut la plus formelle possible. Malheureusement, le langage Mach étant utilisé dans un compilateur réel et de bas niveau, une présentation complète de sa syntaxe et de sa sémantique serait d'une part beaucoup trop importante et d'autre part sans intérêt pour notre étude. Par conséquent, les objets syntaxiques et sémantiques les plus primitifs sont décrits en langage informel. Cependant, mon développement traite le langage tout entier.

2.1 Syntaxe

Syntaxe d'un programme Mach

Un programme Mach se compose de plusieurs fonctions ayant une signature et un code. Le code d'une fonction est une suite d'instructions. Cette syntaxe est grandement simplifiée pour permettre une compréhension aisée de la sémantique.

- r, r_1, r_n représentent les registres du processeur.
- *ofs* représente en entier indiquant une position dans un bloc d'activation (frame).
- *op* représente une opération arithmétique du processeur.
- *addr* représente un mode d'adressage du processeur.
- *faddr* représente l'adresse d'une fonction.
- *lbl* représente une étiquette dans le code (label).

Instructions : $i ::= \texttt{Mgetstack}\ ofs\ r$
$\mid \texttt{Msetstack}\ r\ ofs$
$\mid \texttt{Mgetparam}\ ofs\ r$
$\mid \texttt{Mop}\ op\ (r_1, ..., r_n)\ r$
$\mid \texttt{Mload}\ faddr\ (r_1, ..., r_n)\ r$
$\mid \texttt{Mstore}\ faddr\ (r_1, ..., r_n)\ r$

$$
\begin{array}{l}
\mid \texttt{Mcall}\ faddr \\
\mid \texttt{Mlabel}\ lbl \\
\mid \texttt{Mgoto}\ lbl \\
\mid \texttt{Mcond}\ r\ lbl \\
\mid \texttt{Mreturn}
\end{array}
$$

Code : $c ::= nil \mid i :: c$

Function : $f := (sig; c)$

Program : $p := nil \mid f :: p$

2.2 Sémantique

La sémantique du langage est donnée dans un style opérationnel dit "mixed-step" [7], elle consiste en l'interprétation inductive des règles qui vont suivre. Toutes les instructions, à l'exception de l'appel de fonction qui est décrit dans un style à grands pas, sont décrites dans un style à petit pas. La sémantique d'une suite d'instruction est la clôture réflexive et transitive de la sémantique d'instruction individuelle. La sémantique d'une fonction est l'exécution du corps la de fonction dans un état avec un bloc d'activation[1] contenant les spills de registres[2]. La sémantique d'un programme est la valeur de retour de sa fonction principale, passé par convention dans le registre r_3 du processeur.

Notations

L'état de la machine est noté $[c, rs, fr, m]$ où c est le code restant à exécuter, rs est une fonction qui associe à chaque registre sa valeur, fr est le bloc d'activation courant de la fonction et m l'état mémoire. On note $rs\ x$ la valeur du registre x et $rs\#x \leftarrow v$ le placement de la valeur v dans le registre x.

- *ge* est l'environnement global, qui contient par exemple les variables globales.
- *f* est la fonction en cours d'exécution.
- *sp* est le pointeur de pile qui pointe vers le bloc d'activation courant.
- *parent* est un bloc d'activation contenant certains des arguments de la fonction, les autres étant dans des registres.

Définition 1. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \longrightarrow [c', rs', fr', m']}$

$$(ge, f, sp, parent) \vdash [\texttt{Mlabel}\ lbl :: c, rs, fr, m] \longrightarrow [c, rs, fr, m] \quad \text{(Mlabel)}$$

L'instruction `Mlabel` déclare le point de programme *lbl*, elle n'a pas d'effet sur la machine.

$$\frac{get\_slot\ fr\ ofs = v \qquad rs' = rs\#dst \leftarrow v}{(ge, f, sp, parent) \vdash [\texttt{Mgetstack}\ ofs\ dst :: c, rs, fr, m] \longrightarrow [c, rs', fr, m]} \quad \text{(Mgetstack)}$$

Si la frame *fr* à l'offset *ofs* contient la valeur v, alors l'instruction `Mgetstack` place cette valeur dans le registre *dst*.

$$\frac{set\_slot\ fr\ ofs(\ rs\ src) = fr'}{(ge, f, sp, parent) \vdash [\texttt{Msetstack}\ src\ ofs :: c, rs, fr, m] \longrightarrow [c, rs, fr', m]} \quad \text{(Msetstack)}$$

Soit *fr'* la frame *fr* à laquelle on a stocké à l'offset *ofs* la valeur contenue dans le registre *src*. L'instruction `Msetstack` se réduit vers l'état où la frame est *fr'*.

[1] Le bloc d'activation est une zone de la mémoire utilisée par une fonction pour y stocker ses *informations personnelles*. On l'appellera par son nom anglais *frame*.

[2] Les spills de registres sont des zones du bloc d'activation utilisés pour stocker les valeurs qui n'ont pas pu être placées dans les registres à cause du manque de place.

$$\frac{get\_slot\ parent\ ofs = v \qquad rs' = rs\#dst \leftarrow v}{(ge, f, sp, parent) \vdash [\texttt{Mgetparam}\ ofs\ dst :: c, rs, fr, m] \longrightarrow [c, rs', fr, m]}\ \text{(Mgetparam)}$$

Si la frame *parent* à l'offset *ofs* contient la valeur v, alors l'instruction `Mgetparam` place cette valeur dans le registre dst.

$$\frac{\begin{array}{c}eval\_operation\ ge\ sp\ op(rs\ r_1, ..., rs\ r_n) = v \\ rs' = rs\#res \leftarrow v\end{array}}{(ge, f, sp, parent) \vdash [\texttt{Mop}\ op\ (r_1, ..., r_n)\ res :: c, rs, fr, m] \longrightarrow [c, rs', fr, m]}\ \text{(Mop)}$$

L'instruction `Mop` applique l'opérande op aux valeurs contenues dans les registres $(r_1, ..., r_n)$. La valeur v résultant de cette évaluation est placée dans le registre res.

$$\frac{\begin{array}{c}eval\_addressing\ addr\ (r_1, ..., r_n) = a \\ Mem.loadv\ m\ a = v \qquad rs' = rs\#dst \leftarrow v\end{array}}{(ge, f, sp, parent) \vdash [\texttt{Mload}\ addr\ (r_1, ..., r_n)\ dst :: c, rs, fr, m] \longrightarrow [c, rs', fr, m]}\ \text{(Mload)}$$

L'instruction `Mload` place dans le registre dst la valeur provenant de l'adresse calculée à partir des registres $(r_1, ..., r_n)$ et du mode d'adressage *addr* de la mémoire m.

$$\frac{\begin{array}{c}eval\_addressing\ addr\ (r_1, ..., r_n) = a \\ Mem.storev\ m\ a(\ rs\ src) = m'\end{array}}{(ge, f, sp, parent) \vdash [\texttt{Mstore}\ addr\ (r_1, ..., r_n)\ src :: c, rs, fr, m] \longrightarrow [c, rs, fr, m']}\ \text{(Mstore)}$$

L'instruction `Mstore` place le contenu du regsitre src dans la mémoire m à l'adresse calculée à partir des registres $(r_1, ..., r_n)$ et du mode d'adressage *addr*.

$$\frac{find\_function\ faddr = f' \qquad (ge, f', parent) \vdash [rs, m] \xrightarrow{\lambda} [rs', m']}{(ge, f, sp, parent) \vdash [\texttt{Mcall}\ faddr :: c, rs, fr, m] \longrightarrow [c, rs', fr, m']}\ \text{(Mcall)}$$

L'instruction `Mcall` exécute la fonction à l'adresse *faddr*, ici *f'*, l'exécute, et reprend son exécution dans l'état laissé par *f'*.

$$\frac{find\_label\ lbl\ f.(\ fn\_code) = c'}{(ge, f, sp, parent) \vdash [\texttt{Mgoto}\ lbl :: c, rs, fr, m] \longrightarrow [c', rs, fr, m]}\ \text{(Mgoto)}$$

L'instruction `Mgoto` remplace le code restant à exécuter c par le code trouvé au label lbl du code de la fonction.

$$\frac{\begin{array}{c}eval\_condition\ (r_s\ r) = true \\ find\_label\ lbl\ f.(\ fn\_code) = c'\end{array}}{(ge, f, sp, parent) \vdash [\texttt{Mcond}\ r :: c, rs, fr, m] \longrightarrow [c', rs, fr, m]}\ \text{(Mcondtrue)}$$

Si la condition est vérifiée dans le registre r ; alors l'instruction `Mcond` remplace le code restant à exécuter c par le code c' trouvé à l'adresse lbl.

$$\frac{eval\_condition\ r = false}{(ge, f, sp, parent) \vdash [\texttt{Mcond}\ r :: c, rs, fr, m] \longrightarrow [c, rs, fr, m]}\ \text{(Mcondfalse)}$$

Si la condition n'est pas vérifiée dans le registre r, alors l'instruction `Mcond` passe à l'instruction suivante.

Définition 2. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*} [c', rs', fr', m']}$

$$\frac{}{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*} [c, rs, fr, m]} \text{ (refl)}$$

$$\frac{(ge, f, sp, parent) \vdash [c, rs, fr, m] \longrightarrow [c', rs', fr', m']}{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*} [c', rs', fr', m']} \text{ (one)}$$

$$\frac{(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{*} [c_2, rs_2, fr_2, m_2] \qquad (ge, f, sp, parent) \vdash [c_2, rs_2, fr_2, m_2] \xrightarrow{*} [c_3, rs_3, fr_3, m_3]}{(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{*} [c_3, rs_3, fr_3, m_3]} \text{ (tra}$$

L'exécution d'une suite d'instructions est décrite par la clôture réflexive et transitive de l'exécution d'instruction. On appelle la règle (one) le "pas de base" de l'exécution.

Définition 3. $\boxed{ge, f, parent \vdash [c, rs, fr, m] \xrightarrow{\lambda} [c', rs', fr', m']}$

$$\frac{c = function\_code\ f \qquad (fr, sp) = allocate\_new\_frame \qquad (ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*} [\texttt{Mreturn} :: c', rs', fr', m']}{(ge, f, parent) \vdash [rs, m] \xrightarrow{\lambda} [rs', m']} \text{ (funct)}$$

Soit c le corps de la fonction, fr une frame fraîchement allouée et sp le pointeur en résultant. L'exécution d'une fonction est l'exécution de c dans cet état jusqu'à une instruction de retour.

Définition 4. $\boxed{p \rightsquigarrow r}$

$$\frac{ge = new\_env\ p \qquad main(p) = f \qquad (ge, f, empty\_frame) \vdash [rs, m] \xrightarrow{\lambda} [rs', m'] \qquad rs'\ (r_3) = v}{p \rightsquigarrow v} \text{ (program)}$$

L'exécution d'un programme correspond à l'exécution de la fonction principale *main(p)* dans l'environnement *ge* où on a alloué les variables globales.

2.3 Ce que l'on prouve

Soit $\mathcal{V}$ un validateur qui prend deux programmes p et t_p en entrée et renvoie *true* si la sémantique est préservée et *false* sinon.

La propriété que l'on vérifie sur $\mathcal{V}$ dans la suite de l'exposé, et qu'on appelle *préservation faible*, est la suivante :

$$\forall p\, \forall t_p\, \forall v\, \forall v', \mathcal{V}(p, t_p) = true \wedge p \rightsquigarrow v \wedge t_p \rightsquigarrow v' \Longrightarrow v = v'$$

Bien que cette propriété soit intéressante, supposer que le programme transformé termine sans erreur a ses inconvénients. Une transformation peut produire une instruction erronée et rendre la sémantique indéfinie. Afin d'éviter ce désagrément il conviendrait de prouver la propriété suivante sur $\mathcal{V}$, que l'on appelle *préservation forte* :

$$\forall p\, \forall t_p\, \forall v, \mathcal{V}(p, t_p) = true \wedge p \rightsquigarrow v \Longrightarrow t_p \rightsquigarrow v$$

Il semble tout à fait possible qu'une analyse supplémentaire $\mathcal{V}'$ sur p et t_p implique la propriété de bonne définition de la sémantique du programme transformé. Cette analyse, en complément de l'analyse précédente, permettrait de prouver la préservation sémantique forte.

Notons que ces formules ne signifient que les valeurs de retour à l'exécution seront identiques que si la sémantique est déterministe, ce qui est le cas pour le langage MACH.

Le cadre sémantique ayant été posé, il s'agit maintenant de définir la transformation à étudier. Il s'agit d'une transformation de type ordonnancement d'instructions. Cette transformation est une optimisation de bas niveau. Elle consiste à changer l'ordre des instructions dans le code pour mieux tirer parti du pipeline du processeur et des multiples unités de calcul. Evidemment, on souhaite que cette modification ne change pas la sémantique du programme.

2.4 Ordonnancement d'instructions Mach

Comme on vient de le voire, notre méthode de certification ne prouve rien sur la transformation directement : la transformation n'apparaît pas dans la propriété à prouver. Il n'est donc pas nécessaire de comprendre tous les rouages du *list scheduling* pour comprendre sa vérification, néanmoins, il faut avoir une idée des transformations qui ont eu lieu pour comprendre pourquoi la vérification fonctionne et ses limitations.

Dans la présentation de l'algorithme on suppose que la machine a comme ressources un banc de registres et la mémoire. Pour plus de clarté nous n'utiliserons pas dans les exemples des instructions Mach mais des pseudo instructions qui lisent les ressources apparaissant à gauche de la flèche $\Rightarrow$ et écrivent les ressources apparaissant à droite de cette flèche.

L'algorithme d'ordonnancement choisi est le *list scheduling*. L'élément important à noter est que la transformation s'applique aux blocs de base d'instructions (c'est-à-dire les blocs qui ne contiennent pas de branchements). Cette particularité a son importance lors du processus de certification. L'algorithme s'effectue en quatre étapes principales :

1. calcul du graphe de flot de contrôle
2. calcul des graphes de dépendances
3. assignation de priorités aux instructions
4. simulation de l'exécution du programme et production d'un ordonnancement

Calcul du graphe de flot de contrôle

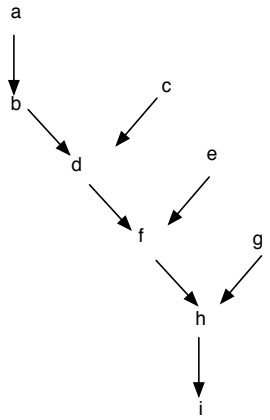
La transformation de list scheduling s'effectue au niveau du bloc. On commence donc par calculer le graphe de flot de contrôle associé afin de pouvoir effectuer la transformation bloc par bloc. La suite de l'algorithme est donc à appliquer à chaque bloc indépendamment.

Calcul du graphe des dépendances

Le graphe de dépendance est calculé à partir de la suite d'instructions à ordonnancer et dénote toutes les dépendances à respecter entre les instructions pour que la sémantique soit préservée. Il existe trois types de dépendances :

- Lecture après écriture (LAE) : une écriture dans une ressource suivie d'une lecture de cette même ressource ne peuvent être permutées.
- Ecriture après écriture (EAE) : une écriture dans une ressource suivie d'une écriture dans cette même ressource ne peuvent être permutées.
- Ecriture après lecture (EAL) : la lecture d'une ressource suivie d'une écriture de cette même ressource ne peuvent être permutées.

L'exemple qui suit expose le principe de cette construction.

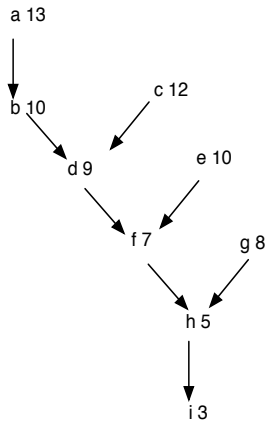


a : $load$ $r_0, 0 \Rightarrow r_1$
b : add $r_1, r_1 \Rightarrow r_1$
c : $load$ $r_0, 8 \Rightarrow r_2$
d : $mult$ $r_1, r_2 \Rightarrow r_1$
e : $load$ $r_0, 16 \Rightarrow r_2$
f : $mult$ $r_1, r_2 \Rightarrow r_1$
g : $load$ $r_0, 24 \Rightarrow r_2$
h : $mult$ $r_1, r_2 \Rightarrow r_1$
i : $store$ $r_1, 0 \Rightarrow r_0$

La suite d'instructions, à gauche, donne lieu, après calcul du graphe de dépendance, au graphe de droite. Par exemple, l'instruction d lit les les registres r_1 et r_2 qui sont eux mêmes écrits respectivement par les instructions b et c, c'est un exemple de dépendance LAE. Les instructions e et g écrivent dans le même registre, il s'agit d'un exemple de EAE. L'instruction f écrit dans un des registres lus par l'instruction h, c'est un exemple de EAL.

Assignation de priorités aux instructions

Le principe est le suivant : on va sélectionner les instructions à ordonnancer en choisissant dans le graphe de dépendance les instructions qui n'ont plus de dépendances. Que faire en cas de choix ? Il faut s'appuyer sur des heuristiques pour pondérer les instructions. Par exemple, la somme des latences des instructions dont on dépend est une heuristique fréquemment utilisée.

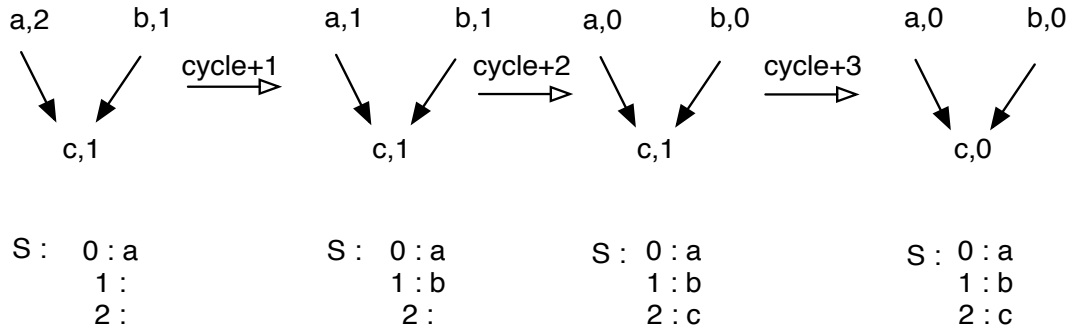


add :	1
$mult$:	2
$load$:	3
$store$:	4

Le tableau de gauche présente les latences de chacune des instructions. Le graphe de dépendance a été décoré avec la somme des latences des instructions qui dépendent de chaque instruction (chemin critique).

Simulation et production d'ordonnancement

Une fois cet arbre de dépendance décoré calculé, on va calculer un ordonnancement, c'est-à-dire une fonction qui à chaque cycle indique quelle est l'instruction à envoyer dans le processeur. Le schéma suivant est un exemple de production d'un ordonnancement (appelé S) à partir du graphe de dépendance décoré en haut à gauche du schéma. Chacune des lettres a, b et c est une instruction et est accompagné d'un entier qui représente sa latence. Au départ, seule a et b sont prêtes à être exécutées. On choisit d'exécuter a en premier car elle est sur le chemin critique. Par conséquent, l'ordonnancement associe a au cycle 0. Nous avons alors au cycle suivant le même graphe où la latence de l'instruction a a été décrémentée de 1 et le processus peut continuer. Quand une latence vaut 0, l'instruction associée n'est plus une dépendance. Dans notre exemple, les latences de a et b tombent à 0 au même cycle ce qui permet d'ordonnancer l'instruction c.



Suit le pseudo-code de l'algorithme.

- D est le graphe de dépendance.
- `Cycle` compte les cycles.
- `Ready` est la liste des instructions n'ayant plus de dépendances.
- `Active` est la liste des instructions en cours de traitement dans le processeur.
- S est l'ordonnancement, et le résultat de l'algorithme.
- *choose* dépend de l'heuristique
- *delay* est la latence
- Le predicat *is ready* indique qu'une instruction n'a plus de dépendances.

```
Cycle ⟵ 1
Ready ⟵ leaves of 𝒟
Active ⟵ ∅
while (Ready ∪ Active ≠ ∅)
 if Ready ≠ ∅then
  choose an op from Ready
  S(op) ⟵ Cycle
  add op to Active
 Cycle = Cycle + 1
 for each op ∈ Active
  if S(op) + delay(op) < Cycle then
   remove op from Active
   for each successor s of op in 𝒟
    if s is ready
     then add s to Ready
```

Chapitre 3

Conception et certification d'un validateur pour le *list scheduling*

L'algorithme de validation utilisé est principalement tiré du travail de George C. Necula [11] mais il est modifié pour mieux se prêter à la preuve. Il a la signature suivante :
Entrée : un programme p et un deuxième programme t_p présumé obtenu par la transformation d'ordonnancement.
Sortie : `vrai` si la sémantique est préservée, `faux` sinon.
En fait, c'est au corps de fonction[1] que l'on va s'intéresser par la suite. La preuve pour un programme dans son entier s'en déduit facilement puisque la transformation se fait fonction par fonction. L'algorithme de vérification procède bloc par bloc : pour chaque pair de blocs (original, transformé) on calcule une abstraction du code qui permet de décider si la sémantique à été préservée. Cette abstraction est calculée par évaluation symbolique [11] : il s'agit du concept clé de notre algorithme de validation et doit donc être formalisé en `Coq`.

3.1 Evaluation symbolique

3.1.1 L'évaluation symbolique

L'évaluation symbolique consiste à décrire la sémantique d'une suite d'instructions sans branchements par un ensemble de termes (un terme par ressource, une ressource étant pour l'instant une zone de stockage quelconque de la machine). Cet ensemble de termes est une représentation du code sous forme d'arbres et qui abstrait les détails syntaxiques, de la même manière que la syntaxe abstraite d'un langage fait disparaître certains détails syntaxiques de sa syntaxe concrète. Les détails syntaxiques qui disparaissent sont, par exemple, l'ordre d'évaluation, la propagation d'une constante ou la propagation d'une copie. C'est ce qui fait son intérêt en validation : si une transformation est purement syntaxique (comme dans les exemples précédents), alors l'évaluation symbolique des suites d'instruction sans branchements initiale et transformée nous permet de vérifier que la sémantique à été préservée. Il suffit pour cela de tester l'égalité entre les deux ensemble de termes associés aux suites d'instructions.

Notons que si deux termes associés à deux suites d'instructions sans branchements sont égaux, alors les sémantiques de ces deux suites d'instructions sont égales. Par contre, il existe deux suites d'instructions sans branchements ayant la même sémantique mais n'ayant pas les mêmes termes associés. C'est le cas par exemple si on transforme la multiplication d'une constante par zéro en zéro.

3.1.2 Machine abstraite

Dans le modèle machine utilisé par le compilateur, on dispose de trois types de ressources : les registres, la mémoire et la frame[2]. La mémoire et la frame sont des ressources pour lesquelles on ne peut pas toujours s'assurer que deux lectures ou deux écritures se font au même endroit. Ceci pose problème pour établir un ordonnancement correct et des techniques comme l'analyse d'aliasing [4] sont utilisées pour essayer de savoir quand deux adresses

[1] La vérification pourrait d'ailleurs se placer au niveau du bloc mais l'aproche serait moins généralisable à d'autres transformations.
[2] La frame est distincte de la mémoire à ce moment de la compilation, c'est un choix de conception [7].

sont distinctes. Nous utilisons une manière plus simple mais moins efficace de régler le problème en considérant une machine abstraite où la mémoire et la frame sont chacun un registre. On rejettera donc toute permutation d'instructions écrivant dans la mémoire même si celles-ci n'écrivaient pas à la même adresse. *mreg* est le type des registres machines. En COQ, nous déclarons :

Inductive *resource* : *Set* :=
| *Reg* : *mreg* → *resource*
| *Mem* : *resource*
| *Stack* : *resource*.

3.1.3 Lectures implicites

Le fait qu'une instruction lise la mémoire n'apparaît en général pas dans la syntaxe de la dite instruction puisque la mémoire est unique. De même, on n'indique pas le fait qu'on accède à la frame, cette dernière n'étant qu'une zone de la mémoire. Pour pouvoir procéder à l'évaluation symbolique de code de façon correcte, il faut néanmoins faire apparaître toutes les ressources lues ou écrites par une instruction. Le fait de modéliser la mémoire et la frame comme un registre a ici son importance : une instruction `store` par exemple se contente *a priori* d'écrire dans la mémoire. Cependant, la mémoire obtenue dans le cas concret par cette écriture est une fonction de la mémoire avant écriture (un seul mot mémoire a été modifié) et on doit donc considérer que l'on lit la mémoire lors d'un `store`.

3.1.4 Valeurs abstraites

On peut définir les valeurs abstraites obtenues par l'évaluation symbolique comme les expressions obtenues en remplaçant les ressources par les valeurs abstraites elles mêmes. L'idée derrière cette définition est d'effectuer les calculs sans connaître les valeurs des ressources. Par exemple, après évaluation symbolique de la suite d'instructions,

$$\begin{array}{lll} i_0 : & add & r_0, r_1 \Rightarrow r_0 \\ i_1 : & mult & r_0, r_1 \Rightarrow r_1 \end{array}$$

le registre r_1 contient $mult(add(r_0, r_1), r_1)$ et le registre r_0 contient $add(r_0, r_1)$

Définition 5. *valeur abstraite*

Expressions : $e ::=$ `Egetstack` *ofs* e
| `Esetstack` e *ofs* e
| `Egetparam` *ofs*
| `Eop` *op el*
| `Eload` *addr el* e
| `Estore` *addr el* e e
| `Ebase` r

Liste d'expressions : $el ::= nil \mid e :: el$

3.1.5 Etat abstrait

On définit l'état abstrait de la machine comme une fonction qui associant à chaque ressource une valeur abstraite. En COQ, il s'agit du type *Rmap.t* (*expression*).

Definition *forest* : *Set* := *Rmap.t* (*expression*).

3.1.6 Calcul de l'état abstrait : l'évaluation symbolique

On peut calculer l'état abstrait correspondant à une suite d'instructions. L'expression *'Rmap.set r e f'* fait correspondre la ressource r à l'expression e dans l'état abstrait f. L'expression *'Rmap.get r f'* renvoie l'expression associée à la ressource r dans l'état abstrait f.

Fixpoint *list_translation* (*l* : *list mreg*) (*f* : *forest*) {*struct l*} : *expression_list* :=

```
match l with
| nil ⇒ Enil
| cons i l ⇒ Econs (Rmap.get (Reg i) f) (list_translation l f)
end.

Definition update (f : forest) (i : instruction) : forest :=
match i with
    | Mgetstack ofs r ⇒
        Rmap.set (Reg r) (Egetstack ofs (Rmap.get Stack f)) f
    | Msetstack r ofs ⇒
        Rmap.set Stack (Esetstack (Rmap.get Stack f) (Rmap.get (Reg r) f) ofs) f
    | Mgetparam ofs r ⇒
        Rmap.set (Reg r) (Egetparam ofs) f
    | Mop op rl r ⇒
        Rmap.set (Reg r) (Eop op (list_translation rl f)) f
    | Mload addr rl r ⇒
        Rmap.set (Reg r) (Eload addr (list_translation rl f) (Rmap.get Mem f)) f
    | Mstore addr rl r ⇒
        Rmap.set Mem (Estore (Rmap.get Mem f) addr (list_translation rl f) (Rmap.get (Reg r) f)) f
    | Mcall sign id ⇒ f[3]
    | Mlabel lbl ⇒ f
    | Mgoto lbl ⇒ f
    | Mcond r lbl ⇒ f
    | Mreturn ⇒ f
end.

Fixpoint abstract_block (f : forest) (c : code) {struct c} : forest :=
  match c with
  | nil ⇒ f
  | cons (Mgoto lbl) l ⇒ f
  | cons (Mcall sig n) l ⇒ f
  | cons (Mlabel lbl) l ⇒ f
  | cons (Mcond cmp rl lbl) l ⇒ f
  | cons (Mreturn) l ⇒ f
  | i : : l ⇒ abstract_block (update f i) l
  end.
```

On note $empty$ la forêt initiale qui associe à chaque ressource r l'expression *Ebase r* représentant abstraitement la valeur de la ressource r au début du bloc de base.
On note $\alpha(i)$ pour *update(empty,i)* et $\alpha(c)$ pour *abstract_block(empty,c)*.
Enfin, on note $\alpha(i, S)$ pour l'application de $\alpha(i)$ à S.

3.1.7 Sémantique relationnelle

On donne aux états abstraits une sémantique relationnelle. On définit par induction mutuelle les relations $e :\longrightarrow^v$, $e :\longrightarrow^f$, $e :\longrightarrow^m$, $e :\longrightarrow^l$, $e :\longrightarrow^r$ et $e :\longrightarrow$.

- La relation ternaire $e : [rs, fr, m] \longrightarrow^v$ lie un état concret (registres, frame, mémoire) et la valeur abstraite e à une valeur.
- La relation ternaire $e : [rs, fr, m] \longrightarrow^f$ lie un état concret et la valeur abstraite e à une frame.
- La relation ternaire $e : [rs, fr, m] \longrightarrow^m$ lie un état concret et la valeur abstraite e à un état mémoire.
- La relation ternaire $e : [rs, fr, m] \longrightarrow^l$ lie un état concret et la valeur abstraite e à une liste de valeurs.
- La relation ternaire $e : [rs, fr, m] \longrightarrow^r$ lie un état concret et la valeur abstraite e à un banc de registres.
- La relation ternaire $e : [rs, fr, m] \longrightarrow$ lie un état concret et la valeur abstraite e à un état concret.

[3] Les valeurs de retour de la fonction update pour les instructions de branchements est un artefact puisqu'elle n'est en pratique appelée que sur des instructions sans branchements.

$$\frac{(ge, f, sp, parent) \vdash \ stack\_exp : [rs, fr, m] \longrightarrow^{f} \ fr' \quad get\_slot\ fr'\ ofs = \ v}{(ge, f, sp, parent) \vdash \ Egetstack\ ofs\ stack\_exp : [rs, fr, m] \longrightarrow^{v} \ v}$$

$$\frac{get\_slot\ parent\ ofs = \ v}{(ge, f, sp, parent) \vdash \ Egetparam\ ofs : [rs, fr, m] \longrightarrow^{v} \ v}$$

$$\frac{(ge, f, sp, parent) \vdash \ mem\_exp : [rs, fr, m] \longrightarrow^{m} \ m' \quad (ge, f, sp, parent) \vdash \ (rs\ r_1, ..., rs\ r_n) : [rs, fr, m] \longrightarrow^{l} \ lv \quad eval\_addressing addr\ lv = a \quad Mem.loadv\ m'\ a = \ v}{(ge, f, sp, parent) \vdash \ Eload\ addr\ (r_1, ..., r_n)\ mem\_exp : [rs, fr, m] \longrightarrow^{v} \ v}$$

$$\frac{(ge, f, sp, parent) \vdash \ (r_1, ..., r_n) : [rs, fr, m] \longrightarrow^{l} \ lv \quad eval\_operation\ ge\ sp\ op\ lv = \ v}{(ge, f, sp, parent) \vdash \ Eop\ op\ (r_1, ..., r_n) : [rs, fr, m] \longrightarrow^{v} \ v}$$

$$(ge, f, sp, parent) \vdash \ Ebase\ (Reg\ r) : [rs, fr, m] \longrightarrow^{v} \ (\ rs\ r)$$

$$\frac{(ge, f, sp, parent) \vdash \ stack\_exp : [rs, fr, m] \longrightarrow^{f} \ fr' \quad (ge, f, sp, parent) \vdash \ val\_exp : [rs, fr, m] \longrightarrow^{v} \ v \quad set\_slot\ fr'\ ofs\ v = \ fr''}{(ge, f, sp, parent) \vdash \ Esetstack\ stack\_exp\ val\_exp\ ofs : [rs, fr, m] \longrightarrow^{f} \ fr''}$$

$$(ge, f, sp, parent) \vdash \ Ebase\ Stack : [rs, fr, m] \longrightarrow^{f} \ fr$$

$$\frac{(ge, f, sp, parent) \vdash \ mem\_exp : [rs, fr, m] \longrightarrow^{m} \ m' \quad (ge, f, sp, parent) \vdash \ val\_exp : [rs, fr, m] \longrightarrow^{v} \ v \quad (ge, f, sp, parent) \vdash \ (rs\ r_1, ..., rs\ r_n) : [rs, fr, m] \longrightarrow^{l} \ lv \quad eval_a ddressing\ addr\ lv = a \quad Mem.storev\ m'\ v = \ m''}{(ge, f, sp, parent) \vdash \ Estore\ (r_1, ..., r_n)\ mem\_exp\ addr\ val\_exp : [rs, fr, m] \longrightarrow^{m} \ m''}$$

$$(ge, f, sp, parent) \vdash \ Ebase\ Mem : [rs, fr, m] \longrightarrow^{m} \ m \qquad (ge, f, sp, parent) \vdash \ Enil : [rs, fr, m] \longrightarrow^{l} \ nil$$

$$\frac{(ge, f, sp, parent) \vdash \ e : [rs, fr, m] \longrightarrow^{v} \ v \quad (ge, f, sp, parent) \vdash \ l : [rs, fr, m] \longrightarrow^{l} \ lv}{(ge, f, sp, parent) \vdash \ Econs\ e\ l : [rs, fr, m] \longrightarrow^{l} \ cons\ v\ lv.}$$

$$\frac{\forall\ x, (ge, f, sp, parent) \vdash \ f\ (Reg\ x) : [rs, fr, m] \longrightarrow \ (rs'(\ x))}{(ge, f, sp, parent) \vdash \ f : [rs, fr, m] \longrightarrow^{r} \ rs'}$$

$$\frac{(ge, f, sp, parent) \vdash \ f : [rs, fr, m] \longrightarrow^{r} \ rs' \quad (ge, f, sp, parent) \vdash \ f\ Stack : [rs, fr, m] \longrightarrow^{f} \ fr' \quad (ge, f, sp, parent) \vdash \ f\ Mem : [rs, fr, m] \longrightarrow^{m} \ m'}{(ge, f, sp, parent) \vdash \ f : [rs, fr, m] \longrightarrow \ rs'\ fr'\ m'}$$

La relation $e\ : [rs, fr, m] \longrightarrow$ a la propriété de déterminisme suivante :

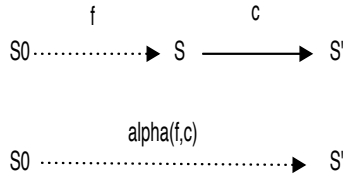
Lemme 1. $\forall (ge, f, sp, parent) \forall e \forall [rs, fr, m] \forall [rs', fr', m'] \forall [rs'', fr'', m''], (ge, f, sp, parent) \vdash e : [rs, fr, m] \longrightarrow [rs', fr', m'] \wedge (ge, f, sp, parent) \vdash e : [rs, fr, m] \longrightarrow [rs'', fr'', m''] \Longrightarrow [rs', fr', m'] = [rs'', fr'', m'']$

En conséquence de cette propriété, on se permettra d'écrire $\alpha(i, S) = S'$ ce qui est justifié par la fonctionnalité de notre sémantique relationnelle. La raison pour laquelle on n'a pas directement écrit une fonction pour décrire la sémantique des états abstraits est que cela se fait en Coq dans la sorte Set. Or, beaucoup d'opérations, comme par exemple *eval_operation*, renvoient un objet de type option ce qui rend l'écriture de la fonction très lourde puisqu'il faut vérifier que le résultat n'est pas *None*. La sémantique a donc été décrite dans la sorte Prop et est donc naturellement relationelle.

3.1.8 Une propriété de l'évaluation symbolique

Dans la preuve finale, on va chercher à montrer que, sous les hypothèses de bonne définition appropriées, l'égalité entre états abstraits implique l'équivalence entre sémantiques. Cette propriété est au coeur de l'algorithme de validation qui suit et elle a été prouvée en Coq. Pour cela, on utilise la propriété suivante qui affirme que si on peut passer d'un état S à un état S' pour un code c dans la sémantique standard, alors l'abstraction de c appliquée à l'état S se réduit vers l'état S'. Cela pose problème car la fonction *abstract_block* utilise un accumulateur pour construire sa valeur de retour. En effet, l'état atteint après $n+1$ instructions doit se calculer à partir de l'état atteint par les n instructions précédentes. Bien qu'on veuille montrer que la propriété est vraie quand on part d'un état abstrait vide, il faut prouver un résultat plus général, c'est-à-dire que la propriété est vraie pour tout état abstrait de départ afin de pouvoir utiliser l'hypothèse d'induction.

Soit f un état abstrait quelconque, c un code quelconque, S, S_0 et S' des états quelconques, les flèches en pointillé la sémantique des états abstraits et les flèches pleines la sémantique du code Mach cela donne (on veut prouver que la première ligne implique la seconde) :



Ce qui se prouve alors aisément par induction sur c. On en déduit notre théorème de correction en prenant $S = S_0$, c étant une séquence d'instructions sans branchement.

Lemme 2. $\forall c \forall c' \forall S \forall S', [c{+}{+}c', S] \xrightarrow{*} [c', S'] \Rightarrow \alpha(c, S) = S'$

3.2 Algorithme de validation

La conception d'un programme de validation certifié pour une transformation est un travail en deux étapes. La première est de concevoir un programme qui implique une préservation de la sémantique. La seconde est de reconcevoir le même programme tel qu'il soit non-seulement implémentable mais également certifiable, ici en COQ. Cela impose au moins deux contraintes sur le programme : d'abord, il doit être purement fonctionnel, ensuite, il doit se prêter à la preuve. Par exemple, on préférera un programme récursif à la complexité moins bonne mais dont les propriétés peuvent se prouver à l'aide d'un principe d'induction simple.

Le principe d'un validateur usuel (c'est-à-dire qu'on ne cherche pas à certifier) fonctionnant pour du *list scheduling* est le suivant : on calcul les deux graphes de flot de contrôle, on reconstruit une bijection entre les blocs de bases et pour chaque pair de blocs (initial, transformé) on vérifie qu'ils sont égaux modulo évaluation symbolique. Dans la suite, nous présentons un programme de validation certifié dont la structure diffère beaucoup de cette idée. Cela est dû, comme on l'a dit, aux contraintes imposées par l'assistant de preuve.

La fonction *validate* est le programme de validation que l'on doit certifier. On définit d'abord la vérification des branchements qui réussit uniquement si les deux branchements sont égaux. La fonction *nbranchset i* réussit si i est une instruction sans branchement. La fonction *validate* parcourt le code et vérifie que les évaluations symboliques des couples (bloc initial, bloc transformé) donne le même état abstrait. La fonction *check* décide de l'égalité entre deux états abstraits.

Definition *check_glue x1 x2* :=
if instruction_eq x1 x2 then true else false [4]

[4] Cette écriture redondante est due au fait que *instruction_eq x1 x2* n'est pas de type booléen mais de type *"l'égalité entre instructions est décidable"*

```
Fixpoint validate(c tc : code) (f tf : forest) {struct c} : bool :=
  match c with
  | nil ⇒ match tc with
                | nil ⇒ true
                | _ ⇒ false
                end
  | cons i l ⇒
                match tc with
                | nil ⇒ false
                | cons i' l' ⇒
                    if nbranchset i && nbranchset i' then validate l l' (update f i) (update tf i')
                    else
                    check_glue i i' && check f tf && validate l l' empty empty
                end
  end.
```

Soient deux suites d'instructions, c et t_c, t_c étant supposé être le résultat de la transformation de c par *list scheduling*. Supposons que c soit de la forme $i_0,\ i_1,\ i_2,\ i_3,\ i_4,\ i_5,\ i_6,\ i_7,\ i_8,\ ..., i_n$ et que t_c soit de la forme $j_0, j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, ..., j_n$. Supposons par ailleurs que i_3 est une instruction de branchement. L'idée de l'algortihme de vérification est la suivante : on considère la suite d'instructions qui précède la première instruction de branchement, ici i_3.
$\underbrace{i_0,\ i_1,\ i_2}$, $i_3,\ i_4,\ i_5,\ i_6,\ i_7,\ i_8,\ ..., i_n$. De même, on considère la suite d'instuction dans t_c allant jusqu'à j_3,
$\underbrace{j_0,\ j_1,\ j_2}, j_3, j_4, j_5, j_6, j_7, j_8, ..., j_n$. On calcule les évaluations symboliques $\alpha(empty, i_0; i_1; i_2)$ et $\alpha(empty, j_0; j_1; j_2)$ de ces deux suites d'instructions respectivement et on teste leur égalité. Si $\alpha(empty, i_0; i_1; i_2) = \alpha(empty, j_0; j_1; j_2)$ alors on vérifie que $i_3 = j_3$ et on reprend l'algorithme avec $c = i_4, i_5, i_6, i_7, i_8, ..., i_n$ et $t_c = j_4, j_5, j_6, j_7, j_8, ..., j_n$. Sinon on considère que la transformation est incorrecte. Si l'algorithme arrive jusqu'au bout des deux séquences d'instructions, alors on considère que la transformation est correcte.

Une limitation de la définition choisie apparaît. En effet, on requiert que les suites d'instructions à comparer entre les deux codes soient de la même taille. Cela ne va pas à l'encontre du *list scheduling* et n'est donc pas, à proprement parler, une limitation, cependant on pourrait lever cette restriction et rendre le validateur plus général. La raison pour laquelle c'est la version restreinte qui est utilisée et que cela simplifie grandement la certification en Coq. En effet, Coq exige que toutes les fonctions définies terminent or c'est trivialement le cas dans notre exemple puisque la fonction est définie par induction structurelle sur la longueur du code.

3.3 Preuve de correction

Nous avons programmé un vérificateur en COQ, il nous faut maintenant prouver qu'il est correct. La validation prend place au niveau du code et nous allons donc prouver que si la validation réussit pour deux codes quelconques, alors ils ont la même sémantique. La preuve pour tout le programme[5], bien que techniquement difficile car impliquant des principes d'induction mutuelle, n'a pas d'intérêt en soit et nous ne nous attarderons donc pas dessus.

3.3.1 Une sémantique de blocs pour le langage Mach

Comme on le voit dans l'algorithme de validation, on vérifie que la sémantique est préservée à chaque branchement, il nous faut donc concevoir une sémantique adaptée à cette vérification. On cherche à prouver que si la sémantique du code initial est bien définie et que l'algorithme de validation réussi alors la sémantique du code transformé est égale à la sémantique du code initial. On va faire cette preuve par induction sur la construction de la sémantique initiale. Le pas de base de la sémantique standard est l'instruction, ce qui ne va pas. On a donc conçu une sémantique dont le pas de base est la suite d'instructions sans branchements et que l'on appelle *sémantique de blocs*. Le pas de base est le bloc d'instructions sans branchement que l'on ajoute au code par la règle `add`.

[5] ainsi que la construction d'un list scheduler pour programme à partir d'un list scheduler pour corps de fonction

Ce principe est probablement généralisable. D'un côté, on peut concevoir des algorithmes de vérification dont les vérifications entre code initial et transformé seraient plus espacées que maintenant (où on vérifie à chaque branchement). De l'autre concevoir une sémantique dont le pas de base corresponde à ces vérifications. Nous reviendrons sur cette idée au chapitre 4.

Cette sémantique de bloc a la forme suivante :

Définition 6. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i} [c', rs', fr', m']}$

Il s'agit des règles `Mgetstack`, `Msetstack`, `Mgetparam`, `Mload`, `Mstore`, `Mop` définies dans la section 2.2. Ces règles correspondent aux instructions sans branchements.

Définition 7. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i^*} [c', rs', fr', m']}$

$$(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i^*} [c, rs, fr, m] \text{ (refl)}$$

$$\frac{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i} [c', rs', fr', m']}{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i^*} [c', rs', fr', m']} \text{ (one)}$$

$$\frac{\begin{array}{l}(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{i^*} [c_2, rs_2, fr_2, m_2] \\ (ge, f, sp, parent) \vdash [c_2, rs_2, fr_2, m_2] \xrightarrow{i^*} [c_3, rs_3, fr_3, m_3]\end{array}}{(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{i^*} [c_3, rs_3, fr_3, m_3]} \text{ (trans)}$$

Un bloc de base est la clôture réflexive et transitive de l'exécution d'instructions sans branchements.

Définition 8. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{b} [c', rs', fr', m']}$

Il s'agit des règles `Mlabel`, `Mcall`, `Mgoto`, `Mcondtrue`, `Mcondfalse` définies dans la section 2.2. Ces règles correspondent aux instructions de branchements. Notre algorithme de validation ne reconstruit pas le graphe de flot de contrôle et on doit donc considérer un label comme un branchement, il s'agit d'un point d'entrée.

Définition 9. $\boxed{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*}_b [c', rs', fr', m']}$

$$\frac{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{i^*} [c', rs', fr', m']}{(ge, f, sp, parent) \vdash [c, rs, fr, m] \xrightarrow{*}_b [c', rs', fr', m']} \text{ (first)}$$

$$\frac{\begin{array}{l}(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{i^*} [c_2, rs_2, fr_2, m_2] \\ (ge, f, sp, parent) \vdash [c_2, rs_2, fr_2, m_2] \xrightarrow{b} [c_3, rs_3, fr_3, m_3] \\ (ge, f, sp, parent) \vdash [c_3, rs_3, fr_3, m_3] \xrightarrow{*}_b [c_4, rs_4, fr_4, m_4]\end{array}}{(ge, f, sp, parent) \vdash [c_1, rs_1, fr_1, m_1] \xrightarrow{*}_b [c_4, rs_4, fr_4, m_4]} \text{ (add)}$$

L'exécution d'une suite d'instructions est soit l'exécution d'un bloc de base soit une suite d'instructions à laquelle on ajoute un bloc de base en plaçant un branchement entre les deux.

Notations

Dans la suite du chapitre on note [c,S] pour l'état [c,rs,fr,m] et on ne précisera pas l'environnement dans les réductions.

Cette sémantique de blocs à la propriété suivante d'équivalence avec la sémantique standard :

Proposition 1. $\forall c \forall c' \forall S \forall S', [c, S] \xrightarrow{*} [c', S'] \Leftrightarrow [c, S] \xrightarrow{*}_b [c, S']$

Cette propriété est une des raisons principales pour le choix du langage Mach. En effet, cette étude portait à l'origine sur un autre langage intermédiaire, l'assembleur POWERPC lui même. Le choix du langage POWERPC est *a priori* plus naturel que celui du langage Mach car c'est généralement sur l'assembleur qu'ont lieu les optimisations d'ordonnancement. Néanmoins, il s'avère que cette propriété d'équivalence sémantique n'est pas prouvable en POWERPC car le concept de bloc n'est en fait pas clair : rien n'empêche en POWERPC de modifier l'adresse de retour d'une fonction et par conséquent toute position dans le code peut être atteinte.

3.3.2 Intuition de la preuve de correction faible

Dans les grandes lignes l'idée de la preuve est la suivante. Les sémantiques standard et de blocs étant équivalentes on peut prouver nos propriétés sur la sémantique de blocs. On prouve que que le validateur implique la préservation faible de la sémantique au niveau du bloc. Par induction sur la sémantique de blocs du code initial, on montre que le validateur implique la propriété pour le code tout entier puis pour le programme tout entier.

Passage du statique au dynamique

La vérification effectuée par l'algorithme est statique : une fois un branchement vérifié, on continue la vérification sur le code restant. A l'exécution, un branchement peut remplacer le code à exécuter par un autre code (issu du corps de la fonction). On a donc besoin de vérifier que la propriété de validation se transmet (b est un branchement quelconque).

Cette propriété est en fait assez évidente. D'une part, il existe toujours une suite à un branchement puisqu'il doit y avoir une correspondance parfaite entre les labels pour que la transformation soit validée et qu'un branchement ne peut avoir de suite que si la version transformée en a aussi. Pour ce qui est de la propagation de la propriété de validation, les codes obtenus par un branchement correspondent toujours à un code postfixe du code de la fonction qui a été validée et la propriété de validation est donc vérifiée.

Format du code

Dans la preuve on utilise abondamment le fait que la vérification implique que les deux codes ont le même format, c'est-à-dire qu' à un branchement dans le code initial correspond un branchement dans le code transformé, idem pour les suites d'instructions sans branchements. De même, la validation implique que si le code initial est non vide alors le code transformé non plus. Il y a de nombreuses propriétés de la sorte qui sont utilisées dans la preuve. Leur intérêt n'est pas grand mais leur preuve peut parfois être étonnament laborieuse.

Idée de la preuve

Le lemme qui suit constitue le coeur de la preuve : on montre que la validation implique une préservation sémantique au niveau du bloc.

Lemme 3. $\forall c \forall c' \forall t_c \forall S \forall S', valid(c, t_c) = true \Rightarrow [c{+}{+}c', S] \xrightarrow{i^*} [c', S'] \Rightarrow \exists t'_c [t_c{+}{+}t'_c, S] \xrightarrow{i^*} [t'_c, S']$

Démonstration. La sémantique du code initial est bien définie, d'après le lemme 2, l'état abstrait associé au bloc se réduit vers le même état concret. De même, d'après l'hypothèse de bonne définition de la sémantique du code transformé, il existe un état concret vers lequel se réduit l'état abstrait associé au code transformé. La validation ayant réussi, les deux états abstraits sont égaux et en conséquence les deux états concrets le sont aussi. La sémantique du code transformé se réduit donc vers le même état concret que la sémantique du code initial. □

Le lemme suivant montre que la sémantique est préservée au niveau du code pour la sémantique de blocs.

Lemme 4. $\forall c \forall c' \forall t_c \forall t'_c \forall S \forall S', valid(c, t_c) = true \Rightarrow [c, S] \xrightarrow{*}_b [c', S'] \Rightarrow [t_c, S] \xrightarrow{*}_b [t'_c, S']$

Démonstration. Par induction mutuellement récursive sur la sémantique du code initial. Le cas de base correspond à une suite d'instructions sans branchement et sa correction découle du lemme précédent. Le cas inductif correspond à une suite d'instructions sans branchements suivie d'un branchement suivi d'un code quelconque pour lequel on sait par induction que la validation implique que la sémantique est équivalent à celle du code transformé. Là encore le lemme précédent nous permet d'arriver au branchement en préservant la sémantique, puis lemme de propagation de la propriété au niveau des branchements nous permet d'utiliser l'hypothèse d'induction. □

Bien que cette preuve soit conceptuellement très simple, sa formalisation requiert de très nombreuses réécritures (les lemmes sur le format) et l'induction mutuellement récursive impose un choix judicieux des hypothèses d'induction.

Le théorème final stipule que la sémantique est préservée au niveau du code en sémantique standard, on suppose que le code transformé termine.

Proposition 2. $\forall c \forall c' \forall t_c \forall t'_c \forall S \forall S', valid(c, t_c) = true \Rightarrow [c, S] \xrightarrow{*} [c', S'] \Rightarrow [t_c, S] \xrightarrow{*} [t'_c, S']$

Démonstration. Les sémantiques de blocs et standard étant équivalentes, les propriétés sur la sémantique de blocs énoncées dans le lemme précédent sont vraies également pour la sémantique standard. □

3.3.3 La validation présentée n'implique pas la propriété de préservation forte

Dans la preuve du théorème de préservation sémantique faible pour le validateur présenté précédemment on fait l'hypothèse que la sémantique du programme transformé est bien définie. Intuitivement, si un code c est transformé par *list scheduling* en un code t_c et si c est sémantiquement bien défini, alors t_c devrait l'être aussi et l'on devrait donc pouvoir se séparer de cette hypothèse.

Cependant, la validation présentée précédemment n'implique pas la bonne définition du code transformé, comme le montre le contre exemple qui suit :

Contre exemple 1. *Considérons le code constitué de l'unique instruction* $move\ 2 \Rightarrow r_1$ *et supposons qu'il soit transformé en la suite d'instructions :*

$$\begin{array}{lll} i_0 : & load & 0 \Rightarrow r_1 \\ i_1 : & move & 2 \Rightarrow r_1 \end{array}$$

L'évaluation symbolique du premier code à partir de l'état S *donne l'état abstrait* $S\#r_1 \leftarrow 2$*. L'évaluation symbolique de second code à partir du même état* S *donne l'état abstrait* $S\#r_1 \leftarrow 2$*. Les deux états abstraits étant égaux ils sont considérés comme sémantiquement équivalents par le validateur. Pourtant, il est clair que le premier est sémantiquement bien défini alors que le second ne l'est pas : son évaluation échoue sur une lecture de la mémoire à l'adresse invalide 0.*

Ce contre exemple ne signifie pas que la propriété de préservation forte n'est pas prouvable mais que la validation actuelle ne l'implique pas. Il est donc nécessaire de rendre la validation plus complète afin de s'assurer que la bonne définition de la sémantique est bien préservée par la transformation. Ce travail est en cours.

Chapitre 4

Algorithmes de validation pour des transformations plus complexes

Le *list shceduling* est un algorithme représentatif de la famille des algorithmes d'ordonnancement mais il en existe de plus complexes. Nous avons commencé l'étude de deux autres algorithmes classiques : le *trace scheduling* et le *software pipelining*. A notre connaissance, il n'existe pas d'algorithmes de validation pour ces transformations. Dans ce chapitre, nous essayons de caractériser les différences entre ces algorithmes et le *list scheduling* et nous proposons un algorithme de validation pour le *trace scheduling* basé sur les idées du chapitre précédent. Nous montrons également que ces idées ne s'étendent pas au *software pipelining*.

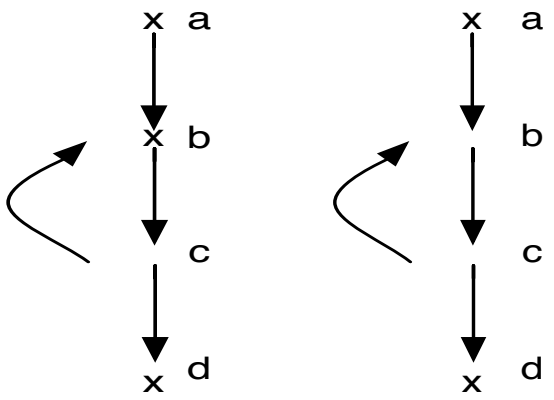
4.1 Complexité des transformations

L'optimisation que l'on a étudiée fait partie des optimisations les plus conservatives : elle ne modifie pas la structure du programme et les transformations sont assez superficielles. Intéressons nous au cas de deux autres optimisations classiques de la famille de l'ordonnancement. Pour pouvoir avoir une idée des problèmes introduits par ces optimisations, posons quelques définitions.

Définition 10. Transformation préservant la structure *On dit qu'une transformation préserve la structure du programme si le graphe de flot de contrôle est inchangé, c'est à dire que seul les blocs sont modifiés.*

Définition 11. Transformation à traces finies *On dit qu'une transformation est à trace finie si, pour la valider, on peut placer des points de vérification de telle manière que les morceaux de codes qui les séparent soient de taille finie.*

Exemple 1. *trace finies et infinies*



Dans les deux graphes de flots de contrôle précédents, on indique par des croix les points de vérification. Dans le graphe de gauche, il n'y a que 4 flots : ab, bcb et bcd. Par contre, dans le graphe de droite on a : $abcd$, $abcbcd$, $abcbcbcd$ et ainsi de suite, c'est à dire, en notation régulière : $a(bc)^*d$, ce qui représente une infinité de flots. Une transformation opérant sur les traces du graphe de gauche est à traces finies tandis que l'autre est à traces infinies.

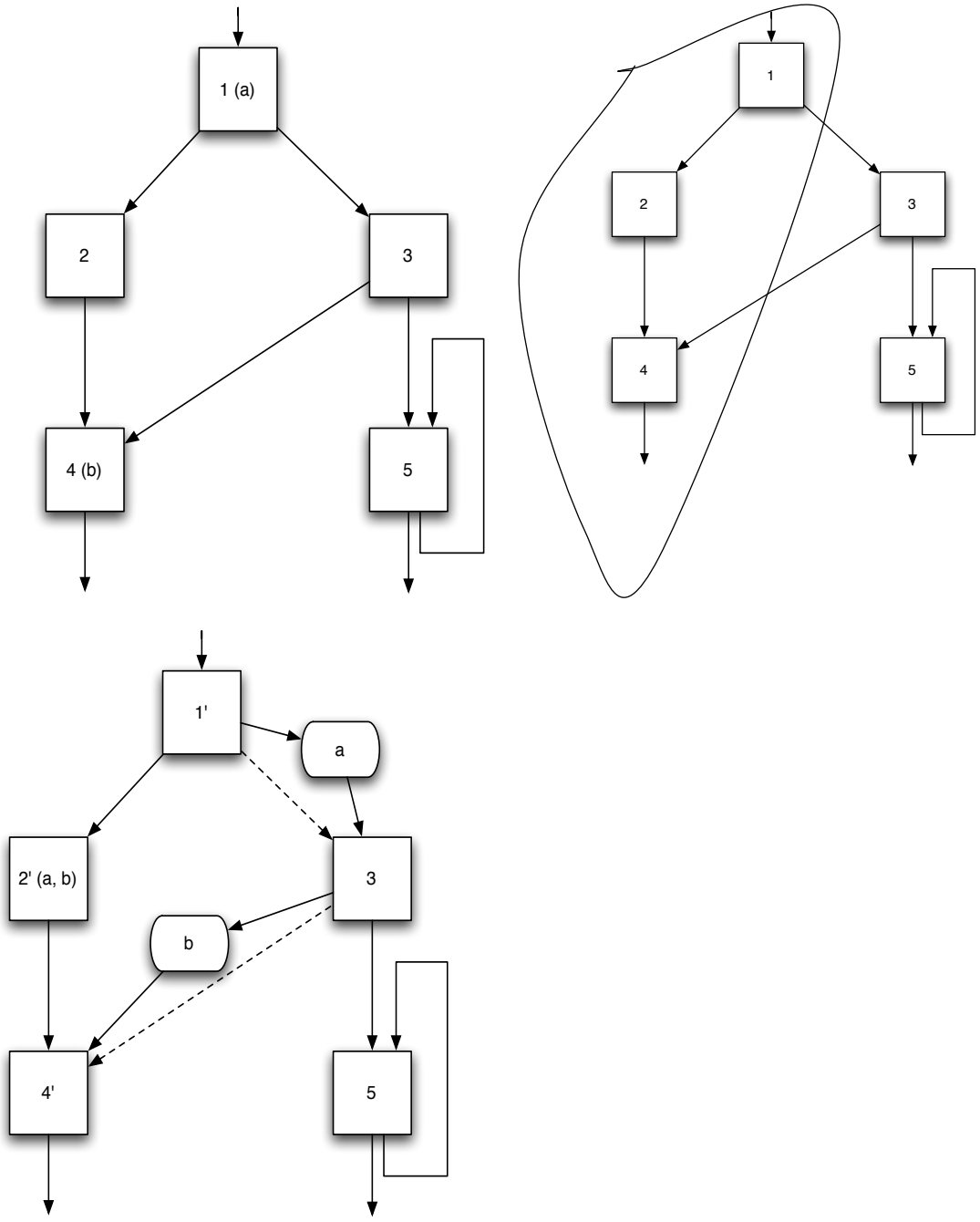
Le *list scheduling* que nous avons étudié est une transformation qui préserve la structure et qui est à trace finies (les traces étant les blocs). Un exemple de transformation ne préservant pas la structure mais qui reste à traces finies est le trace scheduling presenté dans [6] et qui cherche à tirer parti des architectures VLIW[1]. Une transformation qui à la fois modifie la structure et est à traces infinies est le software pipelining presenté dans [4] et qui cherche à tirer profit du pipeline du processeur au niveau des boucles.

4.2 Trace scheduling

Le trace scheduling est une technique d'ordonnancement qui généralise le list scheduling. En trace scheduling, on ne se limite pas à ordonnancer des blocs d'instructions sans branchement : des instructions peuvent passer derrière un branchement conditionnel ou un goto et passer devant un label. Les morceaux de code à ordonnancer sont appelés des traces et sont choisies en utilisant des heuristiques ou des prévisions sur l'exécution. Une fois la trace ordonnancée, en utilisant la même méthode que pour le list scheduling, il faut procéder au *bookkeeping* de cette trace : si une instruction est passée derrière un goto ou un branchement conditionnel alors, dans le cas où le branchement est pris, il faut s'assurer d'exécuter l' instruction manquante avant d'atteindre la cible. Pour une instruction qui est passée devant un label il faut s'assurer qu'elle est exécutée dans le cas où on entre dans la trace par ce label. Une propriété importante du trace scheduling est qu'une trace ne peut pas recouvrir une boucle. Cela signifie qu'une trace externe à une boucle doit s'arrêter avant son commencement et qu'une trace interne à une boucle doit s'arrêter avant la fin de la boucle.

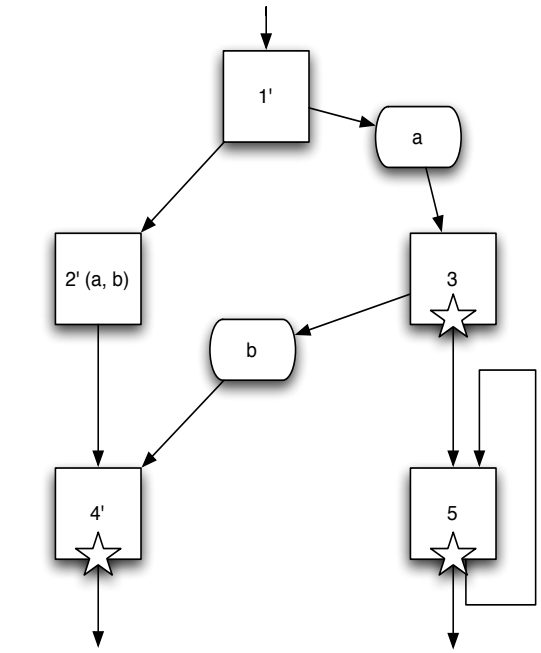
Les schémas qui suivent illustrent cet algorithme. Le premier montre un graphe de flot de contrôle (cfg). Il est constitué de 5 blocs. Le bloc numéro 5 est une boucle. Les blocs 1 et 4 contiennent respectivement une instruction a et une instruction b (entre autres). Le schéma 2 montre quelle est la trace choisie pour être ordonnancée : il s'agit de la suite de blocs 1, 2 et 4. Supposons que l'instruction a soit déplacée du bloc 1 vers le bloc 2 et que l'instruction b soit déplacée du bloc 4 vers le bloc 2 lors de l'application de l'algorithme de *trace scheduling* à la trace. Le schéma 3 montre les modifications à apporter au graphe pour que la sémantique soit préservée. Au cas où l'on sort de la trace à la fin du bloc 1' (le bloc 1 dépourvu de l'instruction a) il faut nécessairement exécuter l'instruction a avant d'arriver au bloc 3 puisqu'elle n'a pas encore été exécutée. Au cas où l'on rentre dans la trace au niveau du bloc 4' (bloc 4 privé dépourvu de son instruction b), il faut nécessairement exécuter l'instruction b avant d'arriver au bloc 4' puisque b doit avoir été exécutée.

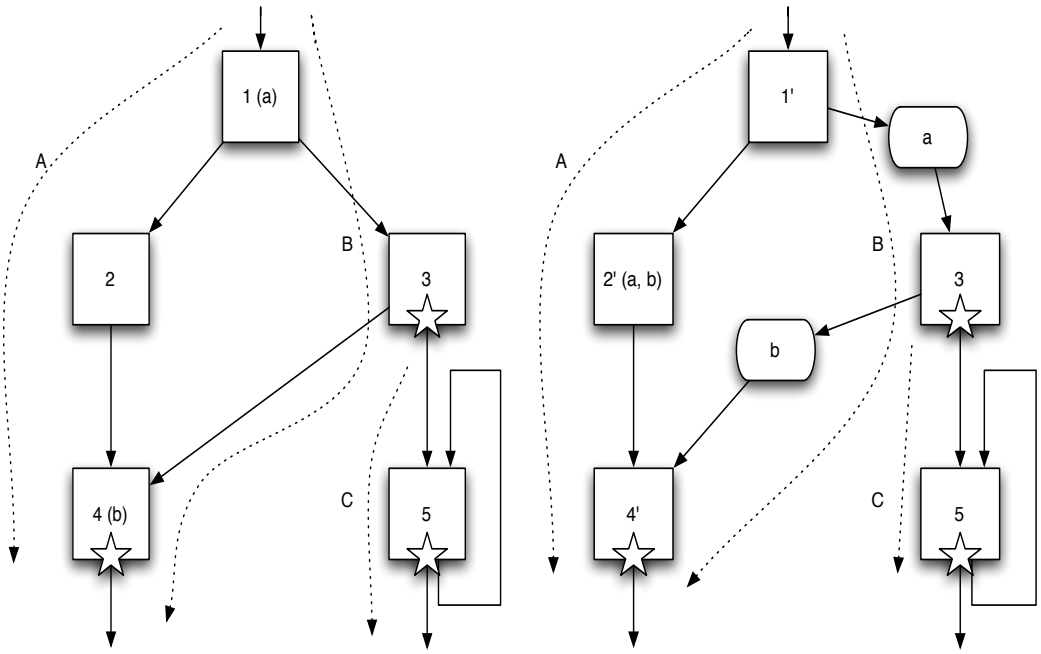
[1] Very Large Instruction Width. Technologie rencontrée notamment dans les processeurs de type *Itanium*



Le validateur présenté au chapitre précédent ne peut pas fonctionner pour cette optimisation. En effet, notre validateur vérifie que la sémantique est préservée dès qu'il atteint un branchement, ce qui est trop restrictif pour le trace scheduling puisque des instructions peuvent passer avant ou après. Par ailleurs, le structure du graphe de flot de contrôle n'est pas préservée par cette transformation (bookkeeping) et donc un parcours linéaire du code n'est plus envisageable, les deux codes n'ont même pas la même longueur. Nous avons implémenté un validateur non certifié pour ce type de transformation : tout d'abord on calcule les graphes de flots de contrôle associés à nos deux codes. Ensuite on calcul les *arbres de domination* [4] afin de déterminer où sont les points d'entrées des boucles qui deviennent nos points de vérification. Ensuite on compare toutes les traces allant du point d'entrée du programme à un point de sortie en effectuant une évaluation symbolique si on rencontre un point de vérification.

Les schémas qui suivent illustrent le principe de l'algorithme de vérification. Dans le premier schéma on indique par des étoiles les points de vérification assortis à la version modifiée du cfg. Un point de contrôle correspond soit à une sortie de programme soit à l'entrée dans une boucle, ce que l'on détermine en calculant l'arbre de domination du cfg. Dans les schémas suivants, les flèches pointillées indiquent les diverses traces des deux versions du cfg. L'algorithme de vérification procède à l'évaluation symbolique des deux versions des traces A, B et C qui doivent être égales pour que la transformation soit considérée comme correcte.



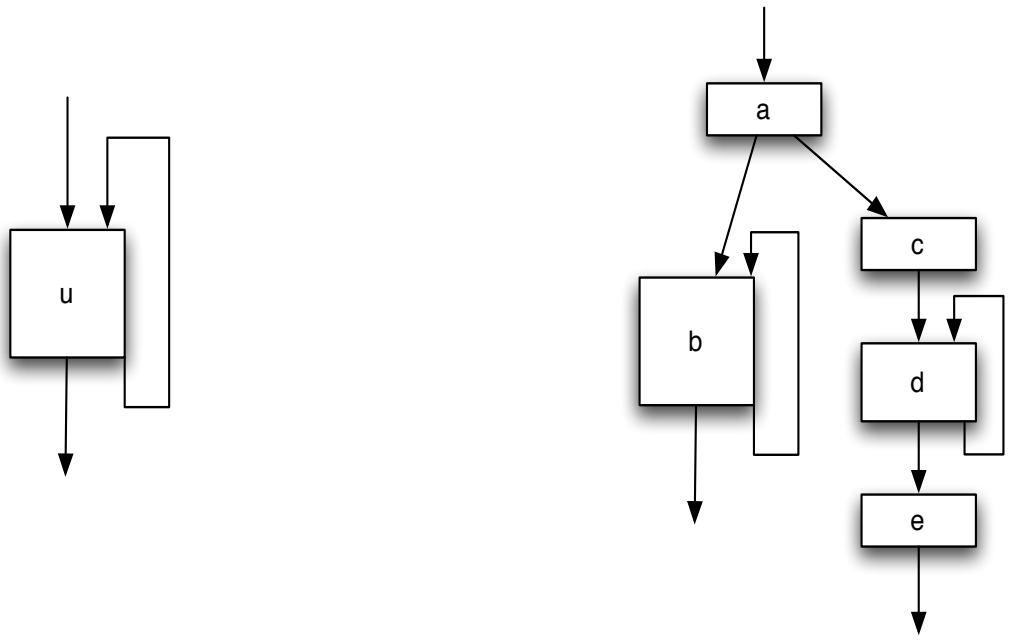


Ce validateur n'a pas encore été implémenté en Coq mais il semble que la méthodologie utilisée pour le list scheduling reste applicable. L'unité de vérification est maintenant la trace. On vérifie donc l'équivalence sémantique entre deux traces de la même façon qu'entre deux blocs : grâce à l'évaluation symbolique. On doit alors définir une sémantique de traces équivalentes à la sémantique standard et ayant comme pas de base la trace au lieu de l'instruction. Cette certification va être développée afin de pouvoir confirmer ces conjectures.

4.3 Software pipelining

Le software pipelining est une technique d'ordonnancement qui se concentre sur les boucles. En effet, dans les techniques précédemment présentées on ne prend pas en compte le fait qu'une boucle puissent être exécutée plusieurs fois. En software pipelining on cherche à réduire le temps d'exécution des itérations de la boucle. Pour cela on va, lors d'une exécution du corps de la boucle, effectuer en parallèle les calculs de différentes itérations de la boucle à un instant donné. L'algorithme est complexe et nous ne le présenterons pas ici : nous n'avons d'ailleurs pas pu l'implémenter complètement car il oblige l'allocation de registre à avoir lieu après l'ordonnancement ! Comme on l'a dit, le software pipelining ne préserve pas la structure et il n'est pas à traces finies : une boucle va généralement être transformée en un branchement conditionnel qui va soit exécuter la boucle normale soit la version rapide (en fonction du nombre d'itérations à faire). Or si la boucle est bel et bien coupée dans la version initiale, il n'en est pas de même dans la version transformée ou le point de vérification se situe avant le branchement conditionnel. Par conséquent l'évaluation symbolique ne peut plus être utilisée et notre algorithme basé sur cette abstraction ne terminerait pas.

Le schéma qui suit présente une boucle (à gauche) et sa version transformée après software pipelining (à droite). Le bloc a teste si le nombre de tours à effectuer est suffisant pour utiliser la version optimisée de la boucle. Si ce n'est pas le cas, le contrôle passe au bloc b qui est une copie de u. Si le nombre de tours à faire est suffisamment grand, on passe à la version optimisée. On traverse d'abord le bloc c dit *prologue* puis on arrive à la boucle optimisée d dit *bloc d'état stable* et dont on sort en passant par le bloc e dit *épilogue*. La boucle d'état stable exécute plusieurs itérations de la boucle initiale simultanément d'où la présence d'un prologue et d'un épilogue.



L'exemple nous permet de comprendre qu'on ne peut pas placer de point de vérification faisant correspondre les deux graphes. Un point correct serait avant le bloc a mais on aurait une boucle infinie en d. Placer un point de vérification avant d afin de couper la boucle ne correspond à aucun point de vérification dans u car on exécute plusieurs itérations de u à la fois.

Chapitre 5

Conclusion

Ce développement, en plus de montrer que la certification de validateurs est faisable, semble indiquer qu'effectivement la validation peut avoir des avantages sur la certification. Certains compilateurs vont, pour leur ordonnancement de blocs de base, calculer plusieurs ordonnancement par bloc en utilisant différentes heuristiques : ils comparent les résultats et gardent le meilleur ordonnancement. On peut implémenter une telle approche sans que cela ait le moindre impact sur le validateur certifié. D'autre part, ce développement suggère une méthode de développement : on décide des points de programmes pour lesquels on vérifie que la sémantique a été préservée (dans notre exemple, les branchements), ce qu'on fait par évaluation symbolique. Enfin, dans le but de prouver que cette validation est correcte, on conçoit une sémantique équivalente à la sémantique standard mais qui ait comme pas de base non plus l'instruction mais les morceaux de codes entre deux points de vérification (dans notre exemple, les suites d'instructions sans branchement). Beaucoup de travail reste néanmoins à faire et nous donnons dans la suite deux pistes.

Amélioration du validateur

On peut essentiellement améliorer un validateur certifié de deux façons : soit en certifiant des propriétés plus fortes, soit en le rendant plus complet, ces deux voies restent ouvertes.

En effet la propriété vérifiée par le validateur actuel est la préservation faible de la sémantique et on souhaite vérifier une préservation forte. On a montré dans le chapitre 3 les différences entre ces deux validations et indiqué comment passer à la version forte.

L'équivalence entre programmes n'est pas décidable et nous faisons le choix de la correction pour nos validateurs, par conséquent ils ne peuvent être complets. On peut combattre ce problème en restreignant le champ d'application du validateur. En effet, un validateur supposé valider n'importe quelle optimisation sera très incomplet mais si on le restreint par exemple aux optimisations d'ordonnancement de blocs, alors on peut le rendre complet. Notre validateur est d'ailleurs probablement complet pour le list scheduling selon ce dernier point de vue. Pour généraliser un peu le validateur sans trop restreindre sa complétude il faudrait relâcher la contrainte sur les longueurs de code détaillées au chapitre 3.

Validation et PCC

Le *proof-carying code* [10] nous permet de concevoir une approche légèrement différente de la validation de transformation. Dans notre approche, la code à transformer c est envoyé au transformateur qui renvoie t_c. Ensuite on vérifie par analyse statique que t_c à la même sémantique que c. Dans une approche plus PCC, on enverrait le code c à un transformateur qui non-seulement nous renverrait le code t_c transformé mais également une preuve π que la sémantique est préservée. On se contente alors de vérifier que π est bel est bien la preuve que t_c à la même sémantique que c.

Il existe en fait un continuum entre les deux approches. Par exemple, dans le cas d'une transformation ne préservant pas la structure mais à traces finies, un transformateur pourrait renvoyer le code transformé *et* une fonction indiquant la correspondance entre les traces. Le validateur peut alors utiliser cette information (cette

preuve) pour vérifier que la sémantique est belle et bien préservée au niveau des traces et donc au niveau du programme tout entier.

Bibliographie

[1] certified translation validation : http ://pauillac.inria.fr/ tristan/tv.

[2] The compcert certified compiler back-end : http ://pauillac.inria.fr/ xleroy/compcert-backend/.

[3] The coq proof assistant : http ://coq.inria.fr/.

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

[5] Yves Bertot Pierre Castéran. *Theorem prooving and program development*. Springer, 2004.

[6] John R. Ellis. *Bulldog : a compiler for VLSI architectures*. ACM Doctoral Dissertation Awards, 1986.

[7] Xavier Leroy. Formal certification of a compiler back-end. In ACM Press, editor, *Symposium on Principles Of Programming Languages Conf. (POPL'06)*, 2006.

[8] L. Zuck A. Pnueli R. Leviathan. Validation of optimizing compilers. Technical report, Weizmann institute of Science, 2001.

[9] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997.

[10] George C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages Conf. (POPL'07)*, 1997.

[11] George C. Necula. Translation validation for an optimizing compiler. In *Conference on Programming Languages Design and Implementation. (PLDI'00)*. ACM Press, 2000.

[12] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Symposium on Principles Of Programming Languages Conf. (POPL'04)*, 2004.

[13] A. Pnueli M. Siegel O. Shtrichman. The code validation tool (cvt)- automatic verification of a compilation process. In *Software Tools for Technology Transfer*, volume 2, 1999.

[14] A. Pnueli M. Siegel E. Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'98*, 1998.

[15] Keith D. Cooper Linda Torczon. *Engineering a Compiler*. Morgan Kaufman, 2004.

[16] L. Zuck A. Pnueli Y.Fang and B. Goldberg. Voc : A translation validator for optimizing compilers. In *Internation workshop on Compiler Optimization meets Compiler Verificaiton, COCV'02*, 2002.

[17] L. Zuck A. Pnueli Y.Fang and B. Goldberg. Voc : A methodology for translation validation of optimizing compilers. In *Journal of Universal Computer Science*, to appear.

Annexe A

Programmes et enoncés COQ

Le développement COQ qui suit est la spécification complète de validateur ainsi que les énoncés prouvés et utilisés dont nous avons enlevé les scripts de preuve.

```
Require Import Coqlib.
Require Import Maps.
Require Import AST.
Require Import Integers.
Require Import Op.
Require Import Mach.
Require Import Locations.
Require Import Maps.
Require Import Floats.
Require Import Machabstr.
Require Import Stackingproof.
Require Import Machblock4.
Require Import Values.
Require Import Globalenvs.
Require Import Mem.

A block semantics and its properties

Section RELSEM.

Variable ge : genv.

Inductive exec_instr :
        function → val → frame →
        code → regset → frame → mem →
        code → regset → frame → mem → Prop :=
  | exec_Mgetstack :
        ∀ f sp parent ofs ty dst c rs fr m v,
        get_slot fr ty (Int.signed ofs) v →
        exec_instr f sp parent
                        (Mgetstack ofs ty dst : : c) rs fr m
                        c (rs#dst ← v) fr m
  | exec_Msetstack :
      ∀ f sp parent src ofs ty c rs fr m fr',
       set_slot fr ty (Int.signed ofs) (rs src) fr' →
       exec_instr f sp parent
                        (Msetstack src ofs ty : : c) rs fr m
                        c rs fr' m
  | exec_Mgetparam :
        ∀ f sp parent ofs ty dst c rs fr m v,
        get_slot parent ty (Int.signed ofs) v →
        exec_instr f sp parent
                        (Mgetparam ofs ty dst : : c) rs fr m
                        c (rs#dst ← v) fr m
  | exec_Mop :
        ∀ f sp parent op args res c rs fr m v,
        eval_operation ge sp op rs##args = Some v →
        exec_instr f sp parent
                        (Mop op args res : : c) rs fr m
                        c (rs#res ← v) fr m
  | exec_Mload :
        ∀ f sp parent chunk addr args dst c rs fr m a v,
        eval_addressing ge sp addr rs##args = Some a →
        Mem.loadv chunk m a = Some v →
        exec_instr f sp parent
                        (Mload chunk addr args dst : : c) rs fr m
                        c (rs#dst ← v) fr m
  | exec_Mstore :
        ∀ f sp parent chunk addr args src c rs fr m m' a,
        eval_addressing ge sp addr rs##args = Some a →
        Mem.storev chunk m a (rs src) = Some m' →
        exec_instr f sp parent
                        (Mstore chunk addr args src : : c) rs fr m
                        c rs fr m'

with exec_block_end :
        function → val → frame →
        code → regset → frame → mem →
        code → regset → frame → mem → Prop :=
  | exec_Mlabel :
        ∀ f sp parent lbl c rs fr m,
        exec_block_end f sp parent
                        (Mlabel lbl : : c) rs fr m
                        c rs fr m
  | exec_Mcall :
        ∀ f sp parent sig ros c rs fr m f' rs' m',
        find_function ge ros rs = Some f' →
        exec_function f' fr rs m rs' m' →
        exec_block_end f sp parent
                        (Mcall sig ros : : c) rs fr m
                        c rs' fr m'
  | exec_Mgoto :
        ∀ f sp parent lbl c rs fr m c',
        find_label lbl f.(fn_code) = Some c' →
        exec_block_end f sp parent
                        (Mgoto lbl : : c) rs fr m
                        c' rs fr m
  | exec_Mcond_true :
        ∀ f sp parent cond args lbl c rs fr m c',
        eval_condition cond rs##args = Some true →
        find_label lbl f.(fn_code) = Some c' →
        exec_block_end f sp parent
                        (Mcond cond args lbl : : c) rs fr m
                        c' rs fr m
  | exec_Mcond_false :
        ∀ f sp parent cond args lbl c rs fr m,
        eval_condition cond rs##args = Some false →
        exec_block_end f sp parent
                        (Mcond cond args lbl : : c) rs fr m
                        c rs fr m

with exec_block_body :
        function → val → frame →
        code → regset → frame → mem →
        code → regset → frame → mem → Prop :=
  | exec_refl_bl :
        ∀ f sp parent c rs fr m,
        exec_block_body f sp parent c rs fr m c rs fr m
  | exec_one_bl :
        ∀ f sp parent c rs fr m c' rs' fr' m',
        exec_instr f sp parent c rs fr m c' rs' fr' m' →
        exec_block_body f sp parent c rs fr m c' rs' fr' m'
  | exec_trans_bl :
        ∀ f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 c3 rs3 fr3 m3,
        exec_block_body f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 →
        exec_block_body f sp parent c2 rs2 fr2 m2 c3 rs3 fr3 m3 →
        exec_block_body f sp parent c1 rs1 fr1 m1 c3 rs3 fr3 m3

with exec_instrs :
        function → val → frame →
        code → regset → frame → mem →
        code → regset → frame → mem → Prop :=
| exec_one_block :
        ∀ f sp parent c rs fr m c' rs' fr' m',
        exec_block_body f sp parent c rs fr m c' rs' fr' m' →
        exec_instrs f sp parent c rs fr m c' rs' fr' m'
| exec_add_block :
        ∀ f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 c3 rs3 fr3 m3 c4 rs4 fr4 m4,
        exec_block_body f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 →
        exec_block_end f sp parent c2 rs2 fr2 m2 c3 rs3 fr3 m3 →
        exec_instrs f sp parent c3 rs3 fr3 m3 c4 rs4 fr4 m4 →
        exec_instrs f sp parent c1 rs1 fr1 m1 c4 rs4 fr4 m4

with exec_function_body :
        function → frame → val → val →
        regset → mem → regset → mem → Prop :=
  | exec_funct_body :
        ∀ f parent link ra rs m rs' m1 m2 stk fr1 fr2 fr3 c,
        Mem.alloc m 0 f.(fn_stacksize) = (m1, stk) →
        set_slot (init_frame f) Tint 0 link fr1 →
        set_slot fr1 Tint 4 ra fr2 →
        exec_instrs f (Vptr stk (Int.repr (-f.(fn_framesize)))) parent
                        f.(fn_code) rs fr2 m1
                        (Mreturn : : c) rs' fr3 m2 →
        exec_function_body f parent link ra rs m rs' (Mem.free m2 stk)

with exec_function :
        function → frame → regset → mem → regset → mem → Prop :=
  | exec_funct :
        ∀ f parent rs m rs' m',
        (∀ link ra,
            Val.has_type link Tint →
            Val.has_type ra Tint →
            exec_function_body f parent link ra rs m rs' m') →
        exec_function f parent rs m rs' m'.

Scheme exec_instr_ind7 := Minimality for exec_instr Sort Prop
  with exec_block_end_ind7 := Minimality for exec_block_end Sort Prop
  with exec_block_body_ind7 := Minimality for exec_block_body Sort Prop
  with exec_instrs_ind7 := Minimality for exec_instrs Sort Prop
  with exec_function_body_ind7 := Minimality for exec_function_body Sort Prop
  with exec_function_ind7 := Minimality for exec_function Sort Prop.
```

```
End RELSEM.

Definition exec_program (p : program) (r : val) : Prop :=
  let ge := Genv.globalenv p in
  let m0 := Genv.init_mem p in
  ∃ b, ∃ f, ∃ rs, ∃ m,
  Genv.find_symbol ge p.(prog_main) = Some b ∧
  Genv.find_funct_ptr ge b = Some f ∧
  f.(fn_sig) = mksignature nil (Some Tint) ∧
  exec_function ge f empty_frame (Regmap.init Vundef) m0 rs m ∧
  rs (Conventions.loc_result f.(fn_sig)) = r.

Lemma exec_function_corr1 :
  ∀ ge f parent rs m rs' m',
  Machabstr.exec_function ge f parent rs m rs' m' →
  exec_function ge f parent rs m rs' m'.

Theorem eq1 :
  ∀ p r,
  Machabstr.exec_program p r → exec_program p r.

Lemma exec_function_corr2 :
  ∀ ge f fr rs m rs' m',
  exec_function ge f fr rs m rs' m' →
  Machabstr.exec_function ge f fr rs m rs' m'.

Theorem eq2 :
  ∀ p r,
  exec_program p r → Machabstr.exec_program p r.

Lemma exec_instr_length :
∀ ge f sp parent c rs fr m c' rs' fr' m',
exec_instr ge f sp parent c rs fr m c' rs' fr' m' →
List.length c = (1 + List.length c')%nat.

Lemma exec_block_body_length :
∀ ge f sp parent c rs fr m c' rs' fr' m',
exec_block_body ge f sp parent c rs fr m c' rs' fr' m' →
(List.length c' ≤ List.length c)%nat.

Lemma no_move_aux :
∀ ge f sp parent c rs fr m c' rs' fr' m',
exec_block_body ge f sp parent c rs fr m c' rs' fr' m' →
List.length c = List.length c' → rs = rs' ∧ fr = fr' ∧ m = m'.

Theorem no_move :
∀ ge f sp parent c rs fr m rs' fr' m',
exec_block_body ge f sp parent c rs fr m c rs' fr' m' →
rs = rs' ∧ fr = fr' ∧ m = m'.

Inductive exec_block_body_normalized :
      genv → function → val → frame →
      code → regset → frame → mem →
      code → regset → frame → mem → Prop :=
  | exec_refl_norm :
      ∀ ge f sp parent c rs fr m,
      exec_block_body_normalized ge f sp parent c rs fr m c rs fr m
  | exec_one_norm :
      ∀ ge f sp parent c rs fr m c' rs' fr' m' c'' rs'' fr'' m'',
      exec_instr ge f sp parent c rs fr m c' rs' fr' m' →
      exec_block_body_normalized ge f sp parent c' rs' fr' m' c'' rs'' fr'' m'' →
      exec_block_body_normalized ge f sp parent c rs fr m c'' rs'' fr'' m''.

Lemma trans_norm :
∀ ge f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 c3 rs3 fr3 m3,
exec_block_body_normalized ge f sp parent c1 rs1 fr1 m1 c2 rs2 fr2 m2 →
exec_block_body_normalized ge f sp parent c2 rs2 fr2 m2 c3 rs3 fr3 m3 →
exec_block_body_normalized ge f sp parent c1 rs1 fr1 m1 c3 rs3 fr3 m3.

Lemma normalization :
∀ ge f sp parent c1 c2 rs fr m rs' fr' m',
exec_block_body ge f sp parent (c1 ++ c2) rs fr m c2 rs' fr' m' →
exec_block_body_normalized ge f sp parent (c1 ++ c2) rs fr m c2 rs' fr' m'.

Lemma normalization2 :
∀ ge f sp parent c rs fr m c' rs' fr' m',
exec_block_body ge f sp parent c rs fr m c' rs' fr' m' →
exec_block_body_normalized ge f sp parent c rs fr m c' rs' fr' m'.

Lemma anti_normalization :
∀ ge f sp parent c c' rs fr m rs' fr' m',
exec_block_body_normalized ge f sp parent c rs fr m c' rs' fr' m' →
exec_block_body ge f sp parent c rs fr m c' rs' fr' m'.

Lemma list_size :
∀ l1 l2 : list instruction,
l1 = l2 → length l1 = length l2.

Lemma list_plus_size :
∀ l1 l2 : list instruction,
length (l1 ++ l2) = (length l1 + length l2)%nat.

Lemma list_diff :
∀ (l2 l : list instruction) (a : instruction),
a : : l2 ++ l = l → False.

Theorem middle_state :
∀ ge p sp parent a c1 c2 rs fr m rs' fr' m',
exec_block_body ge p sp parent (a : : c1 ++ c2) rs fr m c2 rs' fr' m' →
∃ c'', ∃ rs'', ∃ fr'', ∃ m'',
exec_instr ge p sp parent (a : : c1 ++ c2) rs fr m c'' rs'' fr'' m'' ∧
exec_block_body ge p sp parent c'' rs'' fr'' m'' c2 rs' fr' m'.

Theorem one_step :
∀ ge f sp parent a c rs fr m c' rs' fr' m',
exec_instr ge f sp parent (a : : c) rs fr m c' rs' fr' m' →
c = c'.

Parameter schedule_code : code → option code.

Formalisation of the symbolic evaluation and its properties

Inductive resource : Set :=
| Reg : mreg → resource
| Mem : resource
| Stack : resource.

Lemma mreg_eq : ∀ (r1 r2 : mreg), {r1 = r2} + {r1 ≠ r2}.

Lemma resource_eq : ∀ (r1 r2 : resource), {r1 = r2} + {r1 ≠ r2}.

Module R_equality.
Definition t := resource.
Definition eq := resource_eq.
End R_equality.

Module Rmap := EMap(R_equality).

Syntax

Inductive expression : Set :=
| Eimp : expression
| Ebase : resource → expression
| Eop : operation → expression_list → expression
| Eload : memory_chunk → addressing → expression_list → expression → expression
| Estore : expression → memory_chunk → addressing → expression_list → expression → expression
| Egetstack : int → typ → expression → expression
| Esetstack : expression → expression → int → typ → expression
| Egetparam : int → typ → expression
with expression_list : Set :=
| Enil : expression_list
| Econs : expression → expression_list → expression_list.

Scheme expression_ind3 := Minimality for expression Sort Prop
      with expression_list_ind3 := Minimality for expression_list Sort Prop.

Semantics

Inductive sem_value :
genv → val → frame → regset → frame → mem → expression → val → Prop :=
  | Sgetstack :
      ∀ ge sp parent rs fr m stack_exp ty ofs v fr',
      sem_frame ge sp parent rs fr m stack_exp fr' →
      get_slot fr' ty (Int.signed ofs) v →
      sem_value ge sp parent rs fr m (Egetstack ofs ty stack_exp) v

  | Sgetparam :
      ∀ ge sp parent rs fr m ty ofs v,
      get_slot parent ty (Int.signed ofs) v →
      sem_value ge sp parent rs fr m (Egetparam ofs ty) v

  | Sload :
      ∀ ge sp parent rs fr m mem_exp addr chunk args a v m' lv,
      sem_mem ge sp parent rs fr m mem_exp m' →
      sem_val_list ge sp parent rs fr m args lv →
      eval_addressing ge sp addr lv = Some a →
      Mem.loadv chunk m' a = Some v →
      sem_value ge sp parent rs fr m (Eload chunk addr args mem_exp) v

  | Sop :
      ∀ ge sp parent rs fr m op args v lv,
      sem_val_list ge sp parent rs fr m args lv →
      eval_operation ge sp op lv = Some v →
      sem_value ge sp parent rs fr m (Eop op args) v

  | Sbase_reg :
      ∀ ge sp parent rs fr m r,
      sem_value ge sp parent rs fr m (Ebase (Reg r)) (rs r)

with sem_frame :
genv → val → frame → regset → frame → mem → expression → frame → Prop :=
  | Ssetstack :
      ∀ ge sp parent rs fr m stack_exp val_exp ty ofs fr' v fr'',
      sem_frame ge sp parent rs fr m stack_exp fr' →
      sem_value ge sp parent rs fr m val_exp v →
      set_slot fr' ty (Int.signed ofs) v fr'' →
      sem_frame ge sp parent rs fr m (Esetstack stack_exp val_exp ofs ty) fr''

  | Sbase_frame :
      ∀ ge sp parent rs fr m,
      sem_frame ge sp parent rs fr m (Ebase Stack) fr

with sem_mem :
genv → val → frame → regset → frame → mem → expression → mem → Prop :=
  | Sstore :
      ∀ ge sp parent rs fr m mem_exp val_exp m'' addr v a m' chunk args lv,
      sem_mem ge sp parent rs fr m mem_exp m' →
      sem_value ge sp parent rs fr m val_exp v →
      sem_val_list ge sp parent rs fr m args lv →
      eval_addressing ge sp addr lv = Some a →
      Mem.storev chunk m' a v = Some m'' →
      sem_mem ge sp parent rs fr m (Estore mem_exp chunk addr args val_exp) m''
  | Sbase_mem :
      ∀ ge sp parent rs fr m,
      sem_mem ge sp parent rs fr m (Ebase Mem) m

with sem_val_list :
genv → val → frame → regset → frame → mem → expression_list → list val → Prop :=
  | Snil :
      ∀ ge sp parent rs fr m,
      sem_val_list ge sp parent rs fr m (Enil) nil

  |Scons :
      ∀ ge sp parent rs fr m e v l lv,
      sem_value ge sp parent rs fr m e v →
      sem_val_list ge sp parent rs fr m l lv →
      sem_val_list ge sp parent rs fr m (Econs e l) (cons v lv).

Scheme sem_value_ind2 := Minimality for sem_value Sort Prop
      with sem_frame_ind2 := Minimality for sem_frame Sort Prop
      with sem_mem_ind2 := Minimality for sem_mem Sort Prop
      with sem_val_list_ind2 := Minimality for sem_val_list Sort Prop.

Hint Resolve Sgetstack : vali.
```

```
Hint Resolve Sgetparam : vali.
Hint Resolve Sload : vali.
Hint Resolve Sop : vali.
Hint Resolve Sbase_reg : vali.
Hint Resolve Ssetstack : vali.
Hint Resolve Sbase_frame : vali.
Hint Resolve Sstore : vali.
Hint Resolve Sbase_mem : vali.
Hint Resolve Snil : vali.
Hint Resolve Scons : vali.

Definition forest : Set := Rmap.t (expression).

Inductive sem_regset :
genv → val → frame → regset → frame → mem → forest → regset → Prop :=
  | Sregset :
      ∀ ge sp parent rs fr m f rs',
      (∀ x, sem_value ge sp parent rs fr m (f (Reg x)) (rs' (x))) →
      sem_regset ge sp parent rs fr m f rs'.

Inductive sem :
genv → val → frame → regset → frame → mem → forest → regset → frame → mem → Prop :=
  | Sem :
      ∀ ge sp parent rs fr m f rs' fr' m',
      sem_regset ge sp parent rs fr m f rs' →
      sem_frame ge sp parent rs fr m (f Stack) fr' →
      sem_mem ge sp parent rs fr m (f Mem) m' →
      sem ge sp parent rs fr m f rs' fr' m'.
```

a function to test if two abstract states are equals and its proof of correctness

```
Lemma comparison_eq : ∀ (x y : comparison), {x = y} + {x ≠ y}.

Lemma condition_eq : ∀ (x y : condition), {x = y} + {x ≠ y}.

Lemma operation_eq : ∀ (x y : operation), {x = y} + {x ≠ y}.

Lemma addressing_eq : ∀ (x y : addressing), {x = y} + {x ≠ y}.

Lemma memory_chunk_eq : ∀ (x y : memory_chunk), {x = y} + {x ≠ y}.

Lemma typ_eq : ∀ (x y : typ), {x = y} + {x ≠ y}.

Lemma list_typ_eq : ∀ (x y : list typ), {x = y} + {x ≠ y}.

Lemma option_typ_eq : ∀ (x y : option typ), {x = y} + {x ≠ y}.

Lemma signature_eq : ∀ (x y : signature), {x = y} + {x ≠ y}.

Lemma list_operation_eq : ∀ (x y : list operation), {x = y} + {x ≠ y}.

Lemma list_mreg_eq : ∀ (x y : list mreg), {x = y} + {x ≠ y}.

Lemma mreg_plus_ident_eq : ∀ (x y : mreg + ident), {x = y} + {x ≠ y}.

Lemma instruction_eq : ∀ (x y : instruction), {x = y} + {x ≠ y}.

Fixpoint beq_expression (e1 e2 : expression) {struct e1} : bool :=
  match e1, e2 with
  | Eimp , Eimp ⇒ false
  | Ebase r1, Ebase r2 ⇒ if resource_eq r1 r2 then true else false
  | Eop op1 el1, Eop op2 el2 ⇒
      if operation_eq op1 op2 then beq_expression_list el1 el2 else false
  | Eload chk1 addr1 el1 e1, Eload chk2 addr2 el2 e2 ⇒
      if memory_chunk_eq chk1 chk2
      then if addressing_eq addr1 addr2
      then if beq_expression_list el1 el2
      then beq_expression e1 e2 else false else false else false
  | Estore m1 chk1 addr1 el1 e1, Estore m2 chk2 addr2 el2 e2⇒
      if memory_chunk_eq chk1 chk2
      then if addressing_eq addr1 addr2
      then if beq_expression_list el1 el2
      then if beq_expression m1 m2
      then beq_expression e1 e2 else false else false else false else false
  | Egetstack i1 t1 e1, Egetstack i2 t2 e2 ⇒
      if Int.eq_dec i1 i2
      then if typ_eq t1 t2
      then if beq_expression e1 e2
      then true
      else false
      else false
      else false
  | Esetstack f1 e1 i1 t1, Esetstack f2 e2 i2 t2 ⇒
      if Int.eq_dec i1 i2
      then if typ_eq t1 t2
      then if beq_expression e1 e2
      then if beq_expression f1 f2 then true
      else false else false else false else false
  | Egetparam i1 t1, Egetparam i2 t2 ⇒
      if Int.eq_dec i1 i2
      then if typ_eq t1 t2 then true
      else false else false
  | _, _ ⇒ false
  end
with beq_expression_list (el1 el2 : expression_list) {struct el1} : bool :=
  match el1, el2 with
  | Enil, Enil ⇒ true
  | Econs e1 t1, Econs e2 t2 ⇒ beq_expression e1 e2 && beq_expression_list t1 t2
  | _, _ ⇒ false
  end.

Scheme expression_ind2 := Induction for expression Sort Prop
    with expression_list_ind2 := Induction for expression_list Sort Prop.

Lemma beq_expression_correct :
  ∀ e1 e2, beq_expression e1 e2 = true → e1 = e2.

Hint Resolve beq_expression_correct.

Definition check (fa fb : forest) :=
beq_expression (fa (Reg R3)) (fb (Reg R3)) &&
beq_expression (fa (Reg R4)) (fb (Reg R4)) &&
beq_expression (fa (Reg R5)) (fb (Reg R5)) &&
beq_expression (fa (Reg R6)) (fb (Reg R6)) &&
beq_expression (fa (Reg R7)) (fb (Reg R7)) &&
beq_expression (fa (Reg R8)) (fb (Reg R8)) &&
beq_expression (fa (Reg R9)) (fb (Reg R9)) &&
beq_expression (fa (Reg R10)) (fb (Reg R10)) &&
beq_expression (fa (Reg R13)) (fb (Reg R13)) &&
beq_expression (fa (Reg R14)) (fb (Reg R14)) &&
beq_expression (fa (Reg R15)) (fb (Reg R15)) &&
beq_expression (fa (Reg R16)) (fb (Reg R16)) &&
beq_expression (fa (Reg R17)) (fb (Reg R17)) &&
beq_expression (fa (Reg R18)) (fb (Reg R18)) &&
beq_expression (fa (Reg R19)) (fb (Reg R19)) &&
beq_expression (fa (Reg R20)) (fb (Reg R20)) &&
beq_expression (fa (Reg R21)) (fb (Reg R21)) &&
beq_expression (fa (Reg R22)) (fb (Reg R22)) &&
beq_expression (fa (Reg R23)) (fb (Reg R23)) &&
beq_expression (fa (Reg R24)) (fb (Reg R24)) &&
beq_expression (fa (Reg R25)) (fb (Reg R25)) &&
beq_expression (fa (Reg R26)) (fb (Reg R26)) &&
beq_expression (fa (Reg R27)) (fb (Reg R27)) &&
beq_expression (fa (Reg R28)) (fb (Reg R28)) &&
beq_expression (fa (Reg R29)) (fb (Reg R29)) &&
beq_expression (fa (Reg R30)) (fb (Reg R30)) &&
beq_expression (fa (Reg R31)) (fb (Reg R31)) &&
beq_expression (fa (Reg F1)) (fb (Reg F1)) &&
beq_expression (fa (Reg F2)) (fb (Reg F2)) &&
beq_expression (fa (Reg F3)) (fb (Reg F3)) &&
beq_expression (fa (Reg F4)) (fb (Reg F4)) &&
beq_expression (fa (Reg F5)) (fb (Reg F5)) &&
beq_expression (fa (Reg F6)) (fb (Reg F6)) &&
beq_expression (fa (Reg F7)) (fb (Reg F7)) &&
beq_expression (fa (Reg F8)) (fb (Reg F8)) &&
beq_expression (fa (Reg F9)) (fb (Reg F9)) &&
beq_expression (fa (Reg F10)) (fb (Reg F10)) &&
beq_expression (fa (Reg F14)) (fb (Reg F14)) &&
beq_expression (fa (Reg F15)) (fb (Reg F15)) &&
beq_expression (fa (Reg F16)) (fb (Reg F16)) &&
beq_expression (fa (Reg F17)) (fb (Reg F17)) &&
beq_expression (fa (Reg F18)) (fb (Reg F18)) &&
beq_expression (fa (Reg F19)) (fb (Reg F19)) &&
beq_expression (fa (Reg F20)) (fb (Reg F20)) &&
beq_expression (fa (Reg F21)) (fb (Reg F21)) &&
beq_expression (fa (Reg F22)) (fb (Reg F22)) &&
beq_expression (fa (Reg F23)) (fb (Reg F23)) &&
beq_expression (fa (Reg F24)) (fb (Reg F24)) &&
beq_expression (fa (Reg F25)) (fb (Reg F25)) &&
beq_expression (fa (Reg F26)) (fb (Reg F26)) &&
beq_expression (fa (Reg F27)) (fb (Reg F27)) &&
beq_expression (fa (Reg F28)) (fb (Reg F28)) &&
beq_expression (fa (Reg F29)) (fb (Reg F29)) &&
beq_expression (fa (Reg F30)) (fb (Reg F30)) &&
beq_expression (fa (Reg F31)) (fb (Reg F31)) &&
beq_expression (fa (Reg IT1)) (fb (Reg IT1)) &&
beq_expression (fa (Reg IT2)) (fb (Reg IT2)) &&
beq_expression (fa (Reg IT3)) (fb (Reg IT3)) &&
beq_expression (fa (Reg FT1)) (fb (Reg FT1)) &&
beq_expression (fa (Reg FT2)) (fb (Reg FT2)) &&
beq_expression (fa (Reg FT3)) (fb (Reg FT3)) &&
beq_expression (fa Mem) (fb Mem) &&
beq_expression (fa Stack) (fb Stack).

Lemma essai : ∀ (a b : bool), a && b = true → a = true ∧ b = true.

Lemma check_correct : ∀ (fa fb : forest) (x : resource),
check fa fb = true → fa x = fb x.

Lemma forest_extensionnality :
∀ (fa fb : forest),
(∀ x, fa x = fb x) → fa = fb.

Lemma check_correct2 : ∀ (fa fb : forest),
check fa fb = true → fa = fb.

Notation "a # b ← c" := (Rmap.set b c a) (at level 1, b at next level).
```

The empty abstract state

```
Definition empty : forest := fun r ⇒ Ebase r.
```

A proof of funtionanlity for abstract states denotation

```
Lemma fonct :
  ∀ ge tge sp parent rs fr m f,
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  ∀ v v' fv fv' mv mv',
  (sem_value ge sp parent rs fr m f v ∧ sem_value tge sp parent rs fr m f v' → v = v') ∧
  (sem_frame ge sp parent rs fr m f fv ∧ sem_frame tge sp parent rs fr m f fv' → fv = fv') ∧
  (sem_mem ge sp parent rs fr m f mv ∧ sem_mem tge sp parent rs fr m f mv' → mv = mv').

Lemma fonct_value :
  ∀ ge tge sp parent rs fr m f v v',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  sem_value ge sp parent rs fr m f v →
  sem_value tge sp parent rs fr m f v' →
  v = v'.

Lemma s_fonct_value :
  ∀ ge sp parent rs fr m f v v',
  sem_value ge sp parent rs fr m f v →
  sem_value ge sp parent rs fr m f v' →
  v = v'.

Lemma fonct_frame :
  ∀ ge tge sp parent rs fr m f v v',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  sem_frame ge sp parent rs fr m f v →
  sem_frame tge sp parent rs fr m f v' →
```

```
  v = v'.

Lemma s_fonct_frame :
  ∀ ge sp parent rs fr m f v v',
  sem_frame ge sp parent rs fr m f v →
  sem_frame ge sp parent rs fr m f v' →
  v = v'.

Lemma fonct_mem :
  ∀ ge tge sp parent rs fr m f v v',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  sem_mem ge sp parent rs fr m f v →
  sem_mem tge sp parent rs fr m f v' →
  v = v'.

Lemma s_fonct_mem :
  ∀ ge sp parent rs fr m f v v',
  sem_mem ge sp parent rs fr m f v →
  sem_mem ge sp parent rs fr m f v' →
  v = v'.

Lemma fonct_regset :
  ∀ ge tge sp parent rs fr m f v v',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  sem_regset ge sp parent rs fr m f v →
  sem_regset tge sp parent rs fr m f v' →
  v = v'.

Hint Resolve fonct_value : vali.
Hint Resolve fonct_frame : vali.
Hint Resolve fonct_mem : vali.
Hint Resolve fonct_regset : vali.

Lemma font_sem :
  ∀ ge tge sp parent f rs fr m rs' fr' m' rs'' fr'' m'',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  sem ge sp parent rs fr m f rs' fr' m' →
  sem tge sp parent rs fr m f rs'' fr'' m'' →
  rs' = rs'' ∧ fr' = fr'' ∧ m' = m''.

Lemma s_fonct_sem :
  ∀ ge sp parent f rs fr m rs' fr' m' rs'' fr'' m'',
  sem ge sp parent rs fr m f rs' fr' m' →
  sem ge sp parent rs fr m f rs'' fr'' m'' →
  rs' = rs'' ∧ fr' = fr'' ∧ m' = m''.

The abstraction function and its proof of correctness

Fixpoint list_translation (l : list mreg) (f : forest) {struct l} : expression_list :=
match l with
| nil ⇒ Enil
| cons i l ⇒ Econs (Rmap.get (Reg i) f) (list_translation l f)
end.

Definition update (f : forest) (i : instruction) : forest :=
match i with
    | Mgetstack i0 t r ⇒
        Rmap.set (Reg r) (Egetstack i0 t (Rmap.get Stack f)) f
    | Msetstack r i0 t ⇒
        Rmap.set Stack (Esetstack (Rmap.get Stack f) (Rmap.get (Reg r) f) i0 t) f
    | Mgetparam i0 t r ⇒
        Rmap.set (Reg r) (Egetparam i0 t) f
    | Mop op rl r ⇒
        Rmap.set (Reg r) (Eop op (list_translation rl f)) f
    | Mload chunk addr rl r ⇒
        Rmap.set (Reg r) (Eload chunk addr (list_translation rl f) (Rmap.get Mem f)) f
    | Mstore chunk addr rl r ⇒
        Rmap.set Mem (Estore (Rmap.get Mem f) chunk addr (list_translation rl f) (Rmap.get (Reg r) f)) f
    | Mcall sign id ⇒ f
    | Mlabel lbl ⇒ f
    | Mgoto lbl ⇒ f
    | Mcond cond rl lbl ⇒ f
    | Mreturn ⇒ f
end.

Lemma map0 :
  ∀ r,
  empty r = Ebase r.

Lemma map1 :
  ∀ w dst dst',
  dst ≠ dst' →
  (empty # dst ← w) dst' = Ebase dst'.

Lemma genmap1 :
  ∀ (f : forest) w dst dst',
  dst ≠ dst' →
  (f # dst ← w) dst' = f dst'.

Lemma map2 :
  ∀ (v : expression) x rs,
  (rs # x ← v) x = v.

Lemma map3 :
  ∀ (v : val) (x : mreg) rs,
  Regmap.set x v rs x= v.

Lemma map4 :
  ∀ x,
  Rmap.get x empty = Ebase x.

Lemma map5 :
  ∀ x y (v : val) rs,
  x ≠ y →
  Regmap.set y v rs x = rs x.

Lemma tri1 :
  ∀ x y,
  Reg x ≠ Reg y → x ≠ y.

Hint Rewrite → map0 : vali.
Hint Rewrite → map1 : vali.
Hint Rewrite → map2 : vali.
Hint Rewrite → map3 : vali.
Hint Rewrite → map4 : vali.
Hint Rewrite → map5 : vali.

Lemma gen_list_base :
  ∀ ge sp parent l rs rs0 fr0 m0 exps,
  (∀ x, sem_value ge sp parent rs0 fr0 m0 (exps (Reg x)) (rs x)) →
  sem_val_list ge sp parent rs0 fr0 m0 (list_translation l exps) rs ## l.

Lemma h2 :
∀ ge f sp parent i c exps rs fr m rs0 fr0 m0 rs' fr' m',
exec_instr ge f sp parent (i : : c) rs fr m c rs' fr' m' →
sem ge sp parent rs0 fr0 m0 exps rs fr m →
sem ge sp parent rs0 fr0 m0 (update exps i) rs' fr' m'.

Lemma one_schanged_reg :
∀ ge sp parent rs rs' fr m y w,
(∀ x : mreg,
        sem_value ge sp parent rs fr m (empty # (Reg y) ← w (Reg x)) (rs' x)) →
(∀ x, rs' x= (Regmap.set y (rs' y) rs) x).

Lemma gen_one_changed_reg :
  ∀ ge sp parent rs0 fr0 m0 rs fr m rs' y w exps,
  sem ge sp parent rs0 fr0 m0 exps rs fr m →
  (∀ x,
      sem_value ge sp parent rs0 fr0 m0
        (exps # (Reg y) ← w (Reg x))
        (rs' x)) →
  (∀ x, rs' x= (Regmap.set y (rs' y) rs) x).


Lemma regset_unchanged_Stack :
∀ ge sp parent rs fr m rs' w,
(∀ x : mreg,
      sem_value ge sp parent rs fr m (empty # Stack ← w (Reg x)) (rs' x)) →
(∀ x : mreg, rs x = rs' x).

Lemma gen_regset_unchanged_Stack :
  ∀ ge sp parent rs fr m rs0 fr0 m0 rs' w f,
  sem ge sp parent rs0 fr0 m0 f rs fr m →
  (∀ x : mreg,
  sem_value ge sp parent rs0 fr0 m0 (f # Stack ← w (Reg x)) (rs' x)) →
  (∀ x : mreg, rs x = rs' x).

Lemma regset_unchanged_Mem :
∀ ge sp parent rs fr m rs' w,
(∀ x : mreg,
      sem_value ge sp parent rs fr m (empty # Mem ← w (Reg x)) (rs' x)) →
(∀ x : mreg, rs x = rs' x).

Lemma gen_regset_unchanged_Mem :
  ∀ ge sp parent rs fr m rs0 fr0 m0 rs' w f,
  sem ge sp parent rs0 fr0 m0 f rs fr m →
  (∀ x : mreg,
  sem_value ge sp parent rs0 fr0 m0 (f # Mem ← w (Reg x)) (rs' x)) →
  (∀ x : mreg, rs x = rs' x).

Lemma list_translation_base :
  ∀ args ge sp parent rs fr m,
  sem_val_list ge sp parent rs fr m (list_translation args empty) rs##args.

Lemma gen_list_translation_det :
  ∀ ge sp parent rs fr m l l1 l2 f,
  sem_val_list ge sp parent rs fr m (list_translation l f) l1 →
  sem_val_list ge sp parent rs fr m (list_translation l f) l2 →
  l1 = l2.

Lemma to_exten :
∀ ge sp parent rs fr m rs',
 (∀ x : mreg,
      sem_value ge sp parent rs fr m (empty (Reg x)) (rs' x)) →
      (∀ x, rs x = rs' x).

Lemma empty_denotation :
∀ ge sp parent rs fr m rs' fr' m',
sem ge sp parent rs fr m empty rs' fr' m' →
rs = rs' ∧ fr = fr' ∧ m = m'.

Definition nbranchset (i : instruction) : bool :=
match i with
| Msetstack a b c ⇒ true
| Mgetstack a b c ⇒ true
| Mgetparam a b c ⇒ true
| Mop a b c ⇒ true
| Mload a b c d ⇒ true
| Mstore a b c d ⇒ true
| _ ⇒ false
end.

Fixpoint lnbranchset (l : list instruction) : bool :=
match l with
| Msetstack a b c : : k ⇒ lnbranchset k
| Mgetstack a b c : : k ⇒ lnbranchset k
| Mgetparam a b c : : k ⇒ lnbranchset k
| Mop a b c : : k ⇒ lnbranchset k
| Mload a b c d : : k ⇒ lnbranchset k
| Mstore a b c d : : k ⇒ lnbranchset k
| nil ⇒ true
| _ ⇒ false
end.

Lemma lnbranchset_prop :
  ∀ x a,
  lnbranchset (a : : x) = true → nbranchset a = true ∧ lnbranchset x = true.

Lemma hi2 :
  ∀ ge f sp parent i c rs fr m c0 rs0 fr0 m0 rs' fr' m',
```

```
  nbranchset i = true →
  exec_block_body ge f sp parent (c0 ++ (i : : c)) rs0 fr0 m0 (i : : c) rs fr m →
  sem ge sp parent rs fr m (update empty i) rs' fr' m' →
  exec_block_body ge f sp parent (c0 ++ (i : : c)) rs0 fr0 m0 c rs' fr' m'.

Lemma hi2v2 :
  ∀ ge fu sp parent f i w rs0 fr0 m0 rs fr m rs' fr' m',
  nbranchset i = true →
  sem ge sp parent rs0 fr0 m0 f rs fr m →
  sem ge sp parent rs0 fr0 m0 (update f i) rs' fr' m' →
  exec_instr ge fu sp parent (i : : w) rs fr m w rs' fr' m'.

Fixpoint abstract_block2 (f : forest) (c : code) {struct c} : forest :=
  match c with
  | nil ⇒ f
  | cons (Mgoto lbl) l ⇒ f
  | cons (Mcall sig n) l ⇒ f
  | cons (Mlabel lbl) l ⇒ f
  | cons (Mcond cmp rl lbl) l ⇒ f
  | cons (Mreturn) l ⇒ f
  | i : : l ⇒ abstract_block2 (update f i) l
  end.

Lemma little_com :
abstract_block2 empty nil = empty.



Hypothesis weakness :
  ∀ ge tge f tf sp parent c1 rs0 fr0 m0 tc1 rs fr m c2 tc2,
  exec_block_body ge f sp parent (c1 ++ c2) rs fr m c2 rs0 fr0 m0 →
  check (abstract_block2 empty c1) (abstract_block2 empty tc1) = true →
  ∃ rs', ∃ fr', ∃ m',
  exec_block_body tge tf sp parent (tc1 ++ tc2) rs fr m tc2 rs' fr' m'.

Lemma fix_sem :
  ∀ ge sp parent rs fr m,
  sem ge sp parent rs fr m empty rs fr m.

Lemma hgen :
  ∀ ge f sp parent c rs fr m rs0 fr0 m0 c' rs' fr' m' exps,
  exec_block_body ge f sp parent (c ++ c') rs fr m c' rs' fr' m' →
  sem ge sp parent rs0 fr0 m0 exps rs fr m →
  sem ge sp parent rs0 fr0 m0 (abstract_block2 exps c) rs' fr' m'.

Lemma abstract_block2_correct :
  ∀ ge f sp parent c rs fr m c' rs' fr' m',
  exec_block_body ge f sp parent (c ++ c') rs fr m c' rs' fr' m' →
  sem ge sp parent rs fr m (abstract_block2 empty c) rs' fr' m'.

The proof of semantics preservation for blocks

Lemma abstraction_check :
  ∀ ge tge f tf sp parent c1 c2 tc1 tc2 rs fr m rs' fr' m',
  (∀ sp op vl, eval_operation ge sp op vl = eval_operation tge sp op vl) →
  (∀ sp addr vl, eval_addressing ge sp addr vl = eval_addressing tge sp addr vl) →
  check (abstract_block2 empty c1) (abstract_block2 empty tc1) = true →
  exec_block_body ge f sp parent (c1 ++ c2) rs fr m c2 rs' fr' m' →
  exec_block_body tge tf sp parent (tc1 ++ tc2) rs fr m tc2 rs' fr' m'.

Definition check_glue h1 h2 :=
match h1 with
|error ⇒ match h2 with
                |error ⇒ true
                |value x2 ⇒ false
                end
|value x1 ⇒ match h2 with
                  |error ⇒ false
                  |value x2 ⇒ if instruction_eq x1 x2 then true else false
                  end
end.

The validation function for the whole code

Lemma glue_correct :
∀ x y,
check_glue x y = true → x = y.

Fixpoint validate (c tc : code) (f tf : forest) {struct c} : bool :=
  match c with
  | nil ⇒ match tc with
                | nil ⇒ true
                | _ ⇒ false
                end
  | cons i l ⇒
                match tc with
                | nil ⇒ false
                | cons i' l' ⇒
                    if nbranchset i && nbranchset i' then validate l l' (update f i) (update tf i')
                    else
                    check_glue (value i) (value i') && check f tf && validate l l' empty empty
                end
  end.

Form static to dynamic

Fixpoint to_next_block (c : code) {struct c} : code :=
  match c with
  | nil ⇒ nil
  | cons (Mgoto lbl) l ⇒ l
  | cons (Mcall sig n) l ⇒ l
  | cons (Mlabel lbl) l ⇒ l
  | cons (Mcond cmp rl lbl) l ⇒ l
  | cons (Mreturn) l ⇒ l
  | i : : l ⇒ to_next_block l
  end.

Lemma to_next_block_prop :
  ∀ a c,
  nbranchset a = true → to_next_block (a : : c) = to_next_block c.

Lemma to_next_block_prop1 :
  ∀ x0 c x1,
  c = x0 ++ Mreturn : : x1 →
  lnbranchset x0 = true →
  to_next_block c = x1.

Lemma to_next_block_prop2 :
  ∀ x k,
  lnbranchset x = true → to_next_block (x ++ k) = to_next_block k.

Lemma validate_propagation1_aux :
∀ c tc f tf,
validate c tc f tf = true →
validate (to_next_block c) (to_next_block tc) empty empty = true.

Lemma validate_propagation1 :
∀ c tc,
validate c tc empty empty = true →
validate (to_next_block c) (to_next_block tc) empty empty = true.

Lemma find_label_prop :
  ∀ i c lbl,
  nbranchset i = true → find_label lbl (i : : c) = find_label lbl c.

Lemma validate_propagation2_aux :
∀ c tc c' tc' f tf lbl,
validate c tc f tf = true →
find_label lbl c = Some c' →
find_label lbl tc = Some tc' →
validate c' tc' empty empty = true.

Lemma validate_propagation2 :
∀ c tc c' tc' lbl,
validate c tc empty empty = true →
find_label lbl c = Some c' →
find_label lbl tc = Some tc' →
validate c' tc' empty empty = true.

Lemma find_label_validated_aux :
  ∀ c tc k1 lbl f tf,
  validate c tc f tf = true →
  find_label lbl c = Some k1 →
  ∃ k2,
  find_label lbl tc = Some k2.

Lemma find_label_validated :
  ∀ f tf,
  validate (fn_code f) (fn_code tf) empty empty = true →
  ∀ lbl k1, find_label lbl (fn_code f) = Some k1 →
  ∃ k2,
  find_label lbl (fn_code tf) = Some k2.

Some rewriting lemmas

Lemma decomposition2 :
  ∀ ge f sp parent c rs fr m c' rs' fr' m' i l,
  exec_block_body ge f sp parent c rs fr m c' rs' fr' m' →
  c' = i : : l →
  nbranchset i = false →
  ∃ c1,
  c = c1 ++ c' ∧ lnbranchset c1 = true.

Lemma get_infos_aux_gen3 :
  ∀ tc x c i l f tf,
  validate c tc f tf = true →
  c = x ++ (i : : l) → lnbranchset x = true → nbranchset i = false →
  ∃ tc1,
  check (abstract_block2 f x) (abstract_block2 tf tc1) = true ∧
  lnbranchset tc1 = true ∧ ∃ tc', tc = tc1 ++ ( i : : tc').

Lemma get_infos_gen3 :
  ∀ tc x c i l,
  validate c tc empty empty = true →
  c = x ++ (i : : l) → lnbranchset x = true → nbranchset i = false →
  ∃ tc1,
  check (abstract_block2 empty x) (abstract_block2 empty tc1) = true ∧
  lnbranchset tc1 = true ∧ ∃ tc', tc = tc1 ++ ( i : : tc').

Lemma end_of_line :
  ∀ x1,
  validate (Mreturn : : nil) x1 empty empty = true →
  x1 = Mreturn : : nil.


Scheme exec_spec_instrs_ind := Minimality for exec_instrs Sort Prop
      with exec_spec_ind_function_ind := Minimality for exec_function Sort Prop.

Lemma dec1 :
  ∀ ge f sp parent c c' rs fr m rs' fr' m',
  exec_block_end ge f sp parent c rs fr m c' rs' fr' m' →
  ∃ i, ∃ l, c = i : : l ∧ nbranchset i = false.

Scheme glue_block_end_ind := Minimality for exec_block_end Sort Prop
   with glue_function_ind := Minimality for exec_function Sort Prop.

Scheme exec_instrs_ind2 := Minimality for exec_instrs Sort Prop
      with exec_function_ind2 := Minimality for exec_function Sort Prop.

Definition transf_code (c : code) : option code :=
  match schedule_code c with
  | None ⇒ None
  | Some tc ⇒ if validate c tc empty empty then Some tc else None
  end.

Definition transf_function (f : function) : option function :=
  match transf_code f.(fn_code) with
  | None ⇒ None
  | Some tc ⇒
      Some (mkfunction f.(fn_sig) tc f.(fn_stacksize) f.(fn_framesize))
  end.

Lemma talk :
  ∀ f tf,
```

```
  transf_function f = Some tf →
  validate (fn_code f) (fn_code tf) empty empty = true.

Definition transf_program (p : program) : option program :=
  transform_partial_program transf_function p.

Lemma env_doesnt_matter1 :
  ∀ p tp,
  transf_program p = Some tp →
  ∀ sp op vl,
  eval_operation (Genv.globalenv p) sp op vl =
  eval_operation (Genv.globalenv tp) sp op vl.

Lemma env_doesnt_matter2 :
  ∀ p tp,
  transf_program p = Some tp →
  ∀ sp addr vl,
  eval_addressing (Genv.globalenv p) sp addr vl =
  eval_addressing (Genv.globalenv tp) sp addr vl.

Lemma find_function_prop :
  ∀ p tp,
  transf_program p = Some tp →
  ∀ l rs f,
  find_function (Genv.globalenv p) l rs = Some f →
  ∃ tf,
  find_function (Genv.globalenv tp) l rs = Some tf ∧
  validate (fn_code f) (fn_code tf) empty empty = true.

Scheme exec_block_end_ind4 := Minimality for exec_block_end Sort Prop
  with exec_instrs_ind4 := Minimality for exec_instrs Sort Prop
 with exec_function_body_ind4 := Minimality for exec_function_body Sort Prop
  with exec_function_ind4 := Minimality for exec_function Sort Prop.

Lemma transf_function_correct :
  ∀ p tp f rs m,
  transf_program p = Some tp →
  exec_function (Genv.globalenv p) f empty_frame (Regmap.init Vundef) (Genv.init_mem p) rs m →
  ∀ tf,
  transf_function f = Some tf →
  exec_function (Genv.globalenv tp) tf empty_frame (Regmap.init Vundef) (Genv.init_mem p) rs m.

Theorem transf_program_correct :
  ∀ (p tp : program) (r : val),
  transf_program p = Some tp →
  exec_program p r →
  exec_program tp r.
```

Annexe B

Un vérificateur de traces

Le développement qui suit est l'implémentation en CAML du vérificateur de *trace scheduling*.

```
module Generalized =
  struct
    open PPC
    module CFG = Ppcutils.CFG
    exception Semantic_mismatch of string
    type explicitPPC =
        EPadd of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPaddi_ of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPaddis_ of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPaddze of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPallocframe of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPand_ of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPandc of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPandi_ of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPandis_ of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPb of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPbctr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPbctrl of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPbf of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPbl of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPblr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPbt of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPcmplw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPcmplwi of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPcmpw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPcmpwi of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPcror of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPdivw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPdivwu of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPeqv of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPextsb of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPextsh of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfreeframe of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfabs of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfadd of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfcmpu of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfcti of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfdiv of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfmadd of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfmr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfmsub of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfmul of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfneg of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfrsp of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPfsub of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPictf of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPiuctf of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlbz of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlbzx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlfd of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlfdx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlfs of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlfsx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlha of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlhax of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlhz of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlhzx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlfi of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlwz of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPlwzx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmfcrbit of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmflr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmtctr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmtlr of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmulli of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPmullw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPnand of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPnor of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPor of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPorc of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPori of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPoris of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPrlwinm of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPslw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPsraw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPsrawi of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPsrw of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstb of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstbx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstfd of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstfdx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstfs of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPstfsx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPsth of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
      | EPsthx of Ppcutils.resource list * explicitPPC option list *
          PPC.constant list * PPC.crbit list
```

```
  | EPstw of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPstwx of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPsubfc of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPsubfic of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPxor of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPxori of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPxoris of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPiundef of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPfundef of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
  | EPlabel of Ppcutils.resource list * explicitPPC option list *
      PPC.constant list * PPC.crbit list
let transfer_function block =
  let machine = Hashtbl.create 10 in
  let set value =
    if Hashtbl.mem machine value then Hashtbl.replace machine value
    else Hashtbl.add machine value
  and get value =
    if Hashtbl.mem machine value then Some (Hashtbl.find machine value)
    else None
  in
  let rec compute =
    function
      [] -> ()
    | elt : : remain ->
        let (reads, writes, constants, bits) =
          Ppcutils.reads_writes_constants_bits elt
        in
        let reads = List.map get reads in
        let f v = List.iter (fun e -> set e v) writes in
        begin match elt with
          Padd _ -> f (EPadd (writes, reads, constants, bits))
        | Paddi _ -> f (EPaddi_ (writes, reads, constants, bits))
        | Paddis _ -> f (EPaddis_ (writes, reads, constants, bits))
        | Paddze _ -> f (EPaddze (writes, reads, constants, bits))
        | Pallocframe _ ->
            f (EPallocframe (writes, reads, constants, bits))
        | Pand_ _ -> f (EPand_ (writes, reads, constants, bits))
        | Pandc _ -> f (EPandc (writes, reads, constants, bits))
        | Pandi_ _ -> f (EPandi_ (writes, reads, constants, bits))
        | Pandis_ _ -> f (EPandis_ (writes, reads, constants, bits))
        | Pb _ -> f (EPb (writes, reads, constants, bits))
        | Pbctr -> f (EPbctr (writes, reads, constants, bits))
        | Pbctrl -> f (EPbctrl (writes, reads, constants, bits))
        | Pbf _ -> f (EPbf (writes, reads, constants, bits))
        | Pbl _ -> f (EPbl (writes, reads, constants, bits))
        | Pblr -> f (EPblr (writes, reads, constants, bits))
        | Pbt _ -> f (EPbt (writes, reads, constants, bits))
        | Pcmplw _ -> f (EPcmplw (writes, reads, constants, bits))
        | Pcmplwi _ -> f (EPcmplwi (writes, reads, constants, bits))
        | Pcmpw _ -> f (EPcmpw (writes, reads, constants, bits))
        | Pcmpwi _ -> f (EPcmpwi (writes, reads, constants, bits))
        | Pcror _ -> f (EPcror (writes, reads, constants, bits))
        | Pdivw _ -> f (EPdivw (writes, reads, constants, bits))
        | Pdivwu _ -> f (EPdivwu (writes, reads, constants, bits))
        | Peqv _ -> f (EPeqv (writes, reads, constants, bits))
        | Pextsb _ -> f (EPextsb (writes, reads, constants, bits))
        | Pextsh _ -> f (EPextsh (writes, reads, constants, bits))
        | Pfreeframe -> f (EPfreeframe (writes, reads, constants, bits))
        | Pfabs _ -> f (EPfabs (writes, reads, constants, bits))
        | Pfadd _ -> f (EPfadd (writes, reads, constants, bits))
        | Pfcmpu _ -> f (EPfcmpu (writes, reads, constants, bits))
        | Pfcti _ -> f (EPfcti (writes, reads, constants, bits))
        | Pfdiv _ -> f (EPfdiv (writes, reads, constants, bits))
        | Pfmadd _ -> f (EPfmadd (writes, reads, constants, bits))
        | Pfmr _ -> f (EPfmr (writes, reads, constants, bits))
        | Pfmsub _ -> f (EPfmsub (writes, reads, constants, bits))
        | Pfmul _ -> f (EPfmul (writes, reads, constants, bits))
        | Pfneg _ -> f (EPfneg (writes, reads, constants, bits))
        | Pfrsp _ -> f (EPfrsp (writes, reads, constants, bits))
        | Pfsub _ -> f (EPfsub (writes, reads, constants, bits))
        | Pictf _ -> f (EPictf (writes, reads, constants, bits))
        | Piuctf _ -> f (EPiuctf (writes, reads, constants, bits))
        | Plbz _ -> f (EPlbz (writes, reads, constants, bits))
        | Plbzx _ -> f (EPlbzx (writes, reads, constants, bits))
        | Plfd _ -> f (EPlfd (writes, reads, constants, bits))
        | Plfdx _ -> f (EPlfdx (writes, reads, constants, bits))
        | Plfs _ -> f (EPlfs (writes, reads, constants, bits))
        | Plfsx _ -> f (EPlfsx (writes, reads, constants, bits))
        | Plha _ -> f (EPlha (writes, reads, constants, bits))
        | Plhax _ -> f (EPlhax (writes, reads, constants, bits))
        | Plhz _ -> f (EPlhz (writes, reads, constants, bits))
        | Plhzx _ -> f (EPlhzx (writes, reads, constants, bits))
        | Plfi _ -> f (EPlfi (writes, reads, constants, bits))
        | Plwz _ -> f (EPlwz (writes, reads, constants, bits))
        | Plwzx _ -> f (EPlwzx (writes, reads, constants, bits))
        | Pmfcrbit _ -> f (EPmfcrbit (writes, reads, constants, bits))
        | Pmflr _ -> f (EPmflr (writes, reads, constants, bits))
        | Pmr _ -> f (EPmr (writes, reads, constants, bits))
        | Pmtctr _ -> f (EPmtctr (writes, reads, constants, bits))
        | Pmtlr _ -> f (EPmtlr (writes, reads, constants, bits))
        | Pmulli _ -> f (EPmulli (writes, reads, constants, bits))
        | Pmullw _ -> f (EPmullw (writes, reads, constants, bits))
        | Pnand _ -> f (EPnand (writes, reads, constants, bits))
        | Pnor _ -> f (EPnor (writes, reads, constants, bits))
        | Por _ -> f (EPor (writes, reads, constants, bits))
        | Porc _ -> f (EPorc (writes, reads, constants, bits))
        | Pori _ -> f (EPori (writes, reads, constants, bits))
        | Poris _ -> f (EPoris (writes, reads, constants, bits))
        | Prlwinm _ -> f (EPrlwinm (writes, reads, constants, bits))
        | Pslw _ -> f (EPslw (writes, reads, constants, bits))
        | Psraw _ -> f (EPsraw (writes, reads, constants, bits))
        | Psrawi _ -> f (EPsrawi (writes, reads, constants, bits))
        | Psrw _ -> f (EPsrw (writes, reads, constants, bits))
        | Pstb _ -> f (EPstb (writes, reads, constants, bits))
        | Pstbx _ -> f (EPstbx (writes, reads, constants, bits))
        | Pstfd _ -> f (EPstfd (writes, reads, constants, bits))
        | Pstfdx _ -> f (EPstfdx (writes, reads, constants, bits))
        | Pstfs _ -> f (EPstfs (writes, reads, constants, bits))
        | Pstfsx _ -> f (EPstfsx (writes, reads, constants, bits))
        | Psth _ -> f (EPsth (writes, reads, constants, bits))
        | Psthx _ -> f (EPsthx (writes, reads, constants, bits))
        | Pstw _ -> f (EPstw (writes, reads, constants, bits))
        | Pstwx _ -> f (EPstwx (writes, reads, constants, bits))
        | Psubfc _ -> f (EPsubfc (writes, reads, constants, bits))
        | Psubfic _ -> f (EPsubfic (writes, reads, constants, bits))
        | Pxor _ -> f (EPxor (writes, reads, constants, bits))
        | Pxori _ -> f (EPxori (writes, reads, constants, bits))
        | Pxoris _ -> f (EPxoris (writes, reads, constants, bits))
        | Piundef _ -> f (EPiundef (writes, reads, constants, bits))
        | Pfundef _ -> f (EPfundef (writes, reads, constants, bits))
        | Plabel _ -> f (EPlabel (writes, reads, constants, bits))
        end;
        compute remain
  in
  compute block; machine
let abstract traces transfer_function = List.map transfer_function traces
let is_isomorphic src tgt =
  Hashtbl.fold
    (fun resource abs b ->
       b &&
       (if Hashtbl.mem tgt resource then Hashtbl.find tgt resource = abs
        else false))
    src true
let unoption st =
  match st with
    Some t -> t
  | None -> raise (Semantic_mismatch "balooney\n")
let infer_link source_traces target_traces =
  let l1 = List.sort compare source_traces in
  let l2 = List.sort compare target_traces in
  if not (List.length l1 = List.length l2) then
    raise
      (Semantic_mismatch
         "source and target don't have the same number of traces!");
  let b = List.fold_left2 (fun b e1 e2 -> b && e1 = e2) true l1 l2 in
  List.fold_left2 (fun b e1 e2 -> b && is_isomorphic e1 e2) true l1 l2
let validation_graph code =
  let cfg = Ppcutils.reconstruct_cfg code in
  let tip = List.hd (Ppcutils.chop_code code) in
  let back_edges = CFG.back_edges cfg tip in
  let tree = List.fold_left CFG.del_arrow cfg back_edges in
  let be_sources = List.map (fun (a, _) -> a) back_edges in
  let tree =
    if not (List.length be_sources = 0) then
      let (fake, tree) = CFG.add_node tree [] in
      List.fold_left (fun t s -> CFG.add_arrow t s fake) tree be_sources
    else tree
  in
  let bet = List.map (fun (_, b) -> b) back_edges in
  let root = Ppcutils.root code cfg in
  let bet = bet in
  let traces = ref [] in
  let rec windows node trace =
    let succs = CFG.node_sons tree node in
    if List.length succs = 0 then
      traces := (trace @ CFG.get_datum tree node) : :!traces
    else if List.mem node bet then
      begin
        traces := trace : :!traces;
        List.iter (fun elt -> windows elt (CFG.get_datum tree node)) succs
      end
    else
      List.iter (fun elt -> windows elt (trace @ CFG.get_datum tree node))
        succs
  in
  let () = windows root [] in!traces
let validate source target =
  let source_traces = validation_graph source in
  let target_traces = validation_graph target in
  let source_exec_traces = abstract source_traces transfer_function
  and target_exec_traces = abstract target_traces transfer_function in
  let pi = infer_link source_exec_traces target_exec_traces in
  if pi then Printf.fprintf stdout "Generalized : OK\n%!"
  else raise (Semantic_mismatch "well...");
  true
end
```