Minigloca

Vladislas de Haldat
Supervisé par Pierre-Evariste Dagand

3 juin 2023

Table des matières

1 Introduction

Le projet Gloca tient pour but initial d'offrir au programmeur un plus ample contrôle sur la compilation de son programme. Il s'agit de remettre entre ses mains les optimisations qui sont aujourd'hui largement prises en charge par les compilateurs et parfois, au dépend des performances. L'optimisation en compilation repose en partie, depuis les années 1960, sur l'analyse statique de code. Cette dernière permet en quelque sorte de prévoir les différents comportements qui pourraient être émis par un programme. À partir de là, le compilateur doit agir en conséquence et optimise ainsi le code assembleur produit. Chaque analyse a sa propriété particulière à respecter. Par exemple, si le programme contient du code mort, c'est-à-dire du code qui ne sera jamais appelé quelque soit la manière avec laquelle il s'exécute, il n'est pas besoin de le traiter à la compilation. Si le programme doit recalculer une même expression plusieurs fois, au sein d'une boucle par exemple, alors la calculer une seule fois, la stocker en mémoire, puis appeler le résultat obtenu quand nécessaire permet d'économiser énormément d'opérations.

Cependant, il est des optimisations que le compilateur est incapable de réaliser car il existe des cas qu'il ne peut pas prévoir. Certaines optimisations peuvent aussi parfois ne pas servir du tout et au contraire ralentir le programme. Ces cas de figure peuvent s'avérer problématique lorsque qu'on attend d'un programme qu'il réalise de gros calculs en tenant compte d'importantes contraintes, typiquement au sein de systèmes embarqués. Ce projet consistera donc à formaliser un langage impératif expérimental, sur lequel l'on développera une première analyse statique de deux manières différentes. Cela permettra ensuite d'appliquer une généralisation et de formuler une extension de ce langage, laquelle agira à la pré-compilation comme un meta-langage. Pour ce faire, on aura besoin d'établir certaines relations entre les prédicats qui composeront cette extension du langage et les analyses, qui seront appliquées en conséquence. Ces relations permettront enfin de donner un cadre efficace à l'utilisation des prédicats par le programmeur.

Donc, est-il possible de formaliser un langage qui puisse, se réécrire lors de la pré-compilation en fonction de prédicats, lesquels invoqueront certaines analyses statiques sur le programme ? Comment ces analyses statiques se définissent-elles à partir des prédicats ? Ces analyses pourront-elles, en outre, se calculer de manière incrémentale, c'est-à-dire en réutilisant leur dernier résultat ?

2 Un langage impératif simple

Pour commencer, définissons une syntaxe abstraite minimale que nous utiliserons tout le long de cette étude. Cette syntaxe se composera de trois blocs fondamentaux que sont les expressions arithmétiques, les expressions booléennes ainsi que les déclarations. Nous ne considèrerons pas pour le moment la définition de routines au sein de cette syntaxe.

2.1 Expressions arithmétiques

Les expressions arithmétiques sont définies sur l'ensemble des entiers relatifs. On se donne les opérateurs de l'addition, de la soustraction ainsi que de la multiplication. À ces opérateurs l'on pourra appliquer des entiers ainsi que des identifiants de variables.

$$\begin{aligned} Int &::= n && n \in \mathbb{Z} \\ Id &::= x && x \in \mathbb{V} \\ a &::= Int \mid Id \mid op_A(a_1, a_2) && op_A \in \{+, -, \times\} \end{aligned}$$

FIGURE 1 – Expression arithmétique

$\mathbb{V}$ décrira désormais l'ensemble des identifiants de variables affectées dans nos programmes.

2.2 Expressions booléennes

Les expressions booléennes nous permettent d'introduire la comparaison entre deux expressions arithmétiques, ainsi que les opérateurs booléens sur les expressions booléennes.

$$\begin{aligned} b ::= & \textbf{ true} \mid \textbf{false} \\ & \mid op_R(a_1, a_2) && op_R \in \{<, =\} \\ & \mid op_B(b_1, b_2) && op_B \in \{\wedge, \vee\} \\ & \mid \neg b \end{aligned}$$

FIGURE 2 – Expression booléenne

2.3 Déclarations

Les déclarations nous permettent de donner forme à notre langage en définissant la séquence de deux instructions ainsi que les gardes booléennes, sur une condition ou une boucle. On pose la déclaration de la manière suivante,

$$\begin{aligned} Stm ::= & \; x := a \\ & \mid s_1; s_2 \\ & \mid \textit{skip} \\ & \mid \text{if } b \text{ then } s_1 \text{ else } s_2 \\ & \mid \text{while } b \text{ do } s \\ & \mid \text{return } a \end{aligned}$$

La déclaration return servira pour le moment à simuler l'appel à une routine, ce qui permettra de donner plus de sens aux applications de nos analyses. Dans le reste de cette étude, on parlera également de déclaration en désignant un programme étant donné que les analyses statiques n'ont pas besoin d'évaluer la

déclaration pour se construire.

Exemple.
Voici un premier exemple sur ce langage

```
a := 1;
b := 20;
if a = 3 then
  c := 4
else
  c := 6
endif;
while b < 100 do
  a := b + 1
done;
return c
```

L'implémentation de tout ce qui est spécifié dans ce document, est écrite en OCaml et est disponible sur le dépôt Git [1].

2.4 Arbre de syntaxe abstraite

De manière à construire nos programmes à partir de ce qui a été précédemment énoncé, il est nécessaire d'introduire l'arbre de syntaxe abstraite. Ce modèle de représentation permet de parcourir la déclaration et d'en extraire n'importe quelle fonctions (dans notre cas les différents opérateurs arithmétiques, booléens, ou d'affectation). Cela nous servira particulièrement lors des différentes analyses qui seront effectuées, et notamment l'analyse par flot de contrôle. Cette dernière ne requièrant pas la connaissance de l'ordre d'exécution de la déclaration, les AST sont tout-à-fait adaptés.

3 Interpréteur et sémantique

Dans cette section, on s'atèle à décrire l'interpréteur ainsi que la sémantique sur notre petit langage impératif. Rappelons que $\mathbb{V}$ est l'ensemble des variables affectées dans un programme. Dans ce cadre là, on définit l'état d'un programme par une application entre les identifiants de variables et leurs valeurs, dans notre cas, des entiers relatifs,

$$\sigma : \mathbb{V} \longrightarrow \mathbb{Z}$$

Pour la suite, on considèrera l'ensemble des états $\mathcal{S} = \mathbb{Z}^{\mathbb{V}}$. Ces derniers décriront l'évolution de l'exécution du programme et nous permettront de formaliser un interpréteur.

3.1 Interpréteur

Pour le moment, il n'est pas nécessaire d'implémenter un compilateur pour notre langage, un interpréteur suffira. Celui-ci servira notamment à valider les tests unitaires lorsque les premières analyses statiques seront appliquées à nos programmes. Sur les expressions arithmétiques, définies dans la section précédente,

1. https ://github.com/pedagand/minigloca

on pose l'application suivante,

$$\begin{aligned}
[\![a]\!]^A : \mathcal{S} &\longrightarrow \mathbb{Z} \\
[\![n]\!]^A \sigma &\longmapsto n \\
[\![x]\!]^A \sigma &\longmapsto \sigma(x) \\
[\![op_A(a_1, a_2)]\!]^A \sigma &\longmapsto \hat{op}_A([\![a_1]\!]^A(\sigma),\ [\![a_2]\!]^A(\sigma))
\end{aligned}$$

On définit $\mathbb{B} = \{0, 1\}$. De la même manière, on définit l'application qui suit sur les expressions booléennes,

$$\begin{aligned}
[\![b]\!]^B : \mathcal{S} &\longrightarrow \mathbb{B} \\
[\![\mathbf{true}]\!]^B \sigma &\longmapsto \top \\
[\![\mathbf{false}]\!]^B \sigma &\longmapsto \bot \\
[\![op_R(a_1, a_2)]\!]^B \sigma &\longmapsto \hat{op}_R([\![a_1]\!]^A(\sigma),\ [\![a_2]\!]^A(\sigma)) \\
[\![op_B(b_1, b_2)]\!]^B \sigma &\longmapsto \hat{op}_B([\![b_1]\!]^B(\sigma),\ [\![b_2]\!]^B(\sigma)) \\
[\![\neg b]\!]^B \sigma &\longmapsto \hat{\neg}[\![b]\!]^B(\sigma)
\end{aligned}$$

Les opérateurs de la forme $\hat{op}$ représentent les opérateurs natifs à OCaml.

3.2 Sémantique

Maintenant que nous avons correctement défini l'interpréteur, il est possible de construire la sémantique du langage. Cela permet également de développer un ensemble de règles logiques. La syntaxe de la déclaration des règles, prenant en compte nos état et déclaration, sera donc de la forme suivante,

$$s, \Sigma \longrightarrow s', \Sigma'$$

où s est la première déclaration, s' celle qui suit après son exécution, Σ l'état initial et Σ' l'état successeur. Commençons par la déclaration vide,

$$\frac{}{\mathit{skip}, \sigma \longrightarrow \emptyset, \sigma}$$

Poursuivons avec la déclaration de l'affectation,

$$\frac{}{x := a, \sigma \longrightarrow \emptyset, \sigma'[x \longmapsto [\![a]\!]^A(\sigma)]}$$

Ici, le nouvel état σ' lie l'identifiant de la variable assignée, à la valeur de l'expression arithmétique $[\![a]\!]^A$. Poursuivons avec la règle de la séquence entre deux déclarations, d'une part si la première termine,

$$\frac{s_1, \sigma \longrightarrow \emptyset, \sigma'}{s_1; s_2, \sigma \longrightarrow s_2, \sigma'}$$

D'autre part, si la première ne termine pas,

$$\frac{s_1, \sigma \longrightarrow s_1', \sigma'}{s_1; s_2, \sigma \longrightarrow s_1'; s_2, \sigma'}$$

La condition peut se formaliser de la sorte dans le cas où la garde est vérifiée,

$$\frac{[\![b]\!]^B(\sigma)}{\text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \longrightarrow s_1, \sigma}$$

Dans le cas où elle ne l'est pas,

$$\frac{\neg[\![b]\!]^B(\sigma)}{\text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \longrightarrow s_2, \sigma}$$

On peut déclarer la règle qui suit pour la déclaration while,

$$\frac{[\![b]\!]^B(\sigma)}{\text{while } b \text{ do } s, \sigma \longrightarrow s; \text{while } b \text{ do } s, \sigma}$$

Enfin, pour l'opération return on donne la règle,

$$\frac{}{\text{return } a, \sigma \longrightarrow \emptyset, \sigma}$$

4 Prérequis à l'analyse

De manière à pouvoir travailler avec les analyses de flot de données et de flot de contrôle, il est nécessaire d'approfondir les principes de bloc et d'étiquettage. Ceux-ci nous fourniront une bonne représentation de nos programmes, sur laquelle les analyses devront reposer.

4.1 Blocs

De manière à faciliter l'analyse sur une déclaration, il est utile de partitionner et de factoriser notre représentation du code. Les blocs peuvent être perçus comme un concentré des déclarations atomiques dans un programme. Par exemple, les conditions ainsi que les boucles reposent toutes deux sur une garde booléenne, on peut donc les factoriser. Étant donné l'aspect rudimentaire de notre langage, on peut simplement définir le bloc de la sorte,

$$\begin{aligned} \mathit{Block} ::= {} & x := a \\ \mid {} & b \\ \mid {} & \mathit{skip} \\ \mid {} & \text{return } a \end{aligned}$$

On remarquera que le bloc est défini de manière atomique. Cependant il est tout-à-fait possible de considérer des séquences d'instructions comme un bloc à part entière, ce qui pourra dans certains cas largement simplifier la représentation du flot de contrôle.

4.2 Étiquettage

De manière à correctement travailler avec ces blocs, il faut avoir une information sur leur position relative par rapport aux autres blocs. Il est donc nécessaire de

les coupler chacun à un identifiant unique. On définit $\mathbb{L} \subseteq \mathbb{N}$ l'ensemble fini des étiquettes d'une déclaration. Introduisons l'application suivante,

$$\lambda : Block \longrightarrow \mathbb{L}$$

qui prend un bloc et retourne une étiquette unique par rapport au reste des blocs sur une déclaration.

Définition – Soient $s \in Stm$ et $(l_n)_{n\in\mathbb{N}}$ une suite d'étiquettes construite par λ sur s. On dit que λ est bien formée si et seulement si $\forall i \in \mathbb{N}, \forall j \in \mathbb{N}$ tel que $i \neq j$ on a $l_i \neq l_j$.

À terme, on voudrait représenter le programme par un graphe (le graphe de flot de contrôle) dont chaque noeud représenterait un bloc. Étant donné une déclaration s, on a pour b un bloc de s, $\delta_-(b) = 1$ le degré entrant et $\delta_+(b) \geq 1$, le degré sortant du noeud correspondant dans le graphe de flot de contrôle.

Notation – Pour des raisons de commodité, si $s \in Stm$ et $l = \lambda(s)$ alors on notera $s^l \in Stm$ la déclaration munie d'une étiquette. Si la déclaration est une condition ou une boucle, l'étiquette sera rattachée à la garde de ces dernières.

On peut maintenant définir l'application $init$, qui retournera la première étiquette rencontrée dans une déclaration,

$$\begin{aligned}
init : Stm &\longrightarrow \mathbb{L} \\
(x := a)^l &\longmapsto l \\
s_1; s_2 &\longmapsto init(s_1) \\
skip^l &\longmapsto l \\
\text{if } b^l \text{ then } s_1 \text{ else } s_2 &\longmapsto l \\
\text{while } b^l \text{ do } s &\longmapsto l \\
(\text{return } a)^l &\longmapsto l
\end{aligned}$$

Comme expliqué précédemment, il est aussi nécessaire de déclarer une fonction $final$ qui retournera l'ensemble des étiquettes finales à la fin d'un bloc.

$$\begin{aligned}
final : Stm &\longrightarrow \mathcal{P}(\mathbb{L}) \\
(x := a)^l &\longmapsto \{l\} \\
s_1; s_2 &\longmapsto final(s_2) \\
skip^l &\longmapsto \{l\} \\
\text{if } b^l \text{ then } s_1 \text{ else } s_2 &\longmapsto final(s_1) \cup final(s_2) \\
\text{while } b^l \text{ do } s &\longmapsto \{l\} \\
(\text{return } a)^l &\longmapsto \{l\}
\end{aligned}$$

4.3 Flots

Définition – Un graphe de flot de contrôle (CFG) est un graphe orienté dont les noeuds représentent les blocs et les arcs un flot de contrôle (de même orientation que l'exécution du programme). Dans notre cas, le graphe de flot de contrôle aura un seul noeud d'entrée et un seul noeud de sortie.

En voici quelques exemples,

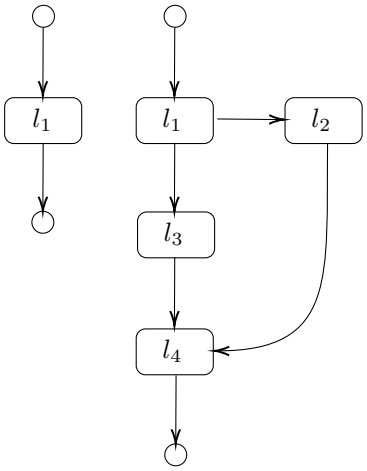

FIGURE 3 – Graphe de flot de contrôle

De manière à correctement identifier chaque bloc lors d'une analyse statique, il nous faut les lier à une étiquette unique. Pour ce faire, on pose l'application *blocks* définie par

$$
\begin{aligned}
\mathit{blocks} : \mathit{Stm} &\longrightarrow \mathcal{P}(\mathbb{L} \times \mathit{Block}) \\
(x := a)^l &\longmapsto \{(l, x := a)\} \\
s_1; s_2 &\longmapsto \mathit{blocks}(s_1) \cup \mathit{blocks}(s_2) \\
\mathit{skip}^l &\longmapsto \{(l, \mathit{skip})\} \\
\text{if } b^l \text{ then } s_1 \text{ else } s_2 &\longmapsto \{(l, b)\} \cup \mathit{blocks}(s_1) \cup \mathit{blocks}(s_2) \\
\text{while } b^l \text{ do } s &\longmapsto \{(l, b)\} \cup \mathit{blocks}(s) \\
(\text{return } a)^l &\longmapsto \{(l, \text{return } a)\}
\end{aligned}
$$

À partir de là, il est possible de formaliser les graphes orientés de flot. Comme dit précédemment, les blocs en représentent les noeuds, et le passage vers le bloc suivant est formulé par un arc.
Nous avons désormais toutes les structures nécessaires à la construction de notre graphe de flot. Pour le moment, il s'agira de le construire naïvement, l'on reviendra plus tard sur les optimisations possibles. Ainsi, une manière simple de représenter ce graphe est de considérer l'ensemble des couples d'étiquettes qui indiqueront un arc d'un bloc à un autre. Considérons donc l'application suivante,

$$
\begin{aligned}
flow : Stm &\longrightarrow \mathcal{P}(\mathbb{L}^2) \\
x := a &\longmapsto \emptyset \\
skip &\longmapsto \emptyset \\
s_1; s_2 &\longmapsto flow(s_1) \cup flow(s_2) \cup [final(s_1) \times \{init(s_2)\}] \\
\text{if } b^l \text{ then } s_1 \text{ else } s_2 &\longmapsto flow(s_1) \cup flow(s_2) \cup \{(l, init(s_1)), (l, init(s_2))\} \\
\text{while } b^l \text{ do } s &\longmapsto flow(s) \cup \{(l, init(s))\} \cup [final(s) \times \{l\}] \\
\text{return } a &\longmapsto \emptyset
\end{aligned}
$$

On introduira aussi un accès à l'ensemble des successeurs d'un bloc par l'application,

$$
\begin{aligned}
succ : \mathbb{L} &\longrightarrow \mathcal{P}(\mathbb{L}) \\
l &\longmapsto \{l' \in \mathbb{L} \mid (l, l') \in \mathcal{G}_V\},
\end{aligned}
$$

où $\mathcal{G}_V$ est l'ensemble des arcs du graphe de flot de contrôle du programme. Réciproquement, on utilisera *pred* qui donnera cette fois-ci accès aux prédecesseurs d'un bloc.

5 Analyse de vivacité

La première analyse statique sur laquelle nous travaillerons est l'analyse de vivacité des variables dans nos programmes. Il s'agira donc de déterminer pour chaque bloc, l'ensemble de variables encore vivantes, c'est-à-dire encore utilisées une fois le présent bloc passé. Pour cela, il faut au préalable déclarer quelques ensembles nécessaires à cette analyse.

5.1 Description ensembliste d'un programme

On se donne $(\mathcal{P}(\mathbb{V}), \subseteq)$ l'ensemble partiellement ordonné des parties de $\mathbb{V}$, le treillis sur lequel on travaillera désormais.
Pour $l \in \mathbb{L}$ l'étiquette d'un bloc, on définit les ensembles qui suivent,

$$gen[l] \subseteq \mathcal{P}(\mathbb{V})$$

l'ensemble des variables appelées dans le bloc en question et,

$$kill[l] \subseteq \mathcal{P}(\mathbb{V})$$

l'ensemble des variables nouvellement affectées, donc considérées pour lors comme mortes. Enfin on se donnera,

$$vars_a : a \longrightarrow \mathcal{P}(\mathbb{V})$$

l'ensemble des identifiants de variables présentes dans une expression arithmétique et,

$$vars_b : b \longrightarrow \mathcal{P}(\mathbb{V})$$

l'ensemble des identifiants de variables présentes dans une expression booléenne. Définissons maintenant la construction de ces deux ensembles à partir d'un bloc comme,

$$\begin{aligned} gen : Block &\longrightarrow \mathcal{P}(\mathbb{V}) \\ x := a &\longmapsto vars_a(a) \\ skip &\longmapsto \emptyset \\ b &\longmapsto vars_b(b) \\ \text{return } a &\longmapsto vars_a(a) \end{aligned}$$

génère l'ensemble $gen[l]$ et,

$$\begin{aligned} kill : Block &\longrightarrow \mathcal{P}(\mathbb{V}) \\ x := a &\longmapsto \{x\} \\ skip &\longmapsto \emptyset \\ b &\longmapsto \emptyset \\ \text{return } a &\longmapsto \emptyset \end{aligned}$$

génère l'ensemble $kill[l]$.

Pour pouvoir donner forme à cette analyse, on déclare deux ensembles de variables vivantes à l'entrée, et à la sortie d'un bloc. Ils nous permettront par la suite de résoudre un système d'équation par itération et d'y trouver un point fixe. On les définira comme tels,

$$LIVE_{out}[l] = \begin{cases} \emptyset & \text{si } succ(l) = \emptyset\,, \\ \bigcup\limits_{p \in succ(s)} LIVE_{in}[p] & \text{sinon}\,, \end{cases}$$

l'ensemble des variables vivantes à la sortie d'un bloc et,

$$LIVE_{in}[l] = gen[l] \cup (LIVE_{out}[l] - kill[l]),$$

l'ensemble des variables vivantes à l'entrée d'un bloc.

Notation – Si Z est une solution du système d'équation sur un treillis $(L, \sqsubseteq)$, on notera aussi $Z_l \subseteq L$ la solution au bloc d'étiquette l.

À ce stade, nous ne pouvons pas encore effectuer l'analyse de vivacité sur nos programmes, il nous manque en effet un algorithme d'itération qui puisse résoudre le système d'équation qui est un point fixe en chaque bloc. La finalité étant de trouver le point fixe minimal.

5.2 Monotonie

La monotonie des ensembles de flot de données nous permettra de garantir ou non que le point fixe déterminé par un algorithme est le point fixe minimal et que cet algorithme termine correctement. Cela sera donc utile pour prouver la terminaison des algorithmes, étant donné que l'ensemble des variables $\mathbb{V}$ d'une déclaration est supposé fini.

Lemme – Pour tout $l \in \mathbb{L}$ alors $LIVE_{in}[l]$ et $LIVE_{out}[l]$ sont monotones.

Démonstration – Posons les applications,

$$\begin{aligned} f : \mathcal{P}(\mathbb{V}) &\longrightarrow \mathcal{P}(\mathbb{V}) \\ X_l &\longmapsto gen[l] \cup (g(X_l) - kill[l]) \end{aligned}$$

où

$$\begin{aligned} g : \mathcal{P}(\mathbb{V}) &\longrightarrow \mathcal{P}(\mathbb{V}) \\ X_l &\longmapsto \bigcup_{p \in succ(l)} f(X_p) \end{aligned}$$

Soient $K_l, K'_l \in \mathcal{P}(\mathbb{V})$ deux analyses de même étiquette sur deux déclarations différentes au bloc d'étiquette l sur $s \in Stm$ une déclaration. Elles sont telles que $K_l \subseteq K'_l$. On a alors,

$$g(K_l) \subseteq g(K'_l)$$

comme g ne traite que les blocs successeurs à ce bloc d'étiquette l, qui sont donc les mêmes. Ensuite on a $gen_K[l] \subseteq gen_{K'}[l]$ (par monotonie croissante de l'union) mais aussi $kill_{K'}[l] \subseteq kill_K[l]$ (par monotonie décroissante de la différence ensembliste) et d'après le précédent résultat, $g(K_l) \subseteq g(K'_l)$. Donc, par monotonie croissante de l'union on a bien

$$f(K_l) \subseteq f(K'_l)$$

Ainsi $LIVE_{in}[l]$ et $LIVE_{out}[l]$ sont monotones croissantes.

□

5.3 Point fixe

Revenons-en à l'existence d'un point fixe, pour pouvoir construire un algorithme itératif. On dispose que $(\mathcal{P}(\mathbb{V}), \subseteq)$ est un ensemble muni d'un ordre partiel et est fini. Ainsi, grâce à la monotonie de nos deux ensembles $LIVE_{in}[\cdot]$ et $LIVE_{out}[\cdot]$, démontrée ci-dessus, il vient qu'à chaque itération de notre algorithme, l'ensemble produit sera soit identique à l'ensemble précédent, soit plus gros et la finitude de l'ensemble des variables assure qu'il ne pourra par grossir indéfiniment. Il faut cependant aussi avoir la garantie que la solution trouvée soit la plus petite qui puisse exister. On se sert pour y parvenir du théorème du point fixe de Kleene.

Théorème – Soit $(L, \sqsubseteq)$ un ordre partiellement ordonné, avec un plus petit élément $\bot$ et soit une application $f : L \longrightarrow L$ monotone. Alors il existe un point fixe minimal qui est le suprémum de la suite,

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots \sqsubseteq f^k(\bot) \sqsubseteq \cdots$$

Démonstration – [admise]

□

Commençons par établir un algorithme de point fixe naïf. Il s'agira simplement de recalculer l'entièreté du système d'équation à chaque itération, jusqu'à trouver le point fixe minimal (par le théorème de Kleene).

Algorithme 1 Itération du point fixe

Soit s la déclaration
$\mathcal{B} \leftarrow blocks(s)$
Soit $\mathbb{L}$ l'ensemble des étiquettes de $\mathcal{B}$
for $l \in \mathbb{L}$ **do**
    $live_{in}[l] \leftarrow \emptyset$
    $live_{out}[l] \leftarrow \emptyset$
end for
while $live_{in} \neq live'_{in}$ **ou** $live_{out} \neq live'_{out}$ **do**
    for $l \in L$ **do**
        $live_{out}[l] \leftarrow \bigcup_{p \in succ(l)} live_{in}[p]$
        $live_{in}[l] \leftarrow gen[l] \cup (live_{out}[l] - kill[l])$
    end for
end while

Si $n = \#\mathbb{L}$, alors cet algorithme a une complexité en $O(kn)$, où k est la hauteur du diagramme de Hasse sur $(\mathcal{P}(\mathbb{V}), \sqsubseteq)$. Cela se montre grâce à la monotonie de $LIVE_{in}[\cdot]$, celle-ci ne faisant que croître vers $\top$. Concernant l'implémentation, il peut être intéressant de définir $live_{in}$ et $live_{out}$ comme une structure binaire de taille au moins m bits avec $m = \#\mathbb{V}$. En outre, une amélioration peut être simplement faite sur l'algorithme décrit ci-dessus en remarquant qu'à chaque modification effective de $LIVE_{out}[\cdot]$, seuls les blocs prédecesseurs seront susceptibles de s'altérer. Cela vient du fait que l'analyse se fait de bas en haut. Donc au lieu d'itérer à nouveau sur l'ensemble des blocs, il suffit d'itérer uniquement sur les blocs, prédecesseurs au dernier bloc altéré. Cet algorithme a pour nom naturel de worklist, on le décrira plus bas dans la section.

Lemme – Étant donné $s \in Stm$ une déclaration, à partir de $\bot$ l'algorithme d'itération du point fixe trouve effectivement un point fixe sur s et termine.

Démonstration – Les correction et terminaison de l'algorithme reposent en partie sur ce qui a été dit en amont de cette section. On utilise deux fonctions croissantes monotones. Posons $f = live_{in}$ étant donné qu'on peut se restreindre à l'étude de $LIVE_{in}[\cdot]$. À chaque itération deux cas s'offrent,

1. si on atteint un point fixe pour tout $l \in \mathbb{L}$, on a $f^k(\bot)$ stationnaire à partir de la k-ème itération. De plus, par monotonie de f, c'est le suprémum des $f^n(\bot)$ pour tout $n \in \mathbb{N}$ donc l'algorithme a trouvé le point fixe minimal, par le théorème de Kleene.
2. sinon, le treillis a crû, par croissance monotone de f. Étant donné qu'on suppose $\mathbb{V}$ fini, le treillis continuera de croître jusqu'à atteindre le premier cas évoqué, ou alors le cas $f^k(\bot) = \top$ pour tout $k \geq N \in \mathbb{N}$.

Ainsi l'algorithme termine et trouve bien le plus petit point fixe.

□

Algorithme 2 Itération du point fixe (worklist)

$\mathcal{Q} \leftarrow \mathbb{L}$ une queue des étiquettes
for $l \in \mathbb{L}$ **do**
  $live_{in}[l] \leftarrow \emptyset$
  $live_{out}[l] \leftarrow \emptyset$
end for
while $\#\mathcal{Q} > 0$ **do**
  $q \leftarrow \mathcal{Q}$.pop
  $live'_{in}[q] \leftarrow live_{in}[q]$
  $live_{out}[q] \leftarrow \bigcup_{p \in succ(q)} live_{in}[p]$
  $live_{in}[q] \leftarrow gen[q] \cup (live_{out}[q] - kill[q])$
  if $live_{in}[q] \neq live'_{in}[q]$ **then**
    $\mathcal{Q}$.push(pred(q))
  end if
end while

Lemme – Étant donné $s \in Stm$ une déclaration, à partir de $\bot$ l'algorithme par worklist trouve effectivement un point fixe sur s et termine.

Démonstration – On remarque qu'à la modification de l'analyse de vivacité $LIVE_{out}[\cdot]$, les blocs successeurs ne seront nullement impactés. En effet seuls les blocs prédecesseurs seront altérés d'après notre définition ensembliste de la vivacité. L'algorithme est donc une restriction de l'algorithme naïf vu plus haut. Posons $f = live_{in}$, on a deux cas,

1. si pour $l \in \mathbb{L}$ on a trouvé un point fixe, c'est que l'élément $live_{in}[l]$ n'a pas été modifié donc l'algorithme n'ajoute pas ses prédecesseurs à la queue. Ainsi, si on a trouvé une solution globale à la k-ème itération, alors $Card_k(\mathcal{Q}) > Card_{k+1}(\mathcal{Q}) > \cdots > Card_{k+n}(\mathcal{Q}) = 0$. Cela est du au fait que l'algorithme n'ajoute plus rien à la queue mais lui retire un élément à chaque itération. Comme f est monotone, on a bien trouvé le point fixe minimal, par le théorème de Kleene.
2. sinon le treillis a crû et l'algorithme traite, en plus, les prédecesseurs du bloc altéré. Étant donné $\mathbb{V}$ fini, on itère jusqu'à atteindre le premier cas, ou bien le cas où $f^k(\bot) = \top$ pour tout $k \geq N \in \mathbb{N}$.

À partir du moment où l'algorithme trouve un point fixe minimal ou bien qu'il atteint $\top$, la queue $\mathcal{Q}$ ne se remplit plus mais est dépilée à chaque itération. Donc l'algorithme trouve bien le point fixe minimal et termine.

□

Dans cette étude, nous travaillerons avec les deux algorithmes évoqués pour réaliser les différents tests unitaires.

Exemple.
Considérons la déclaration suivante pour illustrer l'analyse de vivacité,

```
a := 0;
```

```
b := a;
while a < 100 do
  if a = 2 then
    c := a
  else
    c := 2 * a;
    d := b
  endif;
  a := c + 1
done;
return c
```

On obtient donc l'analyse de vivacité suivante, en appliquant l'un des deux algorithmes explicités ci-dessus,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	b := a	$\{a\}$	$\{a, b\}$
3	while a $<$ 100 do	$\{a, b\}$	$\{a, b\}$
4	if a = 2 then	$\{a, b\}$	$\{a, b\}$
5	c := a	$\{a, b\}$	$\{b, c\}$
6	c := 2 * a	$\{a, b\}$	$\{b, c\}$
7	d := b	$\{b, c\}$	$\{b, c\}$
8	a := c + 1	$\{b, c\}$	$\{a, b, c\}$
9	return c	$\{c\}$	$\emptyset$

6 Élimination de code mort

Cette analyse permet une première optimisation qu'est l'élimination de code mort. Elle consiste, en pré-compilation, à générer un nouveau code à partir du code initial, dans lequel le code dit mort n'est plus présent.

Définition – Soit $(x := a)^l$ un bloc d'affectation où $x \in \mathbb{V}$ et $l \in \mathbb{L}$. Si $x \notin LIVE_{out}[l]$, on dit que la variable x est morte.

En dehors de l'analyse, notre implémentation éliminera également les blocs de condition ou de boucle, si toutes leurs déclarations sont *skip*.

6.1 Réduction naïve

Cette optimisation peut en premier lieu être abordée de manière naïve. On notera $\mathcal{A}$ l'analyse de flot de donnée. Considérons Δ la fonction d'élimination du code mort définie par,

$$\Delta : \mathcal{A} \times Stm \longrightarrow Stm$$

et qui, à partir d'une analyse de flot de donnée et d'une déclaration, produit une nouvelle déclaration. Soit $s \in Stm$, alors $s' = \Delta(\bot, s)$ est itérée jusqu'à atteindre $s' = s$. À chaque itération, l'analyse de vivacité de la nouvelle déclaration est à nouveau calculée à partir de $\bot$.

Exemple.
Reprenons la précédente déclaration. Dans son analyse de vivacité, on remarque que $\{d\} \notin LIVE_{out}[7]$, donc d est morte après avoir été affectée. L'algorithme applique alors sur ce bloc une transformation vers une instruction *skip*, puis calcule à nouveau l'analyse, à partir de $\bot$. On obtient alors,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	b := a	$\{a\}$	$\{a\}$
3	while a $<$ 100 do	$\{a\}$	$\{a\}$
4	if a = 2 then	$\{a\}$	$\{a\}$
5	c := a	$\{a\}$	$\{c\}$
6	c := 2 * a	$\{a\}$	$\{c\}$
7	*skip*	$\{c\}$	$\{c\}$
8	a := c + 1	$\{c\}$	$\{a, c\}$
9	return c	$\{c\}$	$\emptyset$

De la même manière, on a que $\{b\} \notin LIVE_{out}[2]$ donc b est morte après affectation. L'algorithme réduit donc le bloc d'étiquette 2 puis recalcule l'analyse. On obtient finalement,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	*skip*	$\{a\}$	$\{a\}$
3	while a $<$ 100 do	$\{a\}$	$\{a\}$
4	if a = 2 then	$\{a\}$	$\{a\}$
5	c := a	$\{a\}$	$\{c\}$
6	c := 2 * a	$\{a\}$	$\{c\}$
7	*skip*	$\{c\}$	$\{c\}$
8	a := c + 1	$\{c\}$	$\{a, c\}$
9	return c	$\{c\}$	$\emptyset$

Maintenant, aucune variable n'est morte après affectation, donc on a atteint un point fixe sur la réduction, c'est-à-dire que l'algorithme ne peut plus réduire quoi que ce soit et renvoie toujours la même déclaration. Cela termine avec la déclaration réduite,

```
a := 0;
skip;
while a < 100 do
  if a = 2 then
    c := a
  else
    c := 2;
    skip
  endif
  a := c + 1
done;
return c
```

6.2 Incrémentalisation de la réduction

Cette réduction naïve est cependant particulièrement inefficace. En effet, elle doit à chaque itération calculer l'analyse à partir de $\bot$ et cela peut s'avérer lourd lorsque nos programmes se composent de milliers de blocs. On introduit donc la réduction de code mort par incrémentalisation, dans laquelle on essaye plutôt de calculer la nouvelle analyse de vivacité à partir de la précédente, ce qui réduit considérablement le nombre d'opérations. Pour résumer l'idée, voilà comment nous pourrions définir cette fois notre application de réduction,

$$\Delta : \mathcal{A} \times Stm \longrightarrow \mathcal{A} \times Stm$$

qui se rappelle récursivement, en utilisant la précédente analyse pour réduire la nouvelle déclaration.

Notation – Par la suite, on notera,

$$\mathcal{L} = \{l \in \mathbb{L} \mid (x := a)^l \text{ et } \{x\} \notin LIVE_{out}[l]\}$$

l'ensemble des étiquettes dont le bloc d'affectation agit sur une variable morte. De plus, si $s \in Stm$ une déclaration, on notera $s[\mathcal{L} \rightarrow \mathit{skip}]$ cette même déclaration, réduite aux blocs d'étiquette dans $\mathcal{L}$.

Cependant, si on considère $s \in Stm$, μ_s son point fixe minimal et $s' = s[\mathcal{L} \rightarrow \mathit{skip}]$ cette même déclaration réduite, alors il n'est pas possible d'utiliser l'analyse de vivacité de s pour poursuivre la réduction, étant donné qu'on ne peut pas garantir que $\mu_s \subseteq \mu_{s'}$ et que, comme démontré plus haut, la recherche de point fixe est croissante monotone uniquement. Il s'agit donc de trouver un moyen de suffisamment décroître le précédent treillis pour obtenir un pré-point fixe, qu'on pourra alors itérer et obtenir le point fixe minimal. Pour ce faire, on peut introduire un ensemble $I_{\mathcal{L}}$ qui agit comme un filtre de manière à supprimer l'information en trop. Pour convenablement filtrer le treillis de s, il faut ajouter de l'information à cette dernière. En effet, il est nécessaire qu'à chaque bloc b, il soit possible de connaître quel bloc sous-jacent a besoin des variables vivantes en b. Le treillis utilisé manque alors d'information et n'est plus adapté, étant donné qu'il n'est pas en mesure d'indiquer d'où vient la propagation d'une variable vivante. Donc le filtrer ne nous avancerait pas, étant donné qu'on n'obtiendra pas de pré-point fixe, mais quelque chose de beaucoup plus petit. On essayera donc plutôt d'utiliser le treillis $(\mathcal{P}(\mathbb{L} \times \mathbb{V}), \sqsubseteq)$.

En ce qui concerne la comparaison de ces paires, elle dépend de la manière avec laquelle l'algorithme de réduction parcourira la déclaration donnée. En effet, si celui-ci se fait strictement de bas en haut, alors la comparaison peut uniquement se faire sur la variable. Nous préférerons néanmoins que ce treillis serve quelque soit l'ordre dans lequel l'algorithme effectue la réduction, la comparaison se fera donc sur l'étiquette et sur la variable. Une fois cela acquis, il est possible d'énoncer le théorème qui suit.

Exemple.
On illustre dans cet exemple, que la réutilisation en l'état de la précédente

analyse de vivacité ne permet pas correctement d'éliminer le code mort. Considérons la déclaration s suivante.

```
a := 0;
b := a + 1;
c := 2 * b;
return a
```

En appliquant l'un des deux algorithmes de point fixe, on obtient l'analyse de vivacité μ_s suivante,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	b := a + 1	$\{a\}$	$\{a, b\}$
3	c := 2 * b	$\{a, b\}$	$\{a\}$
4	return a	$\{a\}$	$\emptyset$

On a $\{c\} \notin LIVE_{out}[3]$ donc on obtient $s' = s[\{3\} \to \textit{skip}]$. Soit $\mu_{s'}$ l'analyse de vivacité que nous devrions avoir pour poursuivre la réduction. Elle est telle que,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	b := a + 1	$\{a\}$	$\{a\}$
3	*skip*	$\{a\}$	$\{a\}$
4	return a	$\{a\}$	$\emptyset$

On a donc que $\mu_{s'} \subseteq \mu_s$, comme $\mu_{s'}[3] \subseteq \mu_s[3]$. Comme nos algorithmes de point fixe reposent sur la monotonie, croissante dans le cas de l'anayse de vivacité, alors on ne pourra jamais atteindre $\mu_{s'}$ à partir de μ_s. Plus spécifiquement, dans s', la variable au bloc d'affectation d'étiquette 2 est morte, or dans μ_s on a $\{b\} \in LIVE_{out}[2]$. La réduction de s' ne peut donc pas se faire à partir de μ_s.

Théorème – Soient $s \in \mathit{Stm}$, et $s' = s[\mathcal{L} \to \textit{skip}]$ une réduction sur l'ensemble d'étiquettes $\mathcal{L}$. Soient μ_s et $\mu_{s'}$ les point fixes minimaux respectifs des deux déclarations. Alors $\exists I_{\mathcal{L}}$ tel que $\forall l \in \mathbb{L}$,

$$\mu_s[l] - I_{\mathcal{L}} \subseteq \mu_{s'}[l]$$

est un pré-point fixe de $\mu_{s'}[l]$.

Démonstration – Soient $s \in \mathit{Stm}$ et $s' := s[\mathcal{L} \to \textit{skip}]$ la déclaration à partir de s dont les blocs d'étiquette dans $\mathcal{L}$ sont réduits. On se place en outre sur le treillis $(\mathcal{P}(\mathbb{L} \times \mathbb{V}), \subseteq)$. Une fois cela donné, on redéfinit ce que sont $vars$, gen et $kill$ de la manière qui suit,

$$vars_a : \mathbb{L} \times a \longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V})$$

retourne désormais l'ensemble des variables d'une expression arithmétique, liées chacunes à l'étiquette du bloc qui les génère et,

$$vars_b : \mathbb{L} \times b \longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V})$$

retourne désormais l'ensemble des variables d'une expression booléennes, liées chacunes à l'étiquette du bloc qui les génère.

$$\begin{aligned} gen : \mathbb{L} \times Block &\longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V}) \\ (l, x := a) &\longmapsto vars_a(l, a) \\ (l, skip) &\longmapsto \emptyset \\ (l, b) &\longmapsto vars_b(l, b) \\ (l, \text{return } a) &\longmapsto vars_a(l, a) \end{aligned}$$

génère maintenant l'ensemble $gen[l]$ et,

$$\begin{aligned} kill : Block &\longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V}) \\ x := a &\longmapsto \mathbb{L} \times \{x\} \\ skip &\longmapsto \emptyset \\ b &\longmapsto \emptyset \\ \text{return } a &\longmapsto \emptyset \end{aligned}$$

génère maintenant l'ensemble $kill[l]$.
Par commodité pour la suite, on se donnera les applications,

$$\begin{aligned} g : \mathcal{P}(\mathbb{L} \times \mathbb{V}) &\longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V}) \\ X_l &\longmapsto \bigcup_{p \in succ(l)} f(X_p) \end{aligned}$$

monotone croissante, d'après la démonstration sur la monotonie de $LIVE_{out}[\cdot]$ ci-dessus où,

$$\begin{aligned} f : \mathcal{P}(\mathbb{L} \times \mathbb{V}) &\longrightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V}) \\ X_l &\longmapsto gen[l] \cup (g(X_l) - kill[l]) \end{aligned}$$

également monotone croissante par la démonstration sur la monotonie de $LIVE_{in}[\cdot]$ ci-dessus. On pose E_s et $E_{s'}$ les deux systèmes d'équations respectivement de s et de s',

$$E_s : \begin{cases} LIVE_{in}[l] = f(LIVE_{in}[l]) & \forall l \in \mathbb{L} - \mathcal{L}\,, \\ LIVE_{in}[k] = f(LIVE_{in}[k]) & \forall k \in \mathcal{L}\,, \end{cases}$$

et

$$E_{s'} : \begin{cases} LIVE_{in}[l] = f(LIVE_{in}[l]) & \forall l \in \mathbb{L} - \mathcal{L}\,, \\ LIVE_{in}[k] = g(LIVE_{in}[k]) & \forall k \in \mathcal{L}\,. \end{cases}$$

Posons enfin, l'ensemble filtre tel que,

$$I_{\mathcal{L}} = \mathcal{L} \times \mathbb{V}$$

On considère $Z = (Z_l)_{l \in \mathbb{L}}$ le plus petit point fixe de la déclaration s *i.e.* la plus petite solution du système E_s. Montrons d'abord que $\forall X = (X_l)_{l \in \mathbb{L}}$ solution de $E_{s'}$ alors,

$$Z_k - I_{\mathcal{L}} \subseteq X_k, \forall k \in \mathcal{L}$$

D'après la définition de gen sur ce treillis il vient que $gen[k] \subseteq I_{\mathcal{L}}$ donc on a

$$Z_k - I_{\mathcal{L}} = g(Z_k) - (kill[k] - \{(k, v) \mid v \in \mathbb{V}\}) \subseteq g(Z_k) \subseteq g(X_k) = X_k$$

comme on suppose Z la plus petite solution du système E_s. Montrons ensuite que $\forall X = (X_l)_{l \in \mathbb{L}}$ solution de $E_{s'}$ on a,

$$Z_l - I_{\mathcal{L}} \subseteq X_l, \forall l \in \mathbb{L} - \mathcal{L}$$

Dans ce cas, on remarque que

$$Z_l - X_l \subseteq Z_l \cap I_{\mathcal{L}}$$

mais aussi que,

$$X_l - Z_l \subseteq \mathcal{U} = \bigcup_{k \in \mathcal{L}} g(X_k) \cap kill[k]$$

et ainsi que,

$$g(Z_l) - I_{\mathcal{L}} \subseteq g(X_l)$$

Donc on obtient que,

$$Z_l - I_{\mathcal{L}} \subseteq gen[l] \cup (g(X_l) - \mathcal{U} - kill[l]) \subseteq f(X_l) = X_l$$

Étant donné que $[g(X_l) - \mathcal{U}] \cap I_{\mathcal{L}} = \emptyset$ il n'est pas nécessaire de retirer à $kill[l]$ les possibles éléments dans $\mathcal{L}$. Donc pour tout X solution du système $E_{s'}$ et pour tout $l \in \mathbb{L}$, on a montré que $Z_l - I_{\mathcal{L}} \subseteq X_l$. La différence ensembliste étant monotone décroissante, on a bien que la différence de $I_{\mathcal{L}}$ à Z est un pré-point fixe de $\mu_{s'}$, la plus petite solution de $E_{s'}$.

□

Exemple.
Illustrons tout ce qui vient d'être dit, avec une déclaration sur laquelle on applique une incrémentalisation, d'abord à l'aide du premier treillis, puis à l'aide du second. Reprenons la précédente déclaration en ajoutant quelques lignes,

```
a := 0;
b := a;
b := b + 3;
while a < 100 do
  if a = 2 then
    c := a;
    d := b
  else
    c := 2 * a;
    e := b
  endif;
  a := c + 1
done;
return c
```

On trouve la première analyse de vivacité, à partir de $\perp$ sur le treillis $(\mathcal{P}(\mathbb{V}), \subseteq)$

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a\}$
2	b := a	$\{a\}$	$\{a, b\}$
3	b := b + 3	$\{a, b\}$	$\{a, b\}$
4	while a < 100 do	$\{a, b\}$	$\{a, b\}$
5	if a = 2 then	$\{a, b\}$	$\{a, b\}$
6	c := a	$\{a, b\}$	$\{b, c\}$
7	d := b	$\{b, c\}$	$\{b, c\}$
8	c := 2 * a	$\{a, b\}$	$\{b, c\}$
9	e := b	$\{b, c\}$	$\{b, c\}$
10	a := c + 1	$\{b, c\}$	$\{a, b, c\}$
11	return c	$\{c\}$	$\emptyset$

Supposons désormais, qu'on veuille réduire le bloc d'étiquette 9, et produire la nouvelle analyse de vivacité à partir de celle décrite ci-dessus. En ce bloc, on a $gen[9] = \{b\}$ donc il faut propager la perte de vivacité de ce bloc, vers ses blocs prédecesseurs. Vient le problème de l'origine de la vivacité sur une variable. En effet, en l'état de ce treillis, il est impossible de savoir, ni qui utilise ces variables dans les blocs successeurs ni qui maintient la vivacité, pour cause du bloc 9, dans les blocs prédecesseurs. On ne peut pas simplement enlever tous les b de l'analyse étant donné que b est aussi utilisée par 3 et 7. Donc on ne peut pas garantir l'obtention d'un pré-point fixe sur le code réduit. Essayons désormais le second treillis, $(\mathcal{P}(\mathbb{L} \times \mathbb{V}), \subseteq)$. Cette fois-ci, on trouve la première analyse de vivacité, à partir de $\bot$,

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a_4, a_5, a_6, a_8\}$
2	b := a	$\{a_4, a_5, a_6, a_8\}$	$\{a_4, a_5, a_6, a_8, b_3\}$
3	b := b + 3	$\{a_6, a_5, a_6, a_8, b_3\}$	$\{a_4, a_5, a_6, a_8, b_7, b_9\}$
4	while a < 100 do	$\{a_4, a_5, a_6, a_8, b_7, b_9\}$	$\{a_5, a_6, a_8, b_7, b_9\}$
5	if a = 2 then	$\{a_5, a_6, a_8, b_7, b_9\}$	$\{a_6, a_8, b_7, b_9\}$
6	c := a	$\{a_6, b_7, b_9\}$	$\{b_7, b_9, c_{10}, c_{11}\}$
7	d := b	$\{b_7, b_9, c_{10}, c_{11}\}$	$\{b_7, b_9, c_{10}, c_{11}\}$
8	c := 2 * a	$\{a_8, b_7, b_9\}$	$\{b_7, b_9, c_{10}, c_{11}\}$
9	e := b	$\{b_7, b_9, c_{10}, c_{11}\}$	$\{b_7, b_9, c_{10}, c_{11}\}$
10	a := c + 1	$\{b_7, b_9, c_{10}, c_{11}\}$	$\{a_4, a_5, a_6, a_8, b_7, b_9, c_{11}\}$
11	return c	$\{c_{11}\}$	$\emptyset$

Réduisons désormais le bloc d'étiquette 9 et appliquons le filtre $I_{\{9\}}$ sur l'analyse.

Étiq.	Bloc	$LIVE_{in}[\cdot]$	$LIVE_{out}[\cdot]$
1	a := 0	$\emptyset$	$\{a_4, a_5, a_6, a_8\}$
2	b := a	$\{a_4, a_5, a_6, a_8\}$	$\{a_4, a_5, a_6, a_8, b_3\}$
3	b := b + 3	$\{a_6, a_5, a_6, a_8, b_3\}$	$\{a_4, a_5, a_6, a_8, b_7\}$
4	while a < 100 do	$\{a_4, a_5, a_6, a_8, b_7\}$	$\{a_5, a_6, a_8, b_7\}$
5	if a = 2 then	$\{a_5, a_6, a_8, b_7\}$	$\{a_6, a_8, b_7\}$
6	c := a	$\{a_6, b_7\}$	$\{b_7, c_{10}, c_{11}\}$
7	d := b	$\{b_7, c_{10}, c_{11}\}$	$\{b_7, c_{10}, c_{11}\}$
8	c := 2 * a	$\{a_8, b_7\}$	$\{b_7, c_{10}, c_{11}\}$
9	*skip*	$\{b_7, c_{10}, c_{11}\}$	$\{b_7, c_{10}, c_{11}\}$
10	a := c + 1	$\{b_7, c_{10}, c_{11}\}$	$\{a_4, a_5, a_6, a_8, b_7, c_{11}\}$
11	return c	$\{c_{11}\}$	$\emptyset$

On a ainsi réussi à conserver l'information de la vivacité de b, propagée à partir des blocs 3 et 7 tout en précisant la perte de vivacité de b à partir du bloc 9. On a bien obtenu un pré-point fixe de cette déclaration réduite à partir de l'analyse de vivacité précédente. Par la monotonie de l'algorithme d'itération, on peut donc faire croître ce treillis et trouver le point fixe minimal.

7 Généralisation

Ceci achève une première analyse statique qu'est l'analyse de vivacité. On s'essaye maintenant à déterminer un lien plus général entre le prédicat, le treillis, et la transformation de code.

7.1 Motivations

On cherche à implémenter un meta-langage, qui servira entre notre langage et son compilateur, à orienter ce dernier sur les optimisations à fournir. Pour ce faire, on se sert de prédicats qui valideront ou non la transformation du code et cette dernière sera plus ou moins efficace selon le treillis qu'on décidera d'utiliser. Ces prédicats sont des applications prenant des éléments évaluables à la compilation, dans notre cas, des variables et plus tard, des routines. Soit $P : \mathbb{V} \longrightarrow \mathbb{B}$ un prédicat quelconque, voici un exemple d'un tel programme,

```
...
x := a;
...
y := 0;
#if P(x)
  while(y < x) do
    y := y + 1
  done;
#else
  y := 1;
#endif
...
```

La bonne utilisation de ces prédicats doit permettre la production d'un code assembleur minimal et le plus optimal possible, ce qui s'avère essentiel lorsqu'on recherche de hautes performances d'exécution. Revenons-en à un langage impératif tel que C pour illustrer cela. Supposons qu'on ait une simple fonction de division par 32 d'un entier sur 32 bits.

```
int div(int x)
{
  return x/32;
}
```

Le compilateur GCC produit alors le code assembleur suivant,

```
div:
  mov     eax, DWORD PTR [rbp-4]
  lea     edx, [rax+31]
  test    eax, eax
  cmovs   eax, edx
```

```
    sar     eax, 5
    ret
```

On suppose désormais que cette fonction sera appelée dans une boucle, à ne traiter que des entiers signés sur 32 bits, positifs. Alors ce code assembleur devient particulièrement inefficace étant donné qu'il considère le cas d'entiers négatifs. En effet, on aimerait plutôt avoir quelque chose comme

```
div:
  mov     eax, DWORD PTR [rbp-4]
  shr     eax, 5
  ret
```

ce qui est déjà bien mieux ! On pourrait donc dans ce cas poser notre metalangage sur l'appel de *div*, avec un prédicat défini par,

$$P : v \longmapsto [\![v > 0 \vee v = 0]\!]^B(\sigma)$$

où σ est l'état de nos variables. Cela donnerait l'hypothèse d'un entier positif à la pré-compilation et permettrait d'obtenir ce code assembleur spécifiquement.

7.2 Première approche

Entamons cette généralisation en partant d'équations similaires à celles de l'analyse de vivacité. On remarque en effet que pour toute sorte d'analyses statiques, on retrouvera toujours l'opérateur $\sqcup$ sur les blocs successeurs ou prédecesseurs. Définissons donc une fonction de transfert t_l au bloc d'étiquette l telle que,

$$t_l : L \longrightarrow L$$

où $(L, \sqsubseteq)$ est un treillis quelconque. Par exemple, la fonction de transfert pour l'analyse de vivacité est définie par,

$$t_l(s) = gen[l] \cup (s - kill[l])$$

On se donne alors pour commencer, les équations,

$$\begin{aligned} A_{in}[l] &= t_l(A_{out}[l]) \\ A_{out}[l] &= \bigsqcup_{p \in \mathcal{K}_l} A_{in}[p] \end{aligned}$$

L'analyse de vivacité développée plus haut se faisait de bas en haut dans le parcours du CFG, on avait alors

$$\mathcal{K}_l = succ(l)$$

Cependant, si l'analyse se fait de haut en bas, on aurait alors plutôt

$$\mathcal{K}_l = pred(l)$$

Jusqu'à maintenant, nous avions utilisé l'union des blocs dans le cadre de l'analyse de vivacité. Cela nous fournissait une analyse dans laquelle les informations requises étaient possiblement vraies. On peut néanmoins restreindre encore plus nos analyses en prenant $\sqcup = \cap$. Dans ce cas, l'information requise doit tout le

temps être vraie. On notera que cette restriction supplémentaire change l'ordre dans notre treillis, avec $\sqsubseteq = \supseteq$.
Ainsi on peut chercher des contraintes à poser sur les prédicats, en fonction de ces deux dimensions que sont l'analyse par l'avant, par l'arrière, intersectée ou à l'union.

8 Annexe

8.1 Validation

La validation a pu se faire en partie grâce aux différentes preuves vues tout le long de cette étude. La réduction de code mort par incrémentalisaiton fut la partie la plus effective. En effet, elle aura demandée plusieurs essais sur la formalisation ainsi que sur l'implémentation, celle-ci aura en partie été guidée par nos différents tests unitaires. Les résultats obtenus sur la monotonie, mais aussi la correction de nos algorithmes ont fourni plusieurs contraintes pour énoncer notre théorème d'incrémentalisation. Au-delà de cet aspect pratique, il n'est évidemment pas concevable de tester tous les cas de figure possibles. C'est là où se limitent nos tests unitaires et où nos preuves prennent le relais et nous donne une garantie lors de la construction de résultats en amont.

8.2 Tests unitaires

Les tests unitaires sont organisés en deux parties distinctes. L'une se charge de tester l'interpréteur ainsi que la représentation statique de notre langage *i.e.* la bonne construction des ASTs, la bonne construction du graphe de flot de contrôle. L'autre se charge de tester l'analyse de vivacité ainsi que la réduction de code mort qui en découle. De manière à tester un large évantail de cas, on générera aléatoirement des milliers de déclarations sur notre langage. Celles-ci sont traitées par l'algorithme naïf de point fixe et par l'algoritme worklist ce qui permet de valider ou non le test. De même pour la réduction de code mort, les déclarations sont traitées à la fois par la réduction naïve et par la réduction incrémentale. En outre, on peut comparer le résultat que fournit l'interpréteur sur une déclaration et sa déclaration réduite pour valider ou non le test.

8.3 Générateur

Dans l'optique de vérification de la robustesse de nos différentes analyses, il peut être bon de réaliser des tests unitaires sur chacune d'elles. Pour cela on ne se restreindra pas aux tests conçus pas nous-même mais on essayera aussi de trouver tous les cas de figure par la manière forte. Un générateur de code complètement arbitraire est alors une bonne solution pour réaliser de tels tests. Ce générateur de code est ajusté par un paramètre qu'est la quantité de variables affectées dans la déclaration. On cherche également à ce que le code généré soit le plus proche possible de celui qu'aurait pu fournir un humain. Dans notre syntaxe, il suffit pour le moment de donner une certaine répartition lors de la création des déclarations, en permettant plus d'affectations, que de blocs Ifte ou Whiledo par exemple.

8.4 Bugs

Les comportements anormaux ont pu se manifester lors de la formalisation, et son implémentation, de la réduction de code mort par incrémentalisation. Comme déjà expliqué, les différentes preuves de cette étude ont été capitales quant à la validation de notre implémentation. Avant la formalisation du théorème d'incrémentalisation, et la rédaction de sa preuve, le générateur a pu nous fournir plusieurs déclarations qui ne validaient pas les tests et qui ont permis d'aiguiller la manière dont les choses devaient se faire.

9 Conclusion

Cela conclut cette branche du projet Gloca. Dans cette étude, nous avons formalisé un simple langage impératif avec lequel l'analyse de vivacité a pu être décrite. Le reste de l'étude s'est focalisé sur l'utilisation de cette analyse statique comme un premier exemple de prédicat au sein du méta-langage, attestant de la vivacité ou non d'une variable. Ainsi en sommes-nous venu à la réduction de code mort. Dans notre cas, nous nous serons questionnés sur la possibilité de réutiliser la précédente analyse de vivacité pour calculer la suivante. Cela a permis une optimisation considérable sur l'analyse, et sur la réduction de manière plus générale. À travers le théorème d'incrémentalisation et de sa preuve, on a montré qu'une telle optimisation est possible et de ce fait, on aimerait pouvoir généraliser ce théorème de sorte à ce qu'il s'applique à d'autres analyses statiques. Comme éludé dans la section 7, dédiée à la généralisation, on peut observer les comportements selon que l'analyse se fait de haut en bas ou de bas en haut, mais aussi selon la qualité de l'information qu'elle entretient. Parallèlement à ces travaux, une autre analyse statique a été formalisée pour Gloca, elle consiste en la réaffectation des variables plusieurs fois affectées au sein de la déclaration. Il s'agit donc désormais de trouver une abstraction suffisante du théorème d'incrémentalisation pour pouvoir obtenir quelque chose de fonctionnel sur cette seconde analyse.