Documentation sur les extensions

Antoine Turkie, Augustin Chenevois, Aurélien Delory, Mihaja Razafimahefa, Sarah Ramaharobandro

Grenoble INP - Ensimag · Projet GL 2023-2024

Table des matières

Introduction

Motivé par l'algorithmique et la conception de programme rapide mais correct, notre équipe a naturellement choisi l'extension OPTIM qui répond à ce besoin utile à la vie de tout développeur.

Spécification de l'extension

L'extension OPTIM génère un code machine *optimisé* pour IMA et différent de la spécification du projet. Elle est activée au moyen de l'option `-optim` lors de l'appel de `decac`. Les optimisations effectuées sont détaillées dans la partie Choix de conception, d'architecture, et d'algorithmes. L'extension n'engendre aucune limitation sur les fonctionnalités de deca que nous avons développées et agit que ce soit sur la spécification sans objet (dans le programme principal) ou sur la spécification objet (dans les corps des méthodes). Notamment, la compilation avec l'option -optim s'achève avec succès pour tous les tests de non-régression.

Analyse bibliographique

Conseils préliminaires

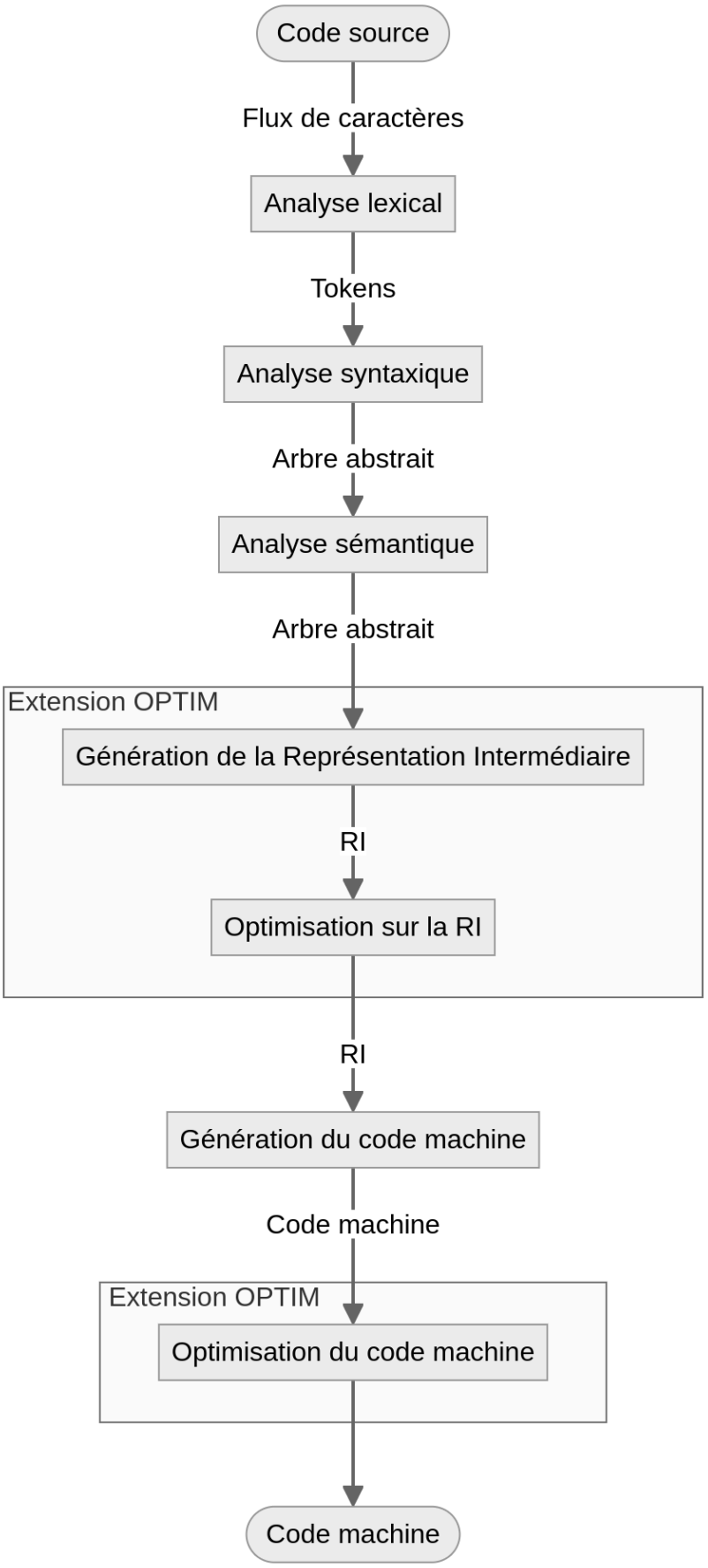
Lors du suivi 1, notre professeur encadrant nous a proposé d'orienter nos recherches sur deux mots clés : SSA et CFG. Leur intérêt semble crucial pour la bonne réussite de l'extension. Nos efforts de recherche bibliographiques se sont alors concentrés sur ces deux termes.

Théorie et définitions

Structure d'un vrai compilateur

Une différence notable existe entre la spécification du compilateur à développer par défaut ainsi qu'un vrai compilateur comme LLVM ou HotSpot JVM est la génération d'une représentation à l'interface entre celle dépendante du langage source (notre arbre abstrait) et du code machine (instructions assembleur pour IMA). Nous appelons cette représentation la "**Représentation Intermédiaire**" ou **RI**.[1] Les étapes A, B et C du projet ne couvrent pas cette représentation comme elle sert avant tout à effectuer divers optimisations par le compilateur, indépendamment du code généré

pour la machine. Notre extension se doit alors de rajouter de nouvelles étapes au compilateur afin d'exécuter ces mêmes algorithmes d'optimisation.
Une autre étape également non couverte par la spécification est l'optimisation du code cible généré pour la machine.



Structure d'un vrai compilateur mettant en évidence les étapes de l'extension OPTIM[2]

Formalisation d'une Représentation Intermédiaire

Une Représentation Intermédiaire est une représentation abstraite du programme source manipulée par le compilateur pour faciliter différentes analyses et optimisations.[3] Elle peut avoir différentes formes : graphe, linéaire, Three Address Code, SSA Form, CPS.[2] Et un compilateur peut également générer et employer plusieurs formes de RI durant son exécution. Par exemple, l'arbre abstrait généré par le projet est une Représentation Intermédiaire.[4] page 9

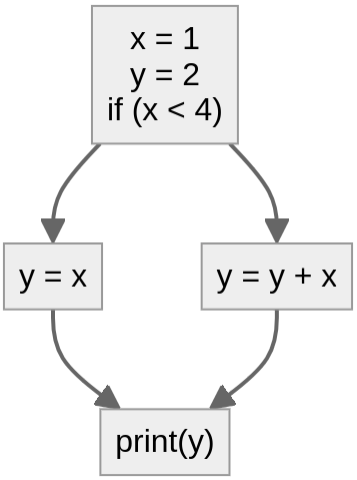
Control Flow Graph

Un Control Flow Graph ou CFG est une Représentation Intermédiaire sous forme d'un graphe orienté dont les sommets sont des **blocs de base**, qui sont une suite d'instructions où un flot de contrôle ne peut entrer que par le début de cette suite et ne peut sortir que par la dernière.[3] Une propriété remarquable de cette définition est la garantie que si on entre dans un bloc de base, toutes les instructions le composant seront exécutées. Les arcs du CFG sont un flot possible du programme.

Par exemple, considérons ce programme que l'on nommera **Programme 1** par la suite :

```
x = 1
y = 2
if (x < 4) {
     y = x
}
else {
     y = y + x
}
print(y)
```

Dont voici son CFG :



Nous pouvons bien distinguer les blocs de base et leurs propriétés ainsi que le flot du programme. Nous remarquons également que nous nous ne soucions plus du résultat de la condition des structures de contrôle (ici `if`), seulement des possibles branchements des blocs de base.
Nous appelons le bloc d'entrée le bloc de base contenant la première instruction du programme source.

Static Single Assignment Form

Un Static Single Assignment Form ou forme SSA est une forme restreinte de Représentation Intermédiaire respectant 2 conditions[2] :

- chaque définition d'une variable a un nom **unique**
- chaque utilisation d'une variable renvoie à une seule définition

Plutôt qu'une forme particulière de Représentation Intermédiaire, la littérature présente SSA comme une discipline, une façon de penser différente de nos programmes et de son analyse.
L'intérêt principal de transformer notre Représentation Intermédiaire sous forme SSA est la simplification des optimisations faites par le compilateur.

En effet, considérant cette exemple :

```
y = 1
y = 2
x = y
```

Ce programme ne respecte pas les conditions d'une forme SSA comme y renvoie deux définitions : 1 et 2.

Une forme SSA de ce programme est :

```
y1 = 1
y2 = 2
x1 = y2
```

Ici, nous remarquons l'ajout d'étiquettes ou de *versions* pour distinguer les deux définitions de y. Ainsi, ce programme respecte les conditions d'assignations **statiques** requises par la SSA. Nous remarquons également grâce à cette représentation que nous n'avons plus d'ambiguïté par rapport à l'**utilisation** de y à la ligne 3. La définition **unique** de y2 nous garantie que x1 renverra toujours 2 comme sa propre définition, ce qui nous permet déjà de faire une optimisation triviale :

```
y1 = 1
y2 = 2
x1 = 2
```

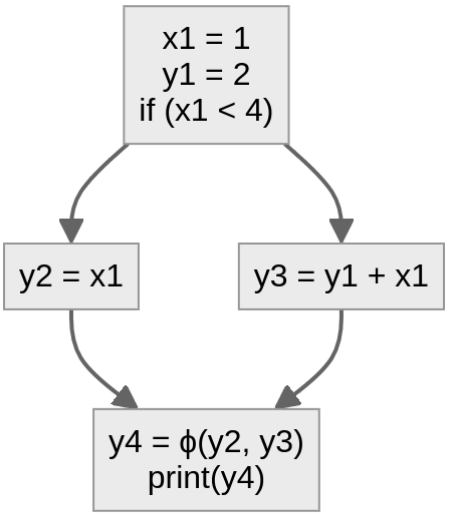
Nous irons un peu plus en détail sur les types d'optimisations possibles d'un compilateur. Ce que nous pouvons conclure ici est que SSA est un outil puissant pour détecter et appliquer des optimisations difficilement remarquables autrement.

Cependant, essayons de convertir **Programme 1** sous forme SSA :

```
x1 = 1
y1 = 2
if (x1 < 4) {
     y2 = x1
}
else {
     y3 = y1 + x1
}
print(y?)
```

Nous rencontrons un problème : comment choisir à l'avance quelle variable parmi `y2` et `y3` utiliser dans notre instruction `print` ? Heureusement, ce dilemme a déjà été résolu par les inventeurs de la SSA Form.[5] Pour cela, nous devons introduire une fonction phi ou ϕ.

Une fonction ϕ est un concept important de la SSA Form. C'est une fonction fictive prenant en paramètre les différentes versions d'une variable venant de plusieurs branchements, convergeant sur un appel de cette variable. Comme nous évoquons de nouveaux des branchements, il est naturel d'utiliser le CFG de **Programme 1** pour illustrer l'implémentation de notre fonction ϕ :



Remarquons l'introduction d'une quatrième variable `y4` qui est notre fonction ϕ et permet de réconcilier les différentes versions de `y` arrivant dans notre bloc de base, venant des différents branchements de l'instruction `if`. Non seulement, `y4` résout notre problème mais elle préserve également la définition d'une forme SSA pour notre Représentation Intermédiaire.

Il est important de rappeler que cette fonction ϕ n'est qu'une notation pour les besoins de la SSA Form, et non une instruction pour le code cible. Plus tard, elle sera effacée en réintroduisant la variable dans les blocs antérieurs.

Construction de la forme SSA

Après avoir compris le potentiel de la forme SSA, nous nous sommes penchés sur son implémentation dans un compilateur classique afin de déterminer nos choix de conception et d'intégration de cette Représentation Intermédiaire au sein de notre propre compilateur.

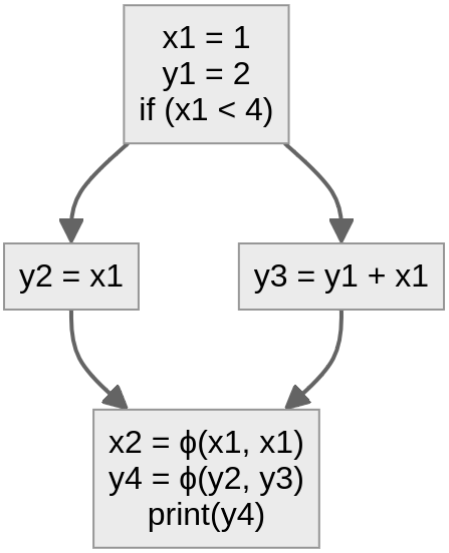
Le défi majeur pour notre compilateur est l'introduction des fonctions ϕ : comment les placer dans la Représentation Intermédiaire ? Peut-on optimiser ce placement ? Puis comment enlever ces fonctions lors de la génération du code machine ?

Approche naïve pour le placement des fonctions ϕ

Une approche naïve[6] (mais efficace) pour gérer les fonctions ϕ est :

- Passe 1: pour chaque variable x de la CFG, à chaque début d'un bloc de base avec plusieurs prédécesseurs, placer une fonction ϕ de la forme $x = \phi(x, \ldots, x)$ avec autant de paramètres que de prédécesseurs de ce bloc de base.
- Passe 2: déterminer la définition de chaque variable, avec les renommages (*versionnages*) nécessaires, et renommer toute usage d'une variable selon la définition unique arrivant des prédécesseurs.

Le résultat de cette approche sur le CFG de **Programme 1** nous donne :



Remarquons l'introduction d'une nouvelle fonction ϕ avec x2. Bien que correct, il est évident que cette déclaration est inutile et redondante (les paramètres sont les mêmes). De plus, x2 ne sera même pas utilisé par les prochaines instructions ce qui rajoute du code mort dans notre programme. Dans la littérature, notre algorithme a construit une forme SSA **maximale**. Notre extension cherche à produire un compilateur optimisé, cependant nous ne souhaitons pas sacrifier sa vitesse de compilation en manipulant des fonctions ϕ inutiles.

Après plus de recherche, nous avons découvert que des placements stratégiques (en opposition à l'approche naïve) des fonctions ϕ permet d'obtenir une forme SSA **minimale**. Par exemple, notre première CFG sous forme SSA du **Programme 1** est minimale.
Une construction d'une forme SSA est minimale si l'on a placé le moins de fonction ϕ redondante possible. D'autres algorithmes plus complexes permettent d'obtenir cette minimalité. Cependant, une forme SSA minimale n'est pas encore optimale et peut contenir des fonctions ϕ non utilisé (ou fonction ϕ mort).[6]

Etat de l'art d'un compilateur efficace sous forme SSA : les travaux de Cytron et al.

L'algorithme introduit par Cytron et al.[7] est l'algorithme classique[6] pour convertir une Représentation Intermédiaire sous forme SSA tout en garantissant sa minimalité. Elle se base sur le Control Flow Graph que nous avons défini plutôt. Les étapes de constructions sont :

- A partir du CFG, construire l'Arbre de Domination, représentant la relation de domination entre les sommets (bloc de base) du CFG. On dit qu'un sommet u domine un sommet v si tous les chemins passant du bloc d'entrée du CFG à v passe par u. Nous remarquons une relation réflexive de cette définition avec u dominant u.
  Ainsi, nous définissons u **domine strictement** v si u domine v et $v \neq u$.
- Construire l'ensemble des Frontières de Domination des sommets du CFG. La Frontière de Domination d'un sommet u est l'ensemble des sommets v tel que u domine un prédécesseur de v mais ne domine pas strictement v.
- Enfin, insérer les fonctions ϕ dans chaque sommet de l'ensemble des Frontières de Domination d'une variable.

Malgré l'efficacité et les nombreux usages de cet algorithme dans la littérature, nous sommes submergés par la quantité de constructions intermédiaires requises pour l'exécuter. Nous avons besoin de la CFG puis d'un Arbre de Domination et des Frontières de Domination et nous n'avons même pas encore discuté de la représentation et de la cohésion de ces structures de données avec l'approche imposée par le sujet. Nous étions tout de même prêts à le manipuler dans notre compilateur. Cependant, nous avons continué de faire plus de recherches pour trouver un algorithme demandant moins de ressources et si possible, se rapprochant du projet.

Une approche novatrice : les travaux de Braun et al.

Après avoir appris et compris le fonctionnement de l'algorithme de Cytron et al., sa complexité et notre temps limité nous a poussé à faire plus de recherches. En effet l'algorithme de Cytron et al. demande en entrée un CFG du programme qui est une Représentation Intermédiaire du code source. Cependant, nous en avons déjà construit une RI : l'Arbre Abstrait décoré. Nous étions intéressés par d'autres manières plus simples et plus en lien avec notre projet de construire une forme SSA **minimale**. Nous avons cherché un algorithme pour traduire directement notre Arbre Abstrait en forme SSA sans passer par les dominations ni par un CFG si possible.

Quelques résultats StackOverFlow[8] nous ont proposé des méthodes de parcours de l'Arbre Abstrait pour notre forme SSA. Cependant, nous avons découvert une publication de Braun et al.[9] présentant une méthode simple pour construire une forme SSA sans passer par un CFG non-SSA, comme l'algorithme de Cytron et al. Cette algorithme accepte une traduction **directe** de l'Arbre Abstrait vers une Représentation Intermédiaire sous forme SSA sans passer par d'autres constructions abstraites. De plus, elle assure que durant la construction de cette représentation, elle est sous forme SSA. Cela nous permet alors d'appliquer différents algorithmes d'optimisation basée sur SSA **durant sa construction**. Enfin, la forme SSA produite est **minimale** et **élaguée**[1], tout en ayant une complexité en temps similaire à celle de Cytron et al.

Pour cela, l'algorithme de Braun et al. utilise une approche récursive de parcours l'Arbre Abstrait et emploie deux analyses[2] : Local Value Numbering et Global Value Numbering. L'algorithme associe une définition courante d'une variable en fonction du bloc où il se trouve. Local Value Numbering permet de stocker cette définition lors de la déclaration de la variable dans un bloc de base. Si l'on essaie de le lire dans le même bloc, nous pouvons alors facilement accéder à sa définition courante. Cependant, si la définition de la variable ne se trouve pas dans le bloc, nous faisons un appel récursive des blocs créés antérieurement pour obtenir sa définition, ce qui est une implémentation du Global Value Numbering. L'algorithme créé les fonctions ϕ s'il détecte une lecture récursive d'une variable mais que le bloc courant a plusieurs prédécesseurs. Puis il essaie tout de suite de supprimer les fonctions ϕ devenues triviales après cette création. Il y a 2 distinctions pour supprimer une fonction ϕ triviale :

[1] élimination des fonction ϕ morts

[2] L'algorithme analyse aussi les CFG non réductibles. Mais comme Deca ne prend pas en charge les instructions `goto`, Deca est réductible et nous pouvons ignorer cette partie de l'algorithme de Braun et al.

- supprimer les fonctions ϕ de la forme $v = \phi(v, \ldots, v)$
- remplacer les fonctions ϕ de la forme $v_i = \phi(v_{i_1}, \ldots, v_{i_n})$ avec $i_1, \ldots, i_n \in \{i, j\}$ par v_j

Enfin dès que la définition est trouvée ou que la fonction ϕ est créée, il stocke cette définition courante dans le bloc concerné.

Ouverture

L'article de Braun et al. présente d'autres algorithmes aussi intéressantes pour la construction de la forme SSA à partir de l'Arbre Abstrait, avec leurs propres atouts et inconvénients. Il est destiné à l'attention du lecteur qui souhaite étudier plus en détail les travaux de recherches dans le domaine de la SSA. Ayant fait de même, nous souhaitons faire une mention spéciale aux algorithmes de Click et Paleczny ainsi que Aycock et Horspool qui semblent prometteurs.

Click et Paleczny[10] présentent en plus de l'algorithme une Représentation Intermédiaire sous forme de graphe et respectant SSA que l'on peut directement utiliser. Cependant, elle n'est pas minimale. Comme nous n'avions pas encore discuté du choix de notre Représentation Intermédiaire finale, Arbre Abstrait ou CFG sous forme SSA, cet article de recherche nous a surtout inspiré sur son approche graphique de la SSA et comment nous pouvons adopter une approche similaire pour notre compilateur.

Aycock et Horspool[11] présente un algorithme simple et qui reprend en partie l'approche naïve en rajouter des fonctions ϕ pour chaque variable à chaque début de bloc de base. Cependant, ils essaient ensuite d'éliminer des fonctions ϕ considérées trivials comme avec l'algorithme de Braun et al. Cela permet à l'algorithme d'obtenir une forme SSA minimale[3].

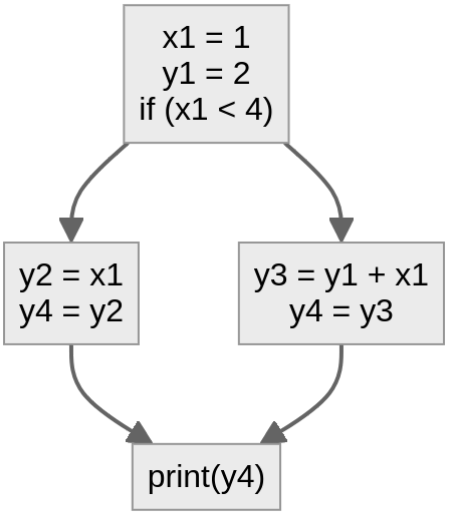
Déconstruction de la forme SSA

La prochaine étape après la construction de la forme SSA et l'exécution des algorithmes d'optimisation sur la Représentation Intermédiaire est bien sûr la suppression des fonctions ϕ spécialement créée mais qu'aucune machine ne supporte. En théorie, la suppression revient à déclarer la variable qui était la fonction

[3] pour les programmes réductibles

ϕ dans les blocs de base prédécesseurs et assigner la définition courante de la variable dans le bloc.[6]

En reprenant notre **Programme 1**, la déconstruction de notre forme SSA minimale donne :



La fonction ϕ a bien été supprimée. Mais la variable y4 est toujours présente et cette fois bien défini. Notre Représentation Intermédiaire est dorénavant compatible et prête pour la génération du code machine.

Choix de conception, d'architecture, et d'algorithmes

Comme précisé dans la partie montrant la structure d'un vrai compilateur avec les étapes concernés par notre extension, nous devons ajouté 3 nouvelles étapes à notre compilateur :

1. Génération de la Représentation Intermédiaire
2. Optimisation sur la Représentation Intermédiaire
3. Optimisation sur le code machine

Si l'option -optim a été spécifiée par l'utilisateur, ces étapes seront exécutées par l'appel de `prog.optimizeProgram()` dans la méthode `doCompile` de `DecacCompiler`.

Génération de la Représentation Intermédiaire

Choix de l'algorithme de construction de la forme SSA

Toujours motivé par la forme SSA, nous avons choisi d'implémenter dans notre compilateur un des algorithmes de construction et de déconstruction que nous avons étudiés précédemment. L'algorithme de Braun et al. très prometteur avec un rapport complexité/bénéfice convaincant nous a plu. Nous avons alors décidé d'implémenter cet algorithme dans la génération de la Représentation Intermédiaire de notre compilateur.
En seconde position, nous avons tout de même envisagé l'algorithme de Aycock et Horspool pour sa très grande simplicité. Cependant, la génération de multiples fonctions ϕ inutiles peut nuire à la vitesse de compilation et nous renvoie au même dilemme que l'approche naïve.

La propriété unique d'optimisation durant la construction de la forme SSA de l'algorithme de Braun et al. nous permet également de regrouper la génération et l'optimisation de la Représentation Intermédiaire en une passe de l'Arbre Abstrait.

Architecture et structures de données

Cependant, après avoir choisi l'algorithme de génération de la forme SSA, il nous fallait encore choisir la forme de notre Représentation Intermédiaire. Comme discuté précédemment, nous souhaitons garder qu'une seule représentation si possible : l'Arbre Abstrait, et l'optimiser directement. Cela est rendu possible par l'algorithme de Braun et al. qui le manipule directement. Néanmoins, nous étions toujours contraint d'avoir un CFG pour garder en mémoire le résultat de l'analyse des blocs de base par le Local Value Numbering et le Global Value Numbering.

Pour rendre la manipulation et les échanges d'informations entre le CFG et l'Arbre Abstrait le plus facile possible, nous nous sommes penchés sur la structure de l'arbre pour voir si nous pouvions réutiliser ou hériter certains de ses composants. En effet, pour construire notre CFG, nous avons besoin de créer deux nouvelles structures de données :

- un bloc de base qui est une liste d'instructions et une liste de blocs de bases de ses prédécesseurs (et de ses successeurs éventuellement), afin de représenter le flot de contrôle.
- la CFG elle-même qui sera une liste de bloc de base.

Nous avons remarqué la classe `ListInst` qui est déjà une liste d'instructions. Ainsi, nous pouvons migrer toute l'analyse contextuelle et syntaxique effectuée sur les instructions de `ListInst` et les lier dans un bloc de base. Tout cela par un héritage de `ListInst` suivi de champs pour stocker la liste des prédécesseurs (et successeurs) du bloc.

De même, si nous pouvons réutiliser les composants de l'Arbre Abstrait, nous remarquons que `ListInst` hérite de `TreeList<AbstractInst>` pour manipuler une liste d'instructions. Nous pouvons faire de même pour notre CFG qui est une liste de bloc de base en l'héritant de `TreeList<>` pour bénéficier des mêmes manipulations de liste que `ListInst`.

Nous avons alors créé un nouveau paquet `fr.ensimag.deca.extension.tree` avec les implémentations concrètes d'un bloc de base par `BasicBlock` et d'un CFG ou liste de blocs de base par `ListBasicBlock`.

La construction du CFG sera évidemment faite pour le programme principal. Par l'intégration de nos structures de données personnalisées dans la structure de l'Arbre Abstrait, cette liste sera stockée dans Main par un champ `blocks`. Le champ `blocks` est également présent dans `MethodBody` qui contient aussi une liste d'instructions. Cette modularité nous permet d'optimiser le corps des méthodes des classes de la même manière que le programme principal. Cela se répercute dans l'appel de `prog.optimizeProgram()` qui commencera par optimiser les méthodes des classes puis le programme principal.

Nous avons procédé de même pour la fonction ϕ avec la classe `Phi`. Pour pouvoir être assignée à une variable, notre expression doit hériter de `AbstractExpr`. Ainsi, `Phi` héritera aussi de `AbstractExpr` pour être compatible avec la structure de l'Arbre Abstrait. De même, les paramètres de la fonction ϕ dénommés opérandes seront aussi des `AbstractExpr` et seront stockés dans une liste.
Une dernière classe, Undef`Phi`, est présente dans notre paquet. C'est une structure spéciale de l'algorithme de Braun et al. permettant de spécifier que la fonction ϕ recherché par un usage d'une variable n'est pas accessible (ou n'est pas dans le bloc d'entrée).

Les travaux de Braun et al. nous proposent des fonctions impératives décrivant la construction de la forme SSA, à adapter pour notre projet. Nous avons alors créé une

classe `SSAFormHelper` dans le paquet `fr.ensimag.deca.extension` contenant ces fonctions ainsi que deux champs :

- `Map<AbstractLValue, Map<BasicBlock, AbstractExpr>> currentDef` : implémentation concrète de `currentDef[variable][block]` de l'algorithme de Braun et al. Dans notre arbre, toutes variables pouvant être assignées une valeur hérite de `AbstractLValue`, et une valeur (concrète comme les littéraux ou à calculer comme les opérations arithmétiques) hérite de `AbstractExpr`. L'association est évidente. L'héritage de `Phi` prend également tout son sens ici.
- `Map<BasicBlock, Map<AbstractLValue, Phi>> incompletePhis` : implémentation concrète de `incompletePhis[block][Variable]` suit le même raisonnement.

Construction du CFG

Pour chaque liste d'instructions (dans le programme principal ou une méthode de classes), nous avons une méthode optimizeX qui construit la liste de bloc (la CFG) ainsi que le premier bloc appelé bloc d'entrée. Ensuite nous ajoutons chaque déclaration de variable de ListDeclVar dans ce bloc d'entrée en veillant à ajouter leur définition dans currentDef. Puis, nous construisons les autres blocs de bases pour les prochaines instructions dans ListInst, qui essaie par défaut d'ajouter l'instruction courante au bloc de base courant, en veillant toujours d'ajouter la définition courante d'une variable dans currentDef si nous avons une instruction Assign.

Nous avons un traitement différent des instructions IfThenElse, While comme ces structures de contrôle crée des branchements. Nous avons choisi de respecter la construction de Braun et al. qui précise la structure de leurs blocs de base.[9] page 7

Construction de la forme SSA selon l'algorithme de Braun et al.

Nous encourageons vivement le lecteur à lire l'article de recherche de Braun et al.[9] pour comprendre les méthodes de la classe `SSAFormHelper`. Elles sont l'implémentation concrète des fonctions de son algorithme et il est inutile de réécrire son très bon article ici.

Malheureusement, à cause du temps qu'il nous manquait, nous n'avons pas pu terminer l'implémentation de l'algorithme de Braun et al. Nous avons une version fonctionnelle du Local Value Numbering, testée et validée. Le Global Value

Numbering bien qu'implémenté dans notre classe, n'a pas été testé. Ainsi, nous avons commenté la ligne

```
return readVariableRecursive(variable, block)
```

du rendu final. Cela présente 2 avantages :

- L'introduction des fonctions ϕ se fait lors du Global Value Numbering. En omettant cette étape, notre Représentation Intermédiaire (l'Arbre Abstrait et le CFG résultante) reste compatible avec la génération du code machine du projet (l'étape C). Nous n'avions pas eu besoin d'implémenter en plus un algorithme de déconstruction de la forme SSA. De plus, nous n'avons pas encore commencé la conception de cette dernière, mettant en doute une réalisation réaliste de cette tâche dans le peu de temps qu'il nous restait.
- Des optimisations peuvent tout de même se faire sans cette étape de l'algorithme de Braun et al. Ces optimisations seront visibles sur le produit final ce qui nous permet d'avoir quelque chose de concret pour le client.

Optimisations sur la Représentation Intermédiaire

Avec le Local Value Numbering permise par l'algorithme de Braun et al., les différents types d'optimisation que nous avons pu faire directement sur l'Arbre Abstrait sont appelés des **optimisations locales**.[12] Ces optimisations s'opèrent sur les expressions héritant de `AbstractExpr`. Bien que restreintes, nous avons tout de même de bonnes performances en utilisant uniquement ces optimisations.

Les résultats des optimisations suivantes sont immédiatement répercutés sur l'Arbre Abstrait, qui sera modifié avec la version optimisée de chaque expression. Pour cela, nous appelons une méthode `evaluate()` sur chaque `AbstractExpr`. Elle renvoie une nouvelle `AbstractExpr` qui sera l'expression optimisée ou l'ancienne si aucune optimisation n'a pu être faite.

Variable propagation

Grâce au champ `currentDef` que nous mettons à jour à la volée, nous pouvons "propager" la définition d'une variable lorsqu'elle est utilisée.

Soit l'exemple deca :

```
int a;
int b = a;
println(a + b);
```

Lors de la déclaration de la variable b, nous mettons sa définition courante à a. Ainsi, lors de son utilisation dans println, nous pouvons remplacer b par a. Ce qui nous donnes comme résultat :

```
int a;
int b = a;
println(a + a);
```

Constant propagation

Pour l'instant, notre optimisation est un peu négligeable, mais c'est parce que nous n'avons pas initialisé notre variable a. Soit l'exemple modifié :

```
int a = 42;
int b = a;
println(a + b);
```

En faisant une variable propagation, nous obtenons le résultat précédent. Mais nous nous sommes arrêtés ainsi seulement parce que a n'était pas défini (et provoque un comportement indéfini du programme). Ici, la définition courante de a est 42, l'algorithme assignera alors la définition courante de b à 42 et fera les remplacements nécessaires à chaque utilisation de ces variables. Ce qui nous donnes comme résultat :

```
int a = 42;
int b = 42;
println(42 + 42);
```

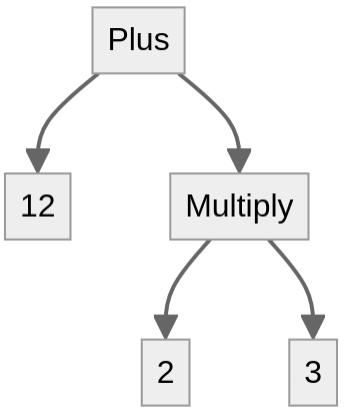
Cette optimisation est très utile comme un LOAD de variable depuis la mémoire est plus coûteuse qu'un LOAD immédiat en Deca.

Constant folding

Sur le résultat précédent, nous pouvons tout de suite faire une autre optimisation intéressante : calculer une opération arithmétique directement dans l'Arbre Abstrait. Ce qui nous donne comme résultat :

```
int a = 42;
int b = 42;
println(84);
```

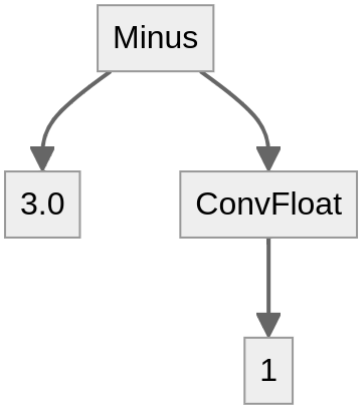
Cette optimisation n'est effectuée que si les deux opérandes d'une opération sont constantes. De plus, elle est **récursive**[4] : si notre arbre est



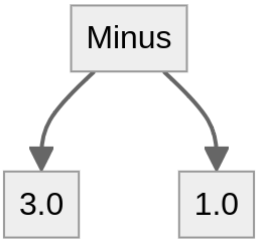
L'algorithme va "plier" l'expression pour ne plus avoir qu'une feuille IntLiteral de valeur 18.

Cette optimisation fonctionne également sur les opérations arithmétiques avec des ConvFloat d'une constante. Soit le programme représenté partiellement par l'Arbre Abstrait suivant :

[4] contrairement à ce qui a été dit en soutenance. L'optimisation sur le code machine des expressions arithmétiques a été dépréciée



L'algorithme va dans un premier temps faire la conversion de la valeur `1` par son équivalent flottant `1.0` puis le stocker dans une nouvelle feuille `FloatLiteral` qui remplacera `ConvFloat` :



La conversion faite, nous pouvons maintenant effectuer l'opération demandée sur les deux opérandes flottantes pour ne plus avoir qu'une feuille `FloatLiteral` de valeur 2.0.

Cette optimisation est essentielle pour notre extension et celle qui contribue le plus à nos bonnes performances. En effet, elle permet de nous abstenir autant que possible d'opérations très coûteuses (la division, modulo ou encore la multiplication).

Optimisations sur le code machine

Division et multiplication d'entiers par une puissance de 2

Les temps d'exécution des instructions de multiplication de de division sont assez importants (20 pour MUL et 40 pour DIV). Étant donné que l'on a à notre disposition les instructions SHL et SHR, il est possible de diminuer le temps nécessaire pour effectuer une opération de multiplication / division par 2. Chaque opération de décalage prend 2 cycles à s'exécuter. Cependant, on ne peut décaler que d'un bit à la fois. On utilise donc cette technique que pour les nombres inférieurs à 2^{10} pour la multiplication et pour les nombres inférieurs à 2^{20} pour la division. Si on se trouve

dans une de ces situations, on remplace une opération de multiplication / division par 2^n par n opérations de décalage vers la gauche / droite.

Opérations modulo 2

L'opération REM prend aussi un temps important de 40 cycles. De plus l'opération modulo avec l'opérande 2 est assez fréquemment utilisée, notamment pour vérifier la parité d'un nombre. Voici comment nous avons optimisé cette opération.

```
n % 2 = n - ((n >> 1) << 1)
```

Cette optimisation nécessite cependant un registre en plus mais elle fait passer le temps d'exécution de l'opération de 40 cycles à 10 cycles.

Stockage des variables dans les registres

Même si le temps pour LOAD / STORE depuis la mémoire et un registre prend le même temps, stocker des variables dans un registre permet d'effectuer certaines opérations en une seule instruction à la place de 3 instructions car les instructions prennent souvent un registre en paramètre. Dans le cas où la variable est en mémoire, on doit alors utiliser LOAD pour transférer la variable dans un registre puis utiliser STORE pour enregistrer le résultat de l'opération dans la mémoire.

Nous avons utilisé cette optimisation pour améliorer les assignations à partir d'une variable ou d'une constante :

```
x = y;
x = 5;
```

Nous avons aussi amélioré les opérations arithmétiques ayant pour un des opérandes la variable elle-même et en deuxième opérande une constante ou une variable :

```
x = x / 2;
x = x + y;
x = x * y;
```

Assignation des variables aux registres

Supposons que l'on ait n registres disponibles avec $4 \leq n \leq 16$. Etant donné que les registres R0 et R1 sont des registres scratch, on a $n - 2$ registres assignables. Cependant, si l'on alloue tous ces registres aux variables, il va falloir constamment utiliser la pile pour effectuer les opérations binaires, ce qui prend 2 instructions de 2 cycles en plus par opération. Nous avons donc décidé d'allouer la moitié de ces n - 2 registres aux variables.

Méthode de validation

Après l'ajout d'une nouvelle optimisation, nous commencions par vérifier que notre compilateur préserve le comportement attendu. Pour cela, on compilait un fichier .deca contenant des instructions que nous avons optimisé puis on observait le .ass créé.

Nous avons rajouté un test manuel `test_optim` pour visualiser l'Arbre Abstrait résultante de nos optimisations sur la Représentation Intermédiaire. Il fonctionne de la même manière que les autres lanceurs par défaut du projet et se trouve dans script/launchers. La classe de test java appelé par le lanceur est `ManuelTestOPTIM.java`.

Pour généraliser les tests de non-régression sur l'extension, nous avons rajouté une autre passe dans notre script `run-gencode.sh` qui vérifie que `decac -optim` compile toujours un programme avec le comportement attendu dans le fichier .res pour les fichiers .deca dans codegen/valid. Nous avons également rajouté les fichiers .deca dans codegen/perf et dans codegen/extension/valid dans l'évaluation.

Résultat de la validation de l'extension

Les optimisations passent avec succès les tests de non-régression, il n'y a aucune fonctionnalité perdu par la modification de l'Arbre Abstrait et par la modification de la génération de code.

Performance

Nous avons utilisé les tests de performances fournies afin d'évaluer les performances de notre compilateur. Nous avons pris pour référence les performances de notre

algorithme avant optimisation. Voici les premières performances que nous avons mesuré (en cycles) :

- Syracuse 1682
- ln2 17112
- ln2_fct 20495

L'implémentation des algorithmes décrit ci-dessus nous a permis d'atteindre les performances suivantes (en cycles) :

- Syracuse 524
- ln2 6800
- ln2_fct 10401

En conclusion, voici le taux d'améliorations de notre algorithme pour ces 3 tests (rapport des temps d'exécution) :

- Syracuse 320 %
- ln2 252 %
- ln2_fct 197 %

Les gains de performances que nous avons pu obtenir pour ces algorithmes sont donc importants, ce qui améliore les temps d'exécution et diminue les ressources énergétiques nécessaires à l'exécution.

Références bibliographiques

[1] Jonathan A.: *Intermediate Code Generation* in *17-363/17-663: Programming Language Pragmatics.* https://www.cs.cmu.edu/~aldrich/courses/17-363/slides/lecture16-intermediate-generation.pdf.

[2] Pierre G.: *Part 5 Intermediate code generation.* https://people.montefiore.uliege.be/geurts/Cours/compil/2013/05-intermediatecode-2013-2014.pdf.

[4] Zhiru Z.: *Control Data Flow Graph* in *ECE 5997 Hardware Accelerator Design & Automation Fall 2021.* https://www.csl.cornell.edu/~zhiruz/5997/pdf/lecture04.pdf

[4] Aho, Alfred V.: *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

[5] Kenneth Z.: *The Development of Static Single Assignment Form.* https://compilers.cs.uni-saarland.de/ssasem/talks/Kenneth.Zadeck.pdf

[6] Michel S.: *SSA Form* in *Advanced Compiler Construction – 2008-05-23*. https://lampwww.epfl.ch/teaching/archive/advanced_compiler/2008/resources/slides/acc-2008-12-ssa-form.pdf

[7] Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K.: *Efficiently computing static single assignment form and the control dependence graph*, Oct 1991.

[8] How could I generate Java CFG(Control Flow Graph) using antlr? - Stack Overflow
Get Control flow graph from Abstract Syntax Tree - Stack Overflow
How to build a Control Flow Graph (CFG) from a JSON object (AST) - Stack Overflow

[9]. Matthias B., Sebastian B., Sebastian H., Roland L., Christoph M., Andreas Z.: *Simple and efficient construction of static single assignment form*, 2013.

[10]. Click C., Paleczny M.: *A simple graph-based intermediate representation*, 1995.

[11]. Aycock J., Horspool N.: *Simple generation of static single-assignment form*, 2000.

[12] Palak J., Jain S.: *Code Optimization in Compiler Design*, 2023. https://www.geeksforgeeks.org/code-optimization-in-compiler-design