COMPILATION OPTIMISANTE VÉRIFIÉE SUR FORME SSA

LÉO STEFANESCO

Stage de L3 effectué du 2 juin au 25 juillet 2014 au sein de l'équipe Celtique de l'INRIA / IRISA à Rennes.

Encadrants :
David Pichardie, ENS Rennes / INRIA
Delphine Demange, Université de Rennes 1 / INRIA

1. Introduction

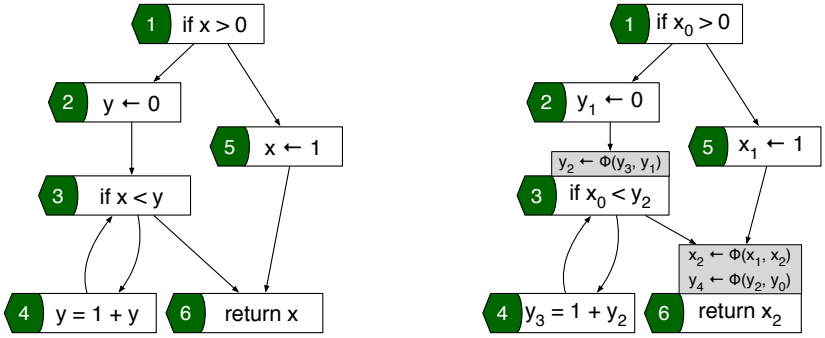
Il existe de nombreux outils permettant de prouver des propriétés d'un programme, tels que son pire temps d'exécution, ou l'absence d'erreurs pendant l'exécution, qui sont très utilisés dans la conceptions de systèmes critiques. Néanmoins, ces outils opèrent en général sur le code source du programme, ainsi un bug dans le compilateur pourrait invalider ces garanties. D'où l'importance de la correction du compilateur et l'intérêt de disposer d'un compilateur certifié, c'est à dire dont on aurait une preuve vérifiée par ordinateur qu'il conserve la sémantique des programmes sources.

Le premier compilateur certifié ayant des performances comparables à un compilateur tel que GCC est CompCert[6] de Xavier Leroy, un compilateur C presque entièrement écrit en Coq.

Toutefois, CompCert n'utilise pas la forme SSA (Static Single Assignment), une représentation intermédiaire qui permet d'effectuer des optimisations plus simplement et rapidement, et qui est utilisée par la plupart des compilateurs modernes, comme LLVM, GCC ou HotSpot. CompCertSSA[2], de Gilles Barthe, Delphine Demange et David Pichardie, ajoute à CompCert un *middle-end* opérant sur une représentation intermédiaire SSA.

L'objet de ce stage a été de rajouter à CompCertSSA une nouvelle passe d'optimisation propre à la forme SSA, *Sparse Conditional Constant Propagation* (SCCP) [10], de prouver sa correction à l'aide de Coq et d'évaluer ses performances en terme de précision et de performance. Pour cela, j'ai d'abord implémenté une autre passe, *Sparse Simple Constant Propagation*, qui est similaire à SCCP, et dont la preuve m'a servi de base pour prouver SCCP.

Plan. Le reste de ce rapport est organisé comme suit. La section 2 présente les résultats de théorie de la compilation qui ont été utilisés et la section 3 traite des méthodes formelles dans le cadre d'un compilateur. Les sections 4 et 5 décrivent les deux passes d'optimisation qui ont été implémentées pendant ce stage, et la section 6 présente les résultats expérimentaux obtenus. Enfin la section 7 discute des travaux connexes.



(A) Exemple de programme (B) Le même programme sous forme SSA

FIGURE 1. La forme SSA

2. COMPILATION

On décrit dans cette section la théorie de la compilation sur laquelle repose ce travail : d'abord la forme SSA, puis l'analyse de flot de données, technique sur laquelle reposent les optimisations implémentées pendant ce stage.

2.1. **SSA.**

La forme SSA ajoute aux représentations intermédiaires traditionnelles certains invariants et une nouvelle instruction, appelée ϕ.

2.1.1. *Représentation.*

On considère des programmes qui sont sous la forme d'un graphe de flot de contrôle (CFG). Un CFG possède un unique point d'entrée.

Dans cette partie, on considère que les instructions sont soit de la forme $r \leftarrow x \ op \ y$, où op est un opérateur — tel que $+$ — et x, y sont des variables, soit des instructions conditionnelles.

2.1.2. *La forme SSA.*

La forme SSA (décrite par exemple dans [9]) impose des contraintes sur le graphe : chaque variable n'est affectée statiquement qu'une seule fois ; concrètement, chaque variable n'est présente à gauche d'une flèche qu'une seule fois.

Pour passer du programme source à SSA, on rajoute en général aux variables un numéro de version pour chaque affectation, ainsi le programme suivant :

$$\begin{aligned} &\mathrm{x} \leftarrow 0 \\ &\mathrm{y} \leftarrow 5 \\ &\mathrm{x} \leftarrow \mathrm{x} + \mathrm{y} \end{aligned}$$

devient :

$$\begin{aligned} &\mathrm{x}_0 \leftarrow 0 \\ &\mathrm{y}_0 \leftarrow 5 \\ &\mathrm{x}_1 \leftarrow \mathrm{x}_0 + \mathrm{y}_0 \end{aligned}$$

Néanmoins, il arrive que, dans le programme source, une variable soit affectée dans deux branche différentes, sa valeur dépend alors de quelle branche a été prise ; c'est pourquoi la forme SSA introduit la fonction ϕ, qui permet de faire dépendre la valeur d'une variable de la branche qui a été prise. Par

exemple, figure 1, au nœud 3 du programme source, la valeur de y peut venir soit du nœud 2 soit du nœud 4. La ϕ-instruction au nœud 3 du programme en forme SSA permet dire que y_2 est égale à y_1 si l'exécution vient du nœud 2, et est égal à y_3 si l'exécution vient du nœud 4, ainsi la valeur de y et y_2 coïncident. De même pour x et y au nœud 6.

Définition 1 (Fonction ϕ)**.** Soit pc un nœud du CFG, soit n le nombre de prédécesseurs de pc. Une ϕ-instruction en pc est de la forme $r \leftarrow \phi(x_1, \ldots, x_n)$, où r et les x_i sont des variables. La valeur de r après l'exécution de cette instruction est la valeur de x_i, si le précédent bloc exécuté était le i-ème prédécesseur de pc.

On requiert de plus que le code soit sous forme SSA stricte.

Définition 2 (Domination et SSA strict)**.** Soit f une fonction (son CFG) et x, y deux points de f.

Alors x domine y si tout chemin du point d'entrée à y passe par x. Si de plus $x \neq y$, alors x domine strictement y.

f est sous forme SSA stricte si f est sous forme SSA et si pour toute variable r, tout point où r est utilisé (c'est à dire toute occurrence de r qui n'est pas une définition) est dominé par le point de définition de r.

Dans la suite, on désignera par forme SSA la forme SSA stricte.

2.1.3. *Le graphe SSA.*

Définition 3 (Graphe SSA et def-use chain)**.** On appelle graphe SSA le graphe orienté ayant les mêmes nœuds que le CFG, mais dont les arcs vont des points de définitions des variables vers leurs points d'utilisations.

On appelle *def-use chain*, et on note DefUse la map qui à une variable x associe la liste de ses points d'utilisations ; ou de manière équivalente, la liste des successeurs du point de définition de x dans le graphe SSA.

L'intérêt de ce graphe est que la forme SSA garantie que les propriétés qui sont vraies au point de définition de x le restent à ses points d'utilisations. Cela permet de voir la forme SSA comme une forme équationnelle, comme le montre le lemme équationnel[2], qui est énoncé en 3.2. De plus, l'utilisation du graphe SSA permet d'implémenter des analyses efficaces, comme on le verra dans les sections 4, 5 et 6.

2.2. **Analyse.** L'objet de l'analyse statique (pour une référence, voir par exemple [8]) est de sur-approximer en un temps fini certaines propriétés d'un programme, ce qui permet au compilateur d'effectuer des optimisations.

2.2.1. *Cadre formel.* Souvent, les propriétés auxquelles on s'intéresse forment un treillis, où la relation d'ordre coïncide avec l'implication.

Définition 4 (Treillis)**.** Un (semi-)treillis est un tuple $(A, \sqsubseteq, \sqcup, \bot, \top)$ tel que $\sqsubseteq$ est une relation d'ordre sur A, $\bot$ et $\top$ sont respectivement le plus petit et le plus grand élément de A pour $\sqsubseteq$ et $\sqcup$ est un *infimum*, qu'on appelle le *join*.

L'exemple présenté dans cette section est la propagation classique de constantes, telle que Constprop, celle de CompCert.

En général, l'ensemble des propriétés qu'on approxime, *le domaine concret*, noté $L^\flat$, n'est pas calculable ou est très coûteux à calculer, c'est pourquoi on le sur-approxime par un autre treillis, le *domaine abstrait*, $L^\sharp$. On relie les deux par une fonction de concrétisation, qui est croissante, $\gamma : L^\sharp \to L^\flat$: pour tout $x \in L^\sharp$, $\gamma(x)$ est la propriété concrète la plus précise qui est approximée par x.

Exemple. Dans le cas de la propagation de constantes, la propriété qu'on approxime est l'ensemble des valeurs que chaque variable prend lors de toutes les exécutions possibles du programme. Le domaine concret est donc le treillis $L^\flat$ des parties de l'ensemble des valeurs, qu'on appelle `val`.

Définition 5. Le type `val` des valeurs dans CompCert est défini par (le mot clé `Inductive` sert en Coq à définir un type algébrique, comme `type` en OCaml) :

```
Inductive val: Type :=
  | Vundef
  | Vint: (x: int)
  | Vfloat: (x: float)
  | Vptr: (b: block) (offset: int)
```

`Vundef` est la valeur que prennent les variables non initialisées.

Le domaine abstrait est le treillis produit des *treillis des constantes* indexés par les variables :

Définition 6 (Treillis des constantes)**.** On appelle treillis des constantes le treillis $L = (\{\bot, \top\} \cup \texttt{int}, \sqsubseteq, \sqcup, \bot, \top)$, où les éléments de `int` ne sont pas comparables entre eux, et le *join* est défini par :

$$a \sqcup b = \begin{cases} a & \text{si } b = \bot \\ b & \text{si } a = \bot \\ a & \text{si } a = b \in \texttt{int} \\ \top & \text{sinon} \end{cases}$$

On définit la fonction de concrétisation par :

$$\gamma(a) = \begin{cases} \texttt{val} & \text{si } a = \top \\ \varnothing & \text{si } a = \bot \\ \{a\} & \text{si } a \in \texttt{int} \end{cases}$$

Ainsi, les éléments de $L^\sharp$ sont des environnements abstraits qui à chaque variable associe sa valeur abstraite.

2.2.2. *Calcul de l'approximation.* Il existe deux types d'analyses de flot de données, les analyses en avant et les analyses en arrière, selon que, respectivement, l'information se propage dans le sens des arcs du CFG ou dans le sens inverse. Le cas en arrière se déduit du cas en avant en considérant le graphe inverse.

Un exemple d'analyse en arrière est l'analyse des variables vivantes : une variable x est dite vivante en pc s'il existe un chemin d'exécution d'origine pc tel que x est lue avant qu'on écrase son contenu. On voit que le fait que x soit vivante en pc dépend des successeurs de x.

Dans le cas d'une analyse en avant, comme la propagation de constantes, pour calculer cette approximation, on associe à chaque instruction i deux éléments de $L^\sharp$, in_i et out_i, qui représentent respectivement ce qu'on sait juste avant l'exécution de l'instruction, et juste après. On dispose aussi d'une fonction de transfert pour chaque instruction i, $f_i : L^\sharp \to L^\sharp$.

Le résultat de l'analyse est alors le plus petit point fixe du système d'équations suivant, où i décrit l'ensemble des instructions de f :

$$\begin{cases} \text{out}_i = f_i(\text{in}_i), \\ \text{in}_i = \sqcup_{j \text{ prédécesseur de } i} \text{out}_i \end{cases}$$

La seconde ligne dit que ce qu'on sait juste avant i est la plus forte propriété qui est vraie quelle que soit l'arête incidente à i qui est prise. Pour calculer ce point fixe, on utilise des algorithmes itératifs, comme l'algorithme de Kildall[4].

On cherche le plus petit point fixe car c'est le plus précis, puisque $x \sqsubseteq y$ signifie que la propriété du programme associée à x implique celle associée à y.

Retour à l'exemple. Pour une instruction i de la forme $r \leftarrow x \; op \; y$, la fonction de transfert est :

$$f_i(lv) = \begin{cases} lv\{r := [\![k_1 \, op \, k_2 \,]\!]\} \sqcup lv & \text{si } lv(x) = k_1 \in \texttt{int} \text{ et } lv(y) = k_2 \in \texttt{int} \\ lv & \text{si } lv(x) = \bot \text{ ou } lv(y) = \bot \\ lv\{r := \top\} & \text{sinon} \end{cases}$$

où $lv\{r := x\}$ est égal à lv sur toutes les composantes sauf r, où il vaut x, et $[\![k_1 \, op \, k_2]\!] \in \texttt{val}$ est le résultat obtenu en appliquant op à ses opérandes.

On remarque qu'exécuter la fonction de transfert f_i revient à interpréter i dans le domaine abstrait $L^\sharp$. L'interprétation abstraite sur le treillis des constantes serait :

$$[\![x \; op \; y]\!]^\sharp lv = \begin{cases} [\![k_1 \, op \, k_2]\!] & \text{si } lv(x) = k_1 \in \texttt{int} \text{ et } lv(y) = k_2 \in \texttt{int} \\ \bot & \text{si } lv(x) = \bot \text{ ou } lv(y) = \bot \\ \top & \text{sinon} \end{cases}$$

A partir de l'interprétation abstraite, on peut définir $f_{x \, op \, y}$ de la manière suivante :

$$f_{x \, op \, y}(lv) = lv\{r := lv(r) \sqcup [\![x \; op \; y]\!]^\sharp lv\}.$$

Optimisation. L'optimisation consiste alors à remplacer toutes les instructions dont on peut prouver qu'elles ont toujours la même valeur par cette valeur.

2.2.3. *Analyse sur SSA.* Comme on le verra aux sections 4 et 5, si le programme est sous forme SSA, comme chaque variable n'est définie qu'en un unique point, on peut parfois, sans perte de précision, ne garder en mémoire qu'un seul environnement abstrait (on dit alors que c'est une analyse *flow insensitive*). C'est possible lorsque l'analyse ne met à jour la valeur abstraite d'une variable que lorsqu'elle visite son point de définition, comme dans le cas de la propagation de constante.

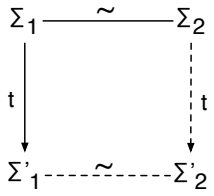


FIGURE 2. *lock-step forward simulation*

3. Méthodes formelles

La particularité du compilateur sur lequel j'ai travaillé est qu'il est certifié. Cette section présente les méthodes utilisées pour prouver la correction d'un compilateur, puis décrit CompCertSSA.

3.1. **CompCert.**

CompCert, un compilateur C vérifié écrit en Coq, est la base sur laquelle repose CompCertSSA, le compilateur sur lequel j'ai travaillé pendant ce stage. [6] présente CompCert à un haut niveau et [7] en fait une présentation détaillée. Pour un compilateur, correct signifie que le code qu'il génère a la même sémantique que le programme source ; plus précisément, le théorème de correction de CompCert est le suivant :

Théorème 1 (Théorème de simulation en arrière)**.** Pour tout programme source S, si C est le résultat de la compilation de S, alors

$$S \text{ safe } \Rightarrow (\forall B \in \text{Behaviors}, C \Downarrow B \Rightarrow S \Downarrow B).$$

Les éléments de Behaviors sont les comportements observables depuis l'extérieur, c'est à dire la suite (finie ou infinie) des appels à des fonctions externes, en particulier les appels systèmes. $X \Downarrow B$ signifie que le comportement de X peut être B (en effet la sémantique de C est non déterministe, à cause par exemple de l'ordre d'évaluation des paramètres qui est laissé au choix du compilateur, donc un programme source peut avoir plusieurs comportements).

L'énoncé ci-dessus signifie donc que si S est bien défini (pas de comportements non définis, par exemple des lectures de variables non initialisées), alors tout comportement que peut avoir C est un comportement que peut avoir S, en d'autres termes, la sémantique du programme compilé est un *raffinement* de celle du programme source.

Pour prouver cela, comme le langage cible (de l'assembleur) est déterministe, il existe un unique comportement B tel que $C \Downarrow B$, il suffit de montrer le théorème suivant, ce qui est en général plus simple.

Théorème 2 (Théorème de simulation en avant)**.** Avec les mêmes notations,

$$S \text{ safe } \Rightarrow (\forall B \in \text{Behaviors}, S \Downarrow B \Rightarrow C \Downarrow B).$$

Pour arriver à ce théorème, on prouve, pour chaque transformation que subit le programme, un théorème de préservation de la sémantique de la forme du théorème précédent, puis on les compose.

3.1.1. *Simulations.* Pour chaque langage, on définit une sémantique petit pas. On note $\Sigma_1 \xrightarrow{t} \Sigma_2$ un transition de cette sémantique, où Σ_1 et Σ_2 sont des états de l'exécutions du programme et t la trace des appels externes effectués par le programme en passant de l'état Σ_1 à l'état Σ_2.

Définition 7 (État d'exécution)**.** L'état Σ du programme est un 5-uplet (s, f, pc, rs, m) composé de :
- s, la pile d'exécution,
- f, la fonction en cours d'exécution,
- $pc \in \texttt{node}$, le pointeur d'instruction *(program counter)*,
- $rs : \texttt{reg} \to \texttt{val}$, l'environnement concret, qui à un registre associe sa valeur,
- m, l'état de la mémoire.

Pour prouver la correction d'une transformation entre un programme S appartenant à un langage L_S vers un programme C appartenant à un langage L_C, on utilise un *diagramme de simulation*. Le plus simple, et celui que j'ai utilisé dans ma preuve, est le *lock-step forward simulation*.

Notons $State_S$ et $State_C$ l'ensemble des états pour S et C.

Définition 8 (Lock-step simulation)**.** Il y a une lock-step simulation entre S et C si on dispose de $\sim \subseteq State_S \times State_C$ tel que, pour tout $\Sigma_1, \Sigma_1' \in State_S$ et $\Sigma_2 \in State_C$ tels que $\Sigma_1 \sim \Sigma_2$ et $\Sigma_1 \xrightarrow{t} \Sigma_1'$, il existe $\Sigma_2' \in State_C$ tels que $\Sigma_1' \sim \Sigma_2'$ et $\Sigma_2 \xrightarrow{t} \Sigma_2'$.

Ce diagramme est illustré figure 2 : les traits pleins sont les hypothèses, et les traits en pointillés sont les conclusions.

On prouve alors la préservation de la sémantique du programme grâce au théorème suivant :

Théorème 3. S'il existe une lock-step simulation entre S et C et que les états initiaux de S et C (les états juste avant le début de l'exécution de S et C) et finaux (l'état des programmes après avoir terminé leurs exécutions) sont en relation pour $\sim$, alors $\forall B,\ S \Downarrow B \Rightarrow C \Downarrow B$.

3.1.2. *Validation a posteriori.* Parfois, on préfère éviter de prouver directement la correction d'une certaine fonction f. A la place, on implémente et on prouve un validateur (une fonction à valeurs dans les booléens), qui accepte le résultat de f uniquement si ce résultat vérifie la spécification de f.

Les avantages de la validation a posteriori sont que c'est souvent plus simple, que cela permet de changer l'implémentation sans changer la preuve et d'utiliser des structures de données impératives (qui sont absentes de Coq car c'est un langage purement fonctionnel), puisque f peut être écrite dans un autre langage, le plus souvent OCaml. L'inconvénient est que la phase de validation peut être coûteuse en calculs, qui n'auraient pas étés nécessaires si on avait fait une preuve directe.

Cette méthode est par exemple utilisée dans CompCert pour l'allocation de registres, plus particulièrement pour colorier le graphe d'interférence, ce qui permet d'une part d'utiliser des listes doublement chaînées impératives, mais aussi de changer les heuristiques indépendamment de la preuve. De plus, la vérification d'un coloriage de graphe est simple et efficace.

3.2. **CompcertSSA.** CompCertSSA [2] ajoute à CompCert un *middle-end* utilisant la forme SSA. Pour ce faire, les invariants définissant la forme SSA (stricte) ont été formalisés en Coq et le passage de `RTL` (une représentation intermédiaire de CompCert, sur laquelle opère Constprop) à `SSA` (le nom du langage intermédiaire sous forme SSA), et la transformation inverse (afin d'utiliser le *back-end* de CompCert pour terminer la compilation), ont été implémentées et prouvées.

De plus, CompCertSSA contient aussi une optimisation propre à la forme SSA, *Global Value Numbering*[1] (GVN), qui permet d'éliminer des calculs redondants en déterminant que deux expressions sont équivalentes, mais elle n'effectue pas d'opérations sur les constantes. De plus GVN est beaucoup plus coûteuse que la propagation de constantes, c'est pourquoi en général les compilateurs ne l'utilisent que pour les niveaux d'optimisation élevés (à partir de O2 pour LLVM par exemple), et tendent donc à implémenter ces deux optimisations.

3.2.1. *Syntaxe.* La syntaxe de `SSA` est très proche de celle de `RTL`, la seule différence étant l'ajout des ϕ-instructions : à chaque nœud est associé un *ϕ-bloc*, une liste de ϕ-instructions (éventuellement vide).

Concrètement, le code de la fonction f est représenté par deux maps : CFG_f, qui à un point de programme (de type `node`, qui sont des entiers) associe l'instruction qui s'y trouve ; et PHI_f, qui à un point de programme associe son ϕ-bloc associé. Chaque instruction contient les points de programme qui lui succèdent.

Les instructions sont les suivantes :

```
Inductive instr : Type :=
  | Inop (pc: node)
  | Iop (op: operation) (args: list reg) (res: reg) (pc: node)
  | Iload (chk:chunk) (addr:addressing) (args: list reg) (res: reg) (pc: node)
  | Istore (chk:chunk) (addr:addressing) (args:list reg) (src: reg) (pc: node)
  | Icall (sig: signature) (fn:ident) (args: list reg) (res: reg) (pc: node)
  | Icond (cond: condition) (args: list reg) (ifso ifnot: node)
  | Ireturn (or: option reg).
```

Les paramètre nommées `pc` (et `ifso`, `ifnot` pour `Icond`) sont les points de programme successeurs dans le graphe. Dans la suite du rapport, on notera $r \leftarrow x\ op\ y$ pour `(Iop op [x, y] r)`.

Une ϕ-instruction est définie par :

```
Inductive phiinstruction: Type :=
  | Iphi (args: list reg) (res: reg)
```

`Iphi [x`$_1$`, ..., x`$_n$`] r` correspond à $r \leftarrow \phi(x_1, \ldots, x_n)$.

3.2.2. *Invariants structurels.* Les invariants de la forme SSA sont traduits de manière directe, par exemple l'invariant de domination (définition 2) est formalisé de la manière suivante dans CompCertSSA, où `f` est une fonction :

```
∀ x u d, use f x u → def f x d → dom f d u.
```

De plus, on demande que le programme soit *normalisé* : si x est un nœud dont l'un des successeurs a plusieurs prédécesseurs, alors x est un `Inop`. Cela simplifie la gestion des ϕ-instruction, car les ϕ-instructions ne sont alors présentes que sur les nœuds qui ont plusieurs prédécesseurs. En outre, le point d'entrée doit être un `Inop`.

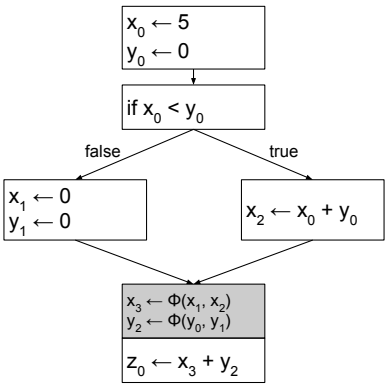


FIGURE 3. Exemple de programme où SSCP et SCCP trouvent des constantes

3.2.3. *Sémantique.* La sémantique de `SSA` est très similaire à celle de `RTL`, la seule différence étant l'addition de la sémantique des ϕ-instructions.

Grâce à la normalisation, on a juste à traiter le cas des transitions de `Inop` vers un nœud ayant plusieurs prédécesseurs. La sémantique de ce cas est :

$$\frac{\begin{array}{c} CFG_f(pc) = \texttt{Inop}\ pc' \\ pc' \text{ a plusieurs prédécesseurs dans } f \\ PHI_f(pc') = phib \\ pc \text{ est le } k\text{-ième prédécesseur de } pc' \end{array}}{(s, f, sp, pc, rs, m) \xrightarrow{\epsilon} (s, f, sp, pc', \texttt{phistore}\ k\ rs\ phib, m)}$$

où ϵ est la trace vide, et `phistore` k rs $phib$ est égal à rs après avoir effectué, pour chaque ϕ-instruction, la copie du k-ième argument, comme spécifié à la définition 1.

3.2.4. *Lemme équationnel.* Le lemme équationnel (mentionné en 2.1.3) dit que si une instruction est une `Iop`, on peut la voir comme une équation qui est valide dans son domaine de domination.

Théorème 4 (Lemme équationnel)**.** Soit f une fonction sous forme SSA, i une instruction de f de la forme $r \leftarrow x\ op\ y$, pc un point de programme dominé strictement par i. Soit de plus (s, f, sp, pc, rs, m) un état accessible de f.
Alors exécuter $x\ op\ y$ avec l'état de registre rs a pour résultat la valeur de r dans rs.

Cela n'est pas vrai si l'on n'est pas en SSA, car alors x peut être redéfini de manière arbitraire entre son définition et pc.

4. Sparse Simple Constant Propagation (SSCP)

SSCP[10] est l'une des deux passes d'optimisations qui ont été implémentées pendant ce stage. C'est la transposition à SSA de Constprop : elle trouve les mêmes constantes, mais elle est plus rapide : sa complexité dans le pire des cas est $O(mn)$, contre $O(mn^2)$ pour Constprop, où n et m sont respectivement le nombre de nœuds et d'arcs du CFG.

Dans le contexte de ce stage, SSCP était une étape intermédiaire pour arriver à SCCP, qu'on peut voir comme un raffinement de SSCP. SCCP trouve strictement plus de constantes que SSCP, par exemple, dans la figure 3, SSCP détermine que $\mathtt{y}_2$ est constant égal à 0, mais seul SCCP détermine que puisque la branche de gauche ne peut être prise, $\mathtt{x}_3$ vaut 5, et donc $\mathtt{z}_0$ vaut toujours 5.

4.1. **Notations.** On note respectivement $rs : \mathtt{reg} \to \mathtt{val}$ et $lv : \mathtt{reg} \to L$ l'environnement concret et l'environnement abstrait, où L est le treillis des constantes (définition 6).

Si e est une expression, on note $[\![e]\!]^\sharp lv$ la valeur de son interprétation abstraite dans le treillis des constantes, et $[\![e]\!]rs$ son interprétation selon la sémantique concrète de `SSA`. On note $[\![r]\!]^\sharp lv \succeq v \Leftrightarrow v \in \gamma(lv(r))$, où $v \in \mathtt{val}$ et $r \in \mathtt{reg}$.

On écrira, par abus de notation, toutes les instructions non conditionnelles autres que ϕ — par exemple les `load`, `store`, ...— sous la forme $r \leftarrow x\ op\ y$, car ce sont les opérations qui nous intéressent, en effet on ne gère pas la mémoire (l'interprétation abstraite d'un `load` est toujours $\top$).

4.2. **Algorithme.** On calcule lv pour une fonction, pour cela on maintient une liste des variables à visiter et tant qu'elle n'est pas vide, on extrait un élément de cette liste et on propage sa valeur abstraite à tout ses points d'utilisation. A chaque fois que la valeur de treillis d'une variable change, on ajoute cette variable à la liste.

Algorithm 1 Sparse Simple Constant Propagation (Analyse)

```
lv ← λr.⊥
WorkList ← ∅
for r ∈ Params do
    lv[r] ← ⊤
    WorkList ← WorkList ∪ DefUse(r)
for {r ← x op y} ∈ Prog do
    if [[x op y]]♯ lv ≠ ⊥ then
        lv[r] ← [[x op y]]♯ lv
        WorkList ← WorkList ∪ DefUse(r)
while WorkList ≠ ∅ do
    i ← Pick (WorkList)
    if i = {r ← φ(x_1, ..., x_k)} then
        newval ← [[x_1]]♯ lv ⊔ ··· ⊔ [[x_k]]♯ lv
    else if i = {r ← x op y} then
        newval ← [[x op y]]♯ lv
    if (newval ⊔ [[r]]♯ lv) ≠ [[r]]♯ lv then
        lv[r] ← newval ⊔ [[r]]♯ lv
        WorkList ← WorkList ∪ DefUse(r)
WorkList ← map (λ(v : L) → (v = ⊥)? ⊤ : v) WorkList
```

Algorithm 2 TRANSFORMATION

1: **for** $\{r \leftarrow x\ op\ y\} \in Prog$ **do**
2: **if** $[\![x\ op\ y]\!]^\sharp lv = k \in \texttt{int}$ **then**
3: Replace $x\ op\ y$ by k at this program point

L'algorithme 1 calcule aux lignes 1–18 le plus petit point fixe lv du système d'équations suivant :

$$\begin{cases} [\![r]\!]^\sharp lv = [\![x\ op\ y]\!]^\sharp lv & \text{pour } \{r \leftarrow x\ op\ y\} \in f \\ [\![r]\!]^\sharp lv = \bigsqcup [\![x_i]\!]^\sharp lv & \text{pour } \{r \leftarrow \phi(x_1, \cdots, x_k)\} \in f \end{cases}$$

Pour chaque instruction, la valeur de treillis de la destination est égale à l'interprétation abstraite de la partie droite de l'instruction. Pour chaque ϕ-instruction, la valeur de treillis de la destination est égale au supremum de celles des arguments de ϕ.

Gestion de $\bot$ et de `Vundef`. Ligne 19, on remplace tous les $\bot$ du point fixe par des $\top$ car dans le domaine abstrait des constantes de CompCert, qu'on utilise, $\gamma(\bot) = \varnothing$, donc `Vundef` n'est abstrait que par $\top$, et pour trouver le *plus petit* point fixe, il faut initialiser les valeurs de treillis à $\bot$. Or pendant l'exécution, les variables dont la valeur de treillis est $\bot$ (celles qui ne sont pas initialisées dans le programme C) vaudrons `Vundef`, il est donc nécessaire qu'elles soient à $\top$ à la fin de l'analyse.

Le résultat de l'analyse reste bien sûr un post-point fixe, qui est la propriété sur laquelle repose la correction de la transformation.

4.3. **Preuve de correction.** L'analyse effectuée étant *flow insensitive* (les valeurs de treillis associées aux registres ne dépendent pas du point de programme), elle n'est pas valable partout pour tout registre (on entend par valable, pour un rs et r donné, $lv(r) \succeq rs(r)$). En effet, pour tout registre r dont la valeur de treillis est une constante, au point d'entrée, r contient `Vundef`, et $\forall k, \texttt{Vundef} \notin \gamma(k)$. En somme, une analyse *flow insensitive* pour une certaine variable n'est valable que dans le sous-graphe où elle est définie, or si le graphe est sous forme SSA, ce sous-graphe est exactement le domaine de domination de son unique point de définition.

Définition 9 (Stricte domination par une définition)**.** Soit f une fonction, r une variable et pc un point de programme.

On dit que la définition de r domine strictement pc dans f, et on note $\mathrm{dsd}\, f\, r\, pc$, lorsque l'une des conditions suivantes est vérifiée :

— r est définie en pc_0 et pc_0 domine strictement pc,
— r est définie par une ϕ-instruction en pc,
— r est un paramètre de la fonction
— r n'est jamais définie.

Définition 10 (Bonne approximation)**.** Soient f une fonction, rs un environnement concret et lv un environnement abstrait. On dit que lv est une bonne approximation par rapport à rs en pc, et on note $BA_f(lv, rs, pc)$, la propriété suivante :

$$\forall r \in \texttt{reg},\ \mathrm{dsd}\, f\, r\, pc \Rightarrow [\![r]\!]^\sharp lv \succeq [\![r]\!] rs.$$

Intuitivement, l'analyse n'est valable pour r que dans le domaine où on est sûr que r a précédemment été défini, c'est à dire dans son domaine de stricte domination.

On note f la fonction originale et tf la fonction transformée par notre algorithme.

La preuve de correction consiste en une *lock-step forward simulation*, avec la relation $\sim$ définie par

$$\frac{\mathrm{BA}(lv, rs, sp)}{(s, f, sp, pc, rs, m) \sim (s', tf, sp, pc, rs, m)}$$

L'hypothèse dénote que notre analyse est correcte et la conclusion dénote que la transformation préserve la sémantique du programme (par exemple, rs doit être le même dans les deux états).

4.3.1. *Correction de l'analyse.* Supposons que $BA(lv, rs, pc)$ et que $\Sigma_1 \xrightarrow{t} \Sigma_2$. Notons pc le point de programme de Σ_1, pc' celui de Σ_2 et rs, rs' leurs états de registres respectifs. Il s'agit de montrer $BA(lv, rs', pc')$. On effectue la preuve par induction sur la dérivation de $\Sigma_1 \xrightarrow{t} \Sigma_2$. Les cas intéressants sont :

- **L'instruction en pc est un *nop* et son successeur pc' possède un ϕ-bloc.** La valeur du registre qui est affecté par la ϕ-instruction prend la valeur du paramètre de ϕ qui correspond à pc.
  Soit r un registre tel que $\mathrm{dsd}\, r\, pc'$.
  - Si r n'est pas défini dans ce ϕ-bloc, alors $rs'(r) = rs(r)$. Notons p_r son point de définition ; p_r domine strictement pc', donc domine au sens large pc. De plus l'instruction en pc est un *nop*, donc $p_r \neq pc$, par suite p_r domine strictement pc, d'où $\mathrm{dsd}\, r\, pc$. On conclut alors grâce à $BA(lv, rs, pc)$.
  - Supposons maintenant que r est défini dans le ϕ-bloc associé à pc', par $r \leftarrow \phi(x_1, \cdots, x_n)$. Si on note k le rang de pc parmi les prédécesseurs de pc', alors $rs'(r) = rs(x_k)$, il suffit donc de montrer que $\mathrm{dsd}\, x_k\, pc$. Or le programme est sous forme SSA stricte, donc la définition de x_k domine son utilisation dans le ϕ-bloc, donc le point de définition de x_k domine pc ; on conclut comme au premier cas.
- **L'instruction en pc est une opération : $r \leftarrow x\ op\ y$.** Par correction (*soundness*) de l'interprétation abstraite, si on a sur-approximé les opérandes de op, alors $[\![x\ op\ y]\!]^\sharp lv \succeq [\![x\ op\ y]\!] rs$. Il suffit donc de montrer que $[\![x]\!]^\sharp ls \succeq [\![x]\!] rs$ et $[\![y]\!]^\sharp lv \succeq [\![y]\!] rs$. Il s'agit donc de montrer que $\mathrm{dsd}\, x\ pc$ (et de même pour y). Le programme étant bien formé, $x \neq r$, donc pc n'est pas le point de définition de x, or la définition de x domine pc, donc elle le domine strictement et on peut conclure.

4.3.2. *Correction de la transformation.* Le seul cas où le programme est modifié est : $[\![x\ op\ y]\!]^\sharp lv = k \in \texttt{int}$. Il faut alors justifier d'avoir remplacé $r \leftarrow x\ op\ y$ par $r \leftarrow k$.

Il s'agit de montrer que dans ce cas, $[\![x\ op\ y]\!] rs = k$. Or on sait par ce qui précède que $[\![x\ op\ y]\!]^\sharp lv \succeq [\![x\ op\ y]\!] rs$, et $\gamma(k) = \{k\}$, donc on a bien le résultat escompté.

De plus, on conserve les invariants SSA, puisqu'on ne change pas la forme du graphe de flot de contrôle et qu'on ne fait que supprimer des utilisations de registres, ce qui assure que les propriétés de domination sont conservées.

5. Sparse *Conditional* Constant Propagation (SCCP)

L'idée de cet algorithme, de Wegman et Zadeck[10], est de modifier SSCP afin de ne pas analyser les nœuds du graphes dont on peut prouver qu'ils ne seront jamais exécutés, ce qui arrive lorsque l'on dispose d'assez d'information pour déterminer statiquement quelle branche d'une condition sera prise. Cela permet d'une part d'accélérer l'analyse, puisque qu'on calcule un point fixe sur un graphe plus petit, et d'autre part d'améliorer sa précision dans le cas où une variable n'est modifiée que dans une branche morte (la variable x_1 figure 3).

5.1. **Algorithme.** On effectue notre calcul de point fixe de manière similaire à SSCP, mais seulement sur la partie du CFG dont on ne peut pas prouver qu'elle n'est pas morte.

Plus concrètement, chaque arête du graphe est munie d'une étiquette indiquant si elle est exécutable, et on dispose de deux *worklists* : l'une, CFGWorkList, contenant les arcs du CFG à examiner et l'autre, SSAWorkList, la liste des registres dont il faut propager les nouvelles valeurs de treillis à tous leurs points d'utilisations.

On initialise CFGWorkList pour ne contenir que l'arc entre l'entrée de la fonction et son unique fils (car c'est un `Inop`), puisqu'on est sûr qu'il est exécutable, et SSAWorkList à $\varnothing$. Les différences avec SSCP sont :

- Lorsqu'on traite un élément (pc, pc$'$) de CFGWorkList, on le marque comme exécutable, on visite toutes les instructions de pc$'$ (y compris les ϕ-instructions), et enfin, s'il n'a qu'un seul fils pc$''$, on ajoute (pc$'$, pc$''$) dans CFGWorkList.
- Lorsqu'on visite une ϕ-instruction $r \leftarrow \phi(x_1, \ldots, x_n)$, la nouvelle valeur de treillis de r est le *join* des valeurs de treillis des arguments, en ayant remplacé ceux qui correspondent à des arcs non exécutable par $\bot$. $\bot$ étant l'élément neutre de $\sqcup$, cela revient à considérer que les arcs non exécutables n'existent pas.
- Lorsqu'on visite une condition, on marque comme exécutable les branches dont on ne peut pas prouver qu'elles sont mortes.

La phase de transformation est exactement la même que pour SSCP.

Point fixe. L'algorithme calcule le couple (lv, Exec), où lv est similaire à celui de SSCP et Exec : `node`$\times$`node` $\to$ `bool` indique si un arc est susceptible d'être emprunté pendant l'exécution du programme.

(lv, Exec) est le plus petit point fixe du système suivant :

$$\begin{cases} [\![r]\!]^\sharp lv = [\![x \text{ op } y]\!]^\sharp lv \text{ pour } \{r \leftarrow x \ op \ y\} = \text{CFG}_f(pc) \text{ si Exec}(pc) \\ [\![r]\!]^\sharp lv = \bigsqcup [\![x_i^*]\!]^\sharp lv \text{ pour } \{r \leftarrow \phi(x_1, \cdots, x_k)\} = \text{CFG}_f(pc) \\ \text{Exec}(pc, pc') = \text{Exec}(pc) \wedge \text{ExecTransf}_{f,lv}(pc, pc') \end{cases}$$

où, en pc, $[\![x_i^*]\!]^\sharp lv$ vaut $\bot$ si l'arc entre le i-ème prédécesseur de pc et pc n'est pas exécutable, et $[\![x_i]\!]^\sharp lv$ sinon.

$\mathrm{ExecTransf}_{f,lv}$, à valeurs dans le treillis des booléens, est défini par :

$$\mathrm{ExecTransf}_{f,lv}(pc, pc') = \begin{cases} True & \text{si } pc' \text{ est l'unique successeur de } pc \\ [\![\texttt{cond args}]\!]^\sharp lv & \text{si } \mathrm{CFG}_f(\mathrm{pc}) = \{\texttt{Icond cond args pc' ifnot}\} \\ \neg[\![\texttt{cond args}]\!]^\sharp lv & \text{si } \mathrm{CFG}_f(\mathrm{pc}) = \{\texttt{Icond cond args ifso pc'}\} \end{cases}$$

5.2. **Ordre de parcours.** On extrait en priorité de CFGWorkList, l'intuition étant que plus on découvre vite la partie exécutable du graphe, moins on aura de variables à $\bot$ dans les instructions qu'on visite.

Afin d'accélérer la vitesse de convergence, SSAWorkList est en fait composée de deux listes, selon la valeur de treillis qu'a le registre lorsqu'on l'ajoute à la SSAWorkList : l'une pour $\top$ et l'autre pour les constantes (on n'ajoute jamais un registre dont la nouvelle valeur de treillis est $\bot$) ; et on extrait en priorité de la première.

Ainsi, lorsqu'on traite une instruction $r \leftarrow x\,op\,y$ suite à l'extraction de x de cette première liste, si y n'est pas à $\bot$, ce qu'on espère rare grâce à la première heuristique, r passera directement à $\top$, on peut ainsi ne traiter r une seule fois au lieu de deux.

5.3. **Preuve.** La preuve de SCCP ressemble beaucoup à celle de SSCP : on a ajouté à l'invariant *BA* l'hypothèse « *pc* est exécutable », ce qui signifie que l'on a rien à prouver pour les nœuds marqués non-exécutable. Cet invariant devient donc :

$$BA(lv, rs, pc) \equiv \forall r, \mathrm{dsd}\, r\, pc \Rightarrow \mathrm{Exec}(pc) \Rightarrow [\![r]\!]^\sharp lv \succeq [\![r]\!] rs.$$

Il est aisé de prouver que si le programme fait un pas d'un état *pc* à un état pc', et que $\mathrm{Exec}(pc)$ alors $\mathrm{Exec}(pc')$ à l'aide du théorème de correction de l'interprétation abstraite des conditions dans le treillis des constantes, déjà présent dans CompCert :

Théorème 5. Soit `args` une liste de variables, alors

$$(\forall \texttt{arg} \in \texttt{args}, [\![\texttt{arg}]\!]^\sharp rs \succeq [\![\texttt{arg}]\!] rs) \Rightarrow [\![\texttt{cond args}]\!]^\sharp lv \succeq [\![\texttt{cond args}]\!] rs.$$

6. Évaluation

6.1. **Développement Coq.** L'approche choisie est de valider *a posteriori* que le résultat retourné par l'algorithme est bien un post point-fixe — car c'est plus simple et que sous sa forme actuelle, il est peu probable que le solveur de point fixe qui a été implémenté puisse être réutilisé en l'état pour d'autres analyses — et de prouver directement la transformation.

J'ai d'abord implémenté et prouvé SSCP, puis SCCP. La preuve de ce dernier est basée sur celle SSCP, mais l'implémentation de l'algorithme est indépendante. L'interprétation abstraite sur le treillis des constantes, et les théorèmes de correction associées, utilisés par CompCert pour Constprop ont été réutilisés.

Pour SCCP, l'implémentation et les spécifications font 792 lignes de code et la preuve tient en 1229 lignes.

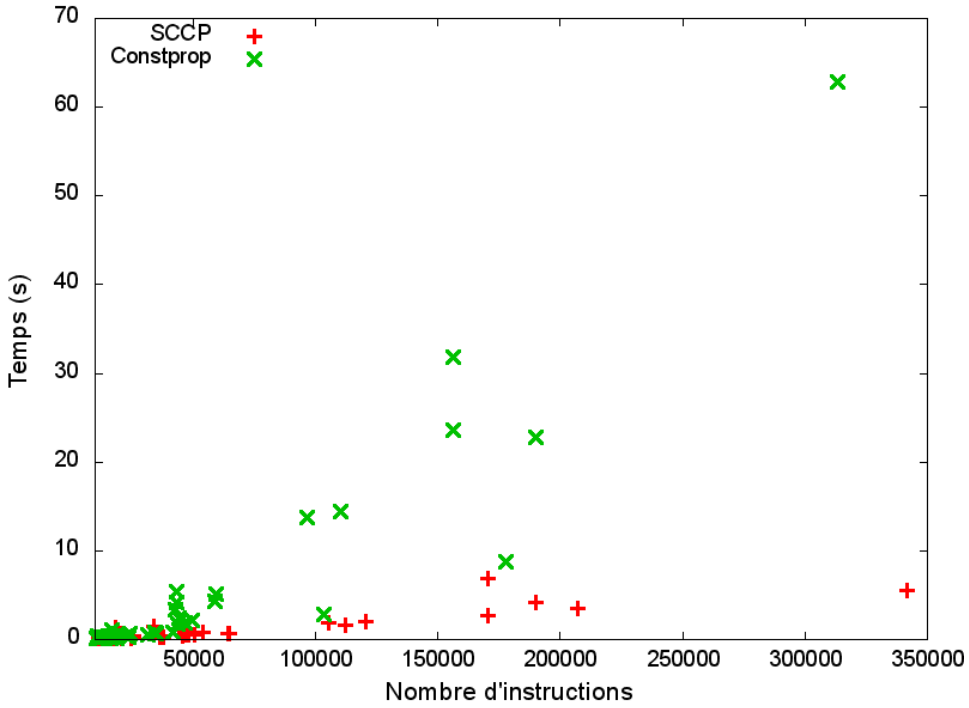


FIGURE 4. Comparaison en temps de SCCP et Constprop

6.2. **Protocole expérimental.** Les tests qui suivent ont été fait en compilant SPASS[11], un démonstrateur automatique de théorèmes de l'institut Max Planck, qui consiste en 51000 lignes de C, et qui est le plus gros exemple publiquement disponible de code C compilable par CompCert. De plus, lorsqu'on a voulu tester le temps de compilation, afin de faire tourner notre algorithme sur de plus grosses fonctions (pour les fonctions de SPASS, les analyses de constantes dépassent rarement le dixième de seconde), on a modifié le compilateur pour qu'il *inline* autant que possible, la passe d'*inlining* précédant le passage en forme SSA.

6.3. **Précision.** De manière générale, les trois analyses (Constprop, SSCP et SCCP) ont la même précision.

Sur SPASS, la passe de CompCert remplace 23882 expressions par des constantes alors que SSCP en remplace 23893, soit une différence relative d'environ $0,04\%$.

Pour comparer SSCP et SCCP, il faut se restreindre aux expressions qui ont été simplifiées dans des nœuds marqués par SCCP comme exécutable, puisque SCCP ne visite pas les autres nœuds. Pour SPASS, toutes les constantes « exécutables » trouvées par SSCP on été trouvées par SCCP, et 13 constantes supplémentaires ont été trouvées par SCCP.

6.4. **Temps pris par l'analyse.** Dans la figure 4, il y a en abscisse la taille en nombre d'instruction des fonctions, et en ordonnée, pour SCCP et Constprop, la passe de propagation de constante de CompCert, le temps que chaque passe met à traiter une fonction. On peut remarquer que les versions

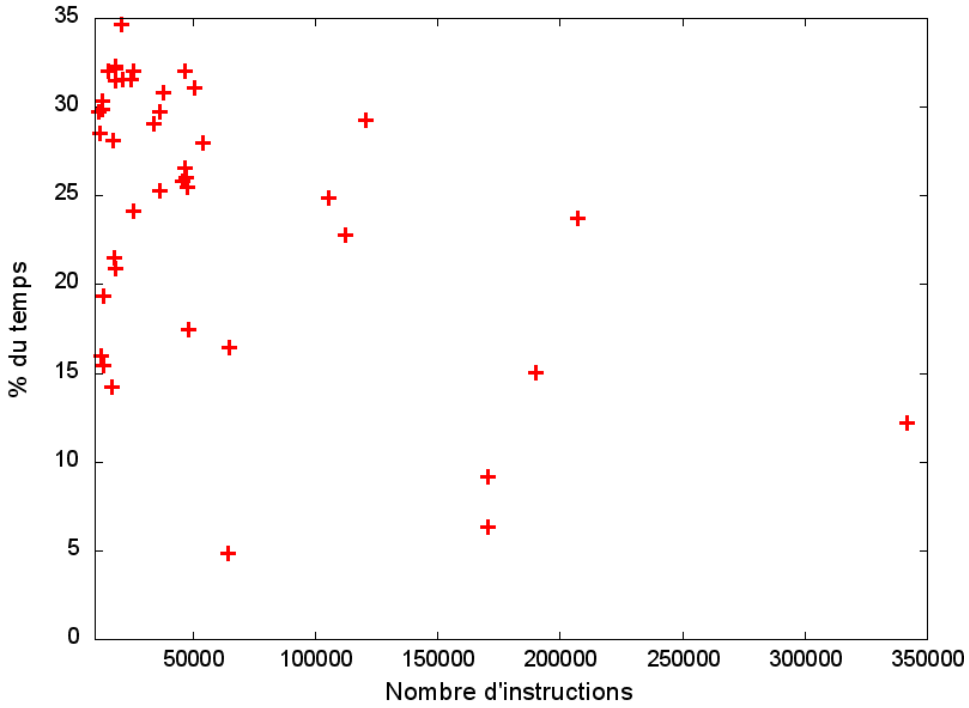


FIGURE 5. Pourcentage du temps pris par le validateur par rapport à la durée totale de SCCP

`RTL` et `SSA` d'une même fonction ne sont pas parfaitement alignées verticalement, cela est dû au fait que lors du passage de `RTL` à `SSA`, CompCertSSA rajoute des instructions (lors de la normalisation).

On voit que l'algorithme sur SSA passe beaucoup mieux à l'échelle, ce qui est l'un des intérêts majeurs de SSA.

6.5. **Temps pris par le validateur.** En abscisse de la figure 5, la taille des CFG, et en ordonnée le pourcentage du temps pris par le validateur par rapport au temps total de SCCP. On voit qu'environ un tiers du temps est consacré à la validation, ce qui tout à fait acceptable et cohérent avec le fait que l'analyse visite au plus chaque instruction à deux reprises.

7. TRAVAUX CONNEXES

Il existe deux formalisations de la forme SSA : CompCertSSA[2] et Vellvm[12]. À la différence de CompCertSSA, Vellvm ne s'intègre pas dans un compilateur certifié mais dans LLVM. Velllum est une formalisation, aussi en Coq, d'un sous-ensemble de la représentation intermédiaire de LLVM (qui est sous forme SSA). Leur formalisation de SSA est proche de celle de CompCertSSA : les invariants sont très similaires et ils utilisent un énoncé similaire au lemme équationnel (théorème 4).

Dans [13], Zhao *et al* ont implémenté une version certifiée de la passe `mem2reg` de LLVM : en général, les *front-ends* (par exemple le compilateur C Clang) qui utilisent LLVM représentent les variables de leurs langages par des emplacements mémoires, qui sont manipulés par des `load` et des `store`.

La passe `mem2reg` sert à transformer certains emplacements mémoires, ceux dont on ne prend jamais l'adresse (par exemple, en C, avec l'opérateur `&`), par des variables SSA. Elle est très importante car d'une part accéder à un registre dans le processeur est beaucoup plus rapide qu'un accès mémoire, et d'autre part parce que les passes d'optimisation ne fonctionnent souvent que sur les variables, car il n'y a pas de possibilité d'*aliasing*.

Ainsi, `mem2reg` combine deux passes de CompCertSSA : la promotion de la mémoire vers les registres (qui fait partie de CompCert) et le passage en forme SSA. Contrairement à CompCertSSA, qui utilise la validation a posteriori pour le passage en forme SSA, la transformation est prouvée directement, mais elle utilise un algorithme moins efficace, quadratique en la taille du programme, alors que celui de CompCertSSA est quasi-linéaire.

8. Conclusion et perspectives

Pendant ce stage, j'ai implémenté et prouvé une passe d'optimisation, SCCP, aussi précise que la passe analogue de CompCert, mais ayant de meilleures performances pour les CFG de grandes tailles.

L'optimisation qui a été implémentée dans CompCertSSA lors de ce stage est fidèle à l'algorithme de Wegman et Zadek[10]. Décrit pour la première fois en 1985, il correspond néanmoins à ce que font les compilateurs à l'état de l'art : par exemple, LLVM a une passe SCCP qui est très comparable.

Une extension possible, qui est discutée dans [10], est de rendre notre analyse interprocédurale, ce qui permettrait par exemple de déterminer que certaines variables globales sont constantes pendant l'exécution du programme, ou que certaines fonctions retournent toujours la même valeur.

Enfin la preuve de SCCP et de GVN sont remarquablement similaires : toutes deux reposent sur des propriétés du graphe SSA (par exemple, elles utilisent dsd dans leurs invariants), et dans les deux cas une partie de la preuve consiste à transporter des propriétés du graphe SSA vers le CFG, puisque l'induction sur la sémantique suit les arcs du CFG. Il semble qu'il devrait être possible de prouver la correction de ces optimisations uniquement par des arguments sur le graphe SSA.

Dans [5], Lerner *et al.* décrivent un framework qui permet d'entrelacer plusieurs optimisations afin de profiter d'éventuelles synergies entre celles-ci (une transformation faite par une optimisation rend possible une autre optimisation) et qui garantie la correction de l'optimisation combinée à condition que chaque optimisation vérifie un théorème de correction local. L'idée est chaque optimisation modifie le graphe de manière spéculative dès qu'elle peut le justifier d'après son environnement abstrait ; c'est par ce mécanisme que les différentes optimisations peuvent communiquer implicitement entre elles. On pourrait s'appuyer sur cet article pour concevoir un framework dont le théorème de correction serait, lui, formellement prouvé, qui fonctionnerait sur le graphe SSA au lieu du CFG comme décrit dans [5]. Chaque optimisation indiquerait en plus au framework qu'une branche est impossible afin de réduire le graphe exploré, et par le même mécanisme que pour SCCP, les optimisations ne verraient que la partie du graphe qui est exécutable. Des optimisations telles que SCCP et GVN seraient exprimable dans un tel framework. Si en plus on y ajoutait quelques techniques d'interprétations

abstraites telles que *l'élargissement*[3], une analyse d'intervalle pourrait permettre de détecter plus de branches mortes.

Références

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[2] Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified ssa-based middle-end : Static single assignment meets compcert. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 47–66, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

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

[5] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 270–282, New York, NY, USA, 2002. ACM.

[6] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[7] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[8] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[9] Fabrice Rastello. *SSA-based Compiler Design*. à paraître.

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

[11] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 140–145, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1) :427–440, January 2012.

[13] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for llvm. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 175–186, New York, NY, USA, 2013. ACM.