Documentation de l'extension Optimisation

Table of Contents

Introduction

Deca est un langage objet faiblement typé, à partir duquel le compilateur Decac produit un code assembleur pour une machine particulière. Nommé IMA, cette machine possède des performances en cycles par instruction proche d'un microcontrôleur classique utilisé en systèmes embarqués. Puisque ce genre de système est soumis à de fortes contraintes, notamment en consommation d'énergie et en temps de réponse, la réduction du temps d'exécution et de la quantité de mémoire flash utilisée se sont rapidement révélées être d'une importance particulière pour garantir la compétitivité de Deca. L'optimisation a donc été intégrée dès le départ au cœur de la chaîne de compilation et garantie de très bonnes performances sur une grande variété de programmes Deca.

Du point de vue de l'utilisateur, il est possible de choisir d'activer ou de désactiver certaines optimisations en fonction des besoins. Il est possible de le faire via des options ajoutées à la compilation:

```
decac -Ox <fichier_source>
```

- L'option **-O0** ou aucune option engendre la compilation du fichier sans aucune optimisation particulière. Une bonne architecture de la génération de code permet d'avoir des performances tous de même très respectables.
- L'option **-O1** fait uniquement le calcul d'expressions constantes à la volée. Le temps de compilation ajouté est négligeable, mais le gain de performance peut-être conséquent.
- L'option **-O2** fait toutes les optimisations implémentées en utilisant un graphe de contrôle de flot. Une mise en forme SSA est appliquée, ce qui permet de ne n'avoir qu'une affectation par variable du graphe. Puis, les deux principales optimisations appliquées sur le graphe sont la propagation des constantes et la suppression de code mort. Cette optimisation n'est faite qu'à l'intérieur du main ou d'une méthode. Cette option permet d'avoir le gain en temps d'exécution le plus important, mais est aussi assez coûteuse en temps de compilation. De plus certaines variables peuvent être supprimées, donc l'écriture en mémoire du résultat n'est pas garanti.
- L'option **-Og** fait les mêmes optimisations que -O2 et génère en plus un fichier .dot pour graphviz, afin que le graphe puisse être lu visuellement. Nous y reviendrons plus tard.

Concepts

Les optimisations apportées à Decac peuvent se découper en 3 parties. La première est toujours utilisée, quelque soit l'option de compilation choisie pour l'optimisation. La deuxième est disponible avec les options **-O1** et **-O2**. Enfin, la dernière, qui est la plus ambitieuse, ne s'active qu'avec l'option **-O2**.

1. Gestion efficace des registres

La machine IMA possède un nombre de registres physiques limité, au maximum 16. Avec cette idée en tête, il devient nécessaire d'avoir une gestion efficaces de ces registres. Dans cette optiques, il a été choisi de ne copier une donnée en registre physique que s'il est vraiment nécessaire de le faire. En effet, on distingue plusieurs arguments en faveur d'une telle approche :

- La construction d'IMA garanti un faible coût des opérations mémoire, le fait d'en utiliser souvent n'est pas pas préjudiciable.
- Le mode d'adressage d'IMA permet d'avoir, par exemple dans le cas d'une opération arithmétique, un des deux opérandes adressé indirectement. De plus, le coût d'une telle opération est le même quelque soit le mode d'adressage. Ainsi, il semble plus intéressant d'utiliser un mode indirecte face à une copie en registre physique lorsque cela s'avère possible.
- La présence d'immédiats dont la taille peut atteindre le mot permet de les utiliser très souvent, et de retarder ou supprimer les copies d'immédiats en registres physique.

Ainsi, l'allocation des registres a été conçue pour n'allouer des registres physiques que si nécessaire. Une fois qu'une données est en registre physique, elle ne peut en sortir que si elle a été libérée ou s'il n'y a plus assez de registres pour stocker une donnée plus récente. La politique de gestion des registres physiques est décrite plus en détails dans **la documentation de conception, section génération de code**.

En plus de cette politique de gestion des registres, une optimisation supplémentaire a été ajoutée pour éviter la relecture d'une donnée qui vient juste d'être écrite. Cette problématique apparaît dans le cas suivant :

```
int x = a;
x = x + 1;
```

En l'absence d'optimisation particulière, le code assembleur produit est le suivant :

```
LOAD A(LB), R2
STORE R2, X(LB)
LOAD X(LB), R2
ADD #1, R2
STORE R2, X(LB)
```

On remque ici que le deuxième LOAD n'est pas nécessaire car R2 contient déjà la valeur contenue à l'adresse mémoire X(LB). Il a donc été décidé de supprimer ce LOAD dans ce cas particulier, on obtient alors le code suivant :

```
LOAD A(LB), R2
STORE R2, X(LB)
ADD #1, R2
STORE R2, X(LB)
```

2. Règles de réécriture et calcul d'expressions constantes

Ces optimisations sont utilisables avec les options **-O1** et **-O2**. Elle n'engendre pas de surcoût important sur le temps de compilation, mais garantissent un premier niveau d'optimisation intéressant. En effet, l'objectif des règles de réécriture et du calcul d'expressions constantes et qu'ils soient applicables directement sur l'arbre au moment de la génération finale de l'arbre, à la volée.

Les règles de réécritures désignent les différentes optimisations faites pour remplacer une instructions ou un groupe d'instructions très coûteux. Elles ne concerne, dans le cas de Decac, que les instructions **MUL , REM, QUO**, qui demandent beaucoup de cycles d'horloge pour être exécutées. On a alors les règles suivantes :

- **MUL** est remplacé par une série de décalage à gauche si les opérandes sont de type entier, et qu'au moins un d'entre eux est un immédiat correspondant 2^i, avec $i \in [1, 10]$ entier. on obtient alors, pour **4*x** :

```
LOAD X(LB), R2
SHL R2
SHL R2
```

- **QUO** est remplacé par une série de décalage à droite si les opérandes sont de type entier, et que le diviseur est un immédiat correspondant 2^i, avec $i \in [1, 10]$ entier. On obtient alors, pour **x/4** :

```
LOAD X(LB), R2
SHR R2
SHR R2
```

- **REM** est remplacé par une série de décalage à droite puis à gauche et enfin une soustraction, si les opérandes sont de type entier, et que le diviseur est un immédiat correspondant 2^i, avec $i \in [1, 5]$ entier. Dans ce cas, on applique i décalages à droite suivis de i décalages à gauche afin de retirer le reste de la dividende. Puis, une soustraction finale entre la dividende et ce qui vient d'être calculé permet de récupérer le reste. On obtient alors, pour **x%4** :

```
LOAD X(LB), R2
SHR R2
SHR R2
SHL R2
SHL R2
SUB X(LB), R2
```

Le calcul d'expression constantes se fait de façon récursive sur l'arbre, il permet d'éviter d'avoir des instructions arithmétiques ou booléennes lorsque les opérandes sont des immédiats, donc des constantes. Un algorithme récursif permet de propager localement les constantes à travers toute l'expression, jusqu'à ce quelle soit entièrement calculée ou qu'une variable n'apparaisse. On obtient alors, pour **x=1+2+3** :

```
STORE #6, X(LB)
```

3. Contrôle de flot

Cette partie décrit les optimisations qui ont été implémentés grâce au contrôle de flot. Il s'agit d'optimisations beaucoup plus ambitieuses que les précédentes, puisqu'elles interviennent avant la génération finale du code. La première étape consiste à transformer la liste d'instructions en un graphe dit *de contrôle de flot*. Puis un premier traitement est appliqué sur le graphe pour le transforme en forme *SSA*, c'est à dire une forme où chaque variable n'est affectée qu'une seule fois. Puis, deux principales optimisation sont appliqués au graphe : la propagation de constantes et la suppression de code mort. Enfin, le graphe est parcouru une dernière fois pour générer le code avec les composants déjà présents.

1. Création du graphe de contrôle de flot

Un graphe de contrôle de flot est une représentation du programme alternative à l'arbre abstrait. Il s'agit d'un graphe orienté, où les nœuds contiennent une liste d'instructions exécutées linéairement, et les arcs représentent le flot du programme. Les structures de contrôles *if* et *while* engendrent des blocs ayant plusieurs arcs sortants, un par branche. Afin d'illustrer toute l'optimisation du contrôle de flot, un exemple de programme va être suivi durant et traité dans les différentes sous-parties.

L'implémentation actuelle du contrôle de flot ne s'applique que sur les blocs de code, c'est à dire l'ensemble es instructions qui composent le main ou une méthode. Aucune gestion de contrôle de flot n'est appliquée sur les objets ou les appels de méthodes. Cela représente néanmoins une piste sérieuse d'amélioration possible.

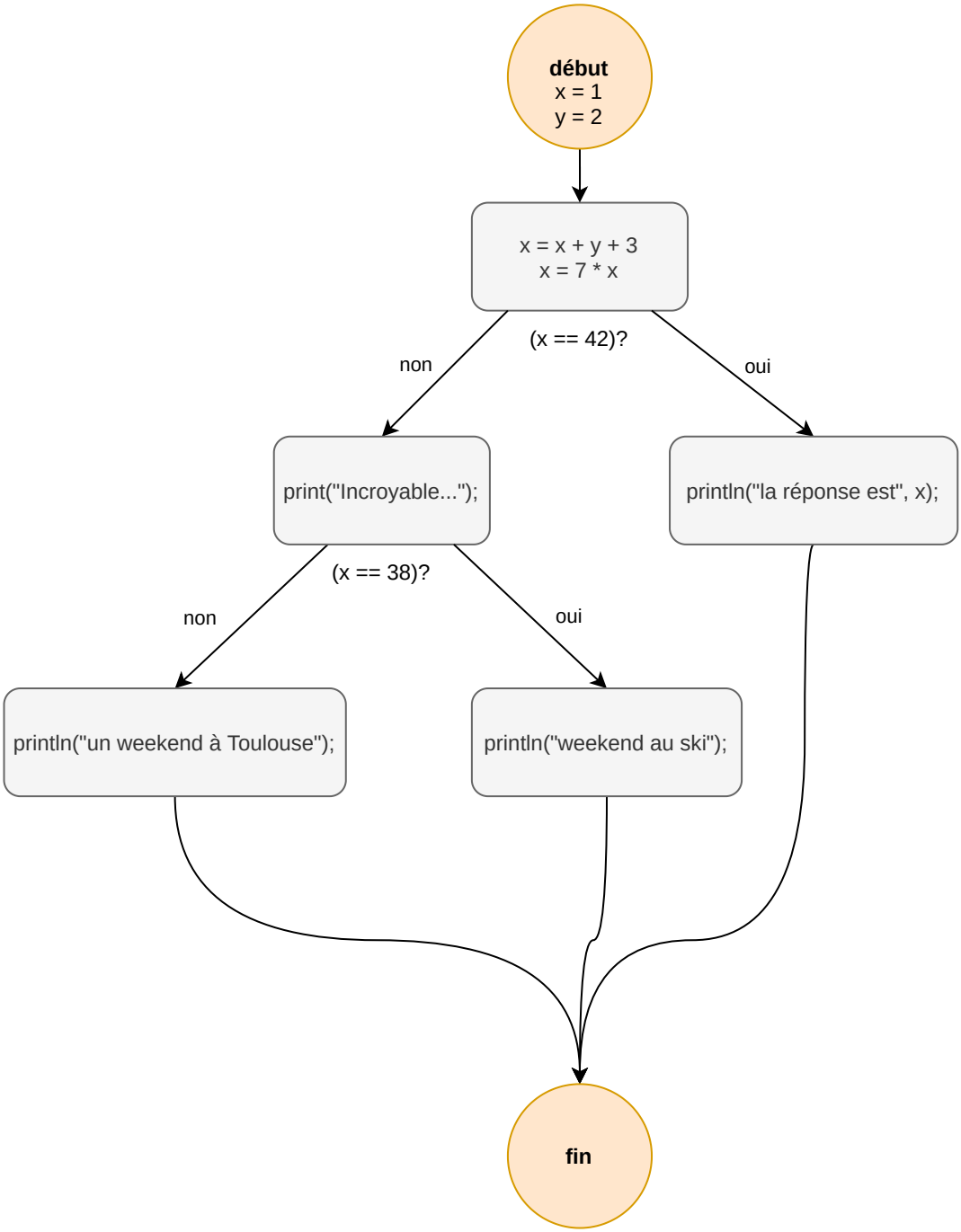
Le programme traité pour l'exemple est le suivant :

```
{
    int x = 1;
    int y = 2;

    x = x + y + 3;
    x = 7 * x;

    if (x == 42) {
        println("la réponse est : ", x);
    }
    else {
        print("Incroyable vous êtes le 1000000e visiteur, rentrez votre numéro de carte
bancaire ici pour gagner un ");
        if (x == 38) {
            println("weekend au ski !");
        }
        else {
            println("weekend à Toulouse !");
        }
    }
}
```

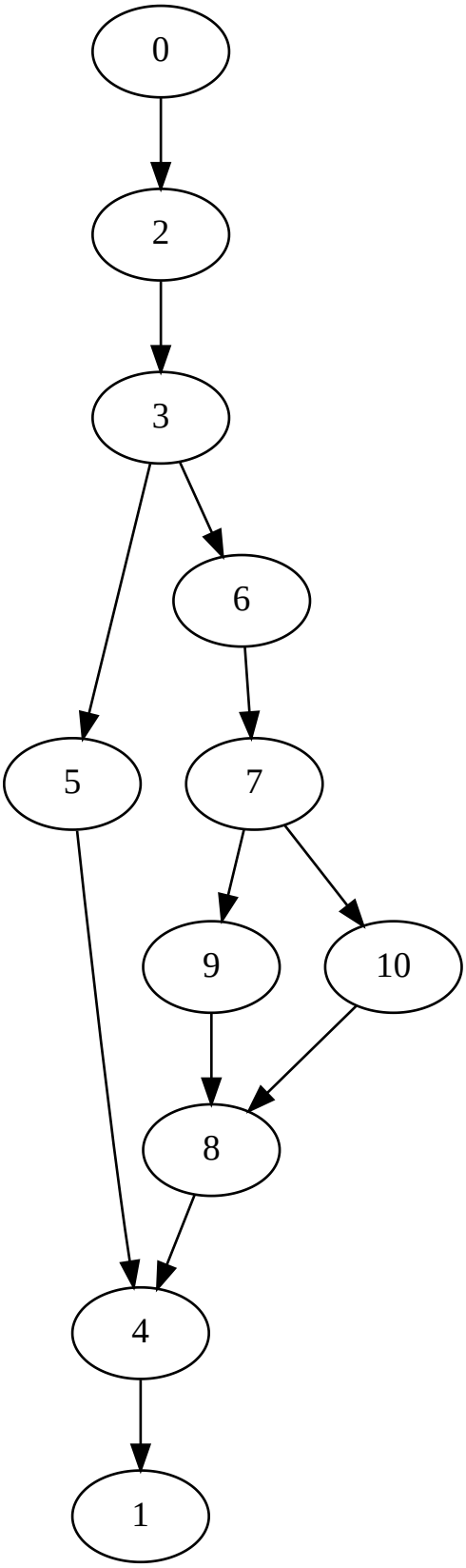
La première étape consiste donc, à partir de l'arbre, à créer le graphe de contrôle de flot. En le dessinant manuellement, on obtient :



Il est également possible d'obtenir ce graphe de façon automatique, en utilisant l'option **-O2g**. Un fichier **.dot** sera alors produit à côté des fichiers **.deca** et **.ass**. Il est alors possible d'utiliser **graphviz** pour obtenir une image de graphe., et l'afficher avec un visionneur d'image. Une fois le paquet installé, la commande à utiliser est par exemple :

```
dot -Tsvg fichier.dot > fichier.svg
```

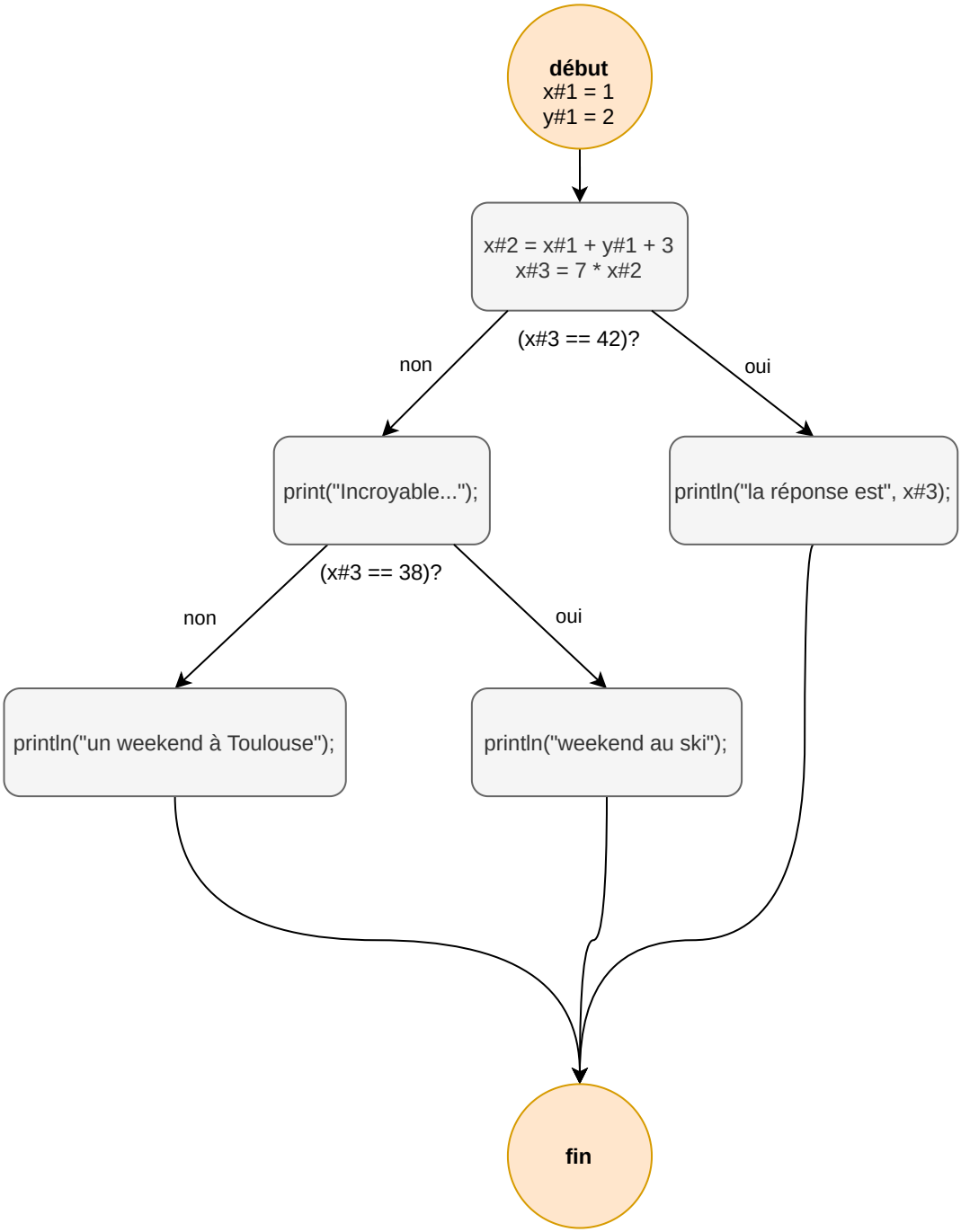
Pour cet exemple, le résultat est alors :



On retrouve alors la même structure, mais certains blocs vide ont été ajoutés par soucis de simplification des algorithme de création du graphe.

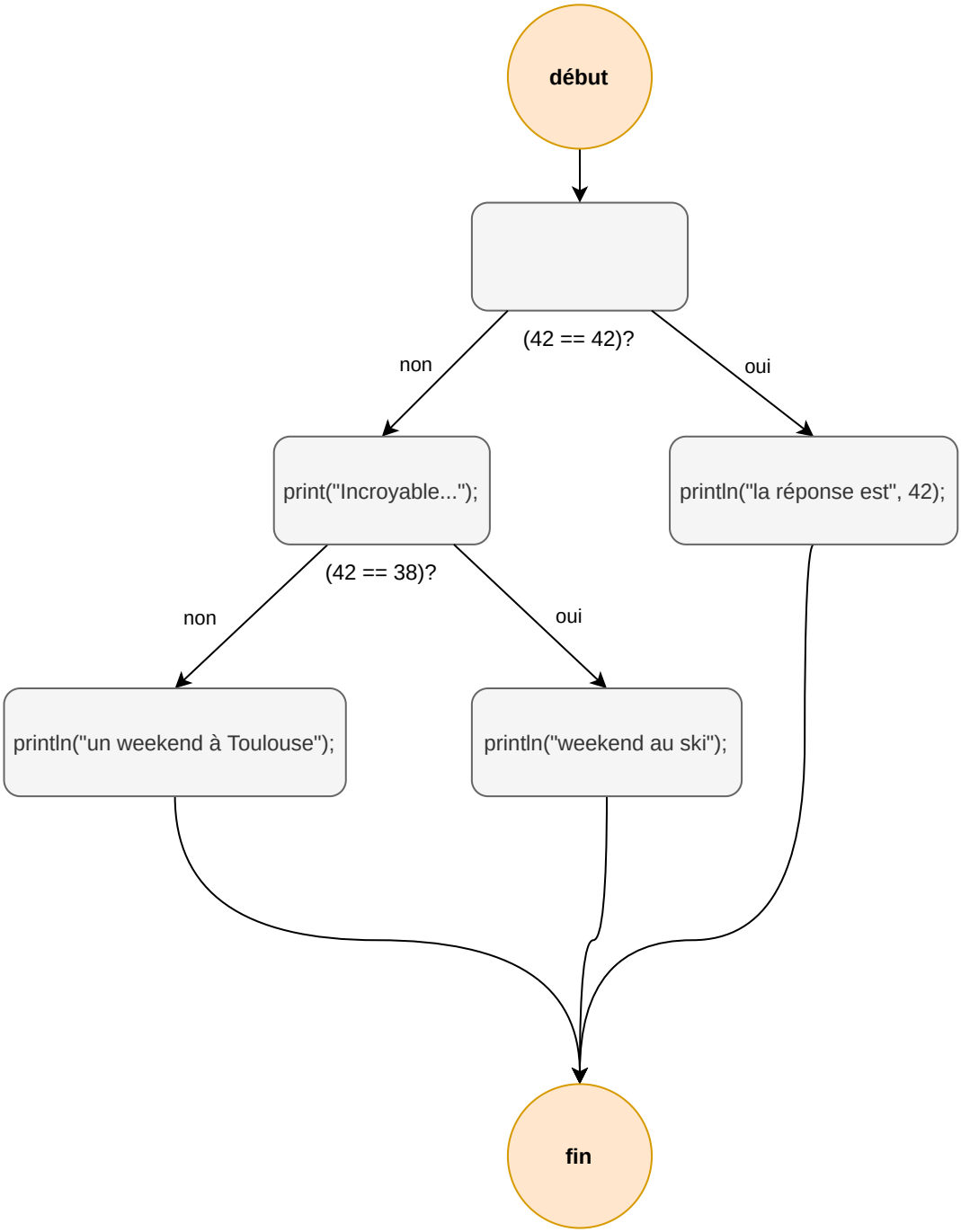
2. mise en forme SSA

Il s'agit du premier algorithme appliqué au graphe nouvellement créé. SSA signifie "Static Single Assignment" .L'objectif est de n'avoir qu'une seul affectation de par variable. Pour cela, on notera **<nom_de_la_variable>#id** chacune des variables du programme. Le nom est celui de la variable d'origine, et **id** est un numéro unique donnée à l'affectation unique. Pour cet exemple, on obtient :



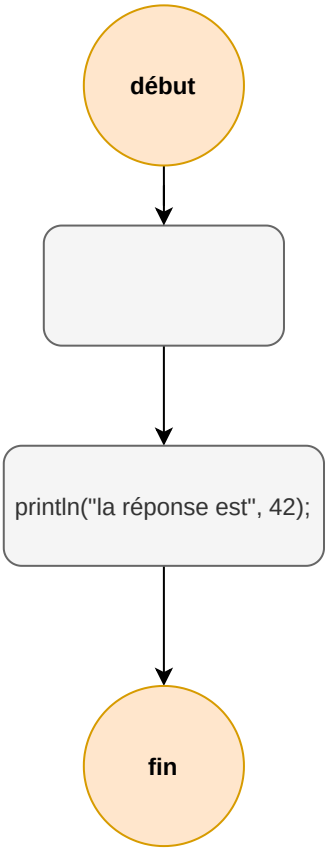
3. propagation des constantes

Cet étape est la première des deux optimisations principales implémentées. Il s'agit, pour chaque variable en forme SSA, de vérifier si l'expression qu'elle représente est une constante. Dans ce cas, on remplace alors chacune des endroits où apparaît cette variable par la constante associée. Puis, l'algorithme est réitéré pour toutes les instructions qui contenaient cette variable, en recalculant les expressions pour trouver d'autres constantes. Cela permet alors de réduire le nombre d'instructions exécuté et donc le temps d'exécution. Sur cet exemple, on obtient :

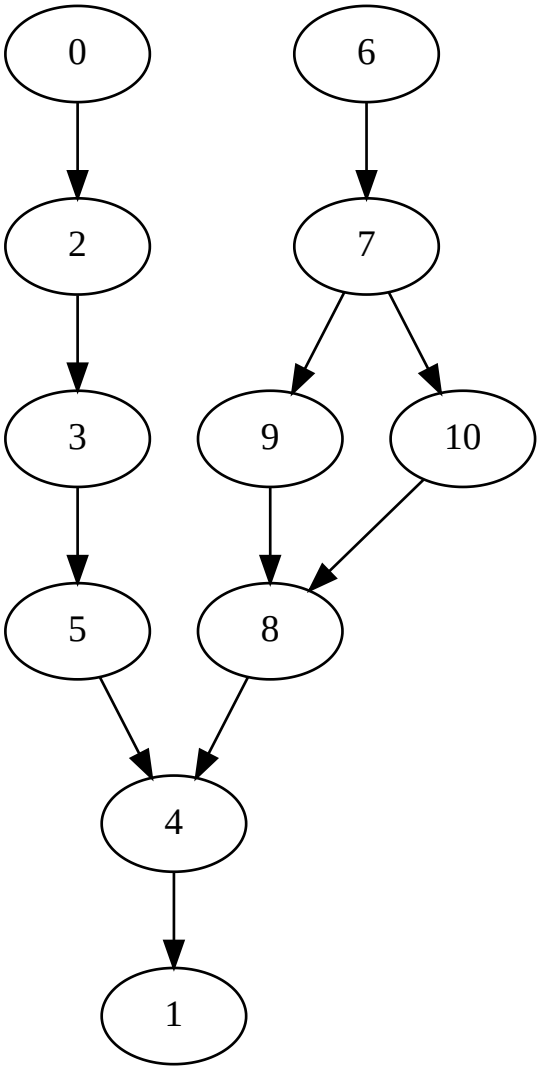


4. suppression du code mort

Cette deuxième et dernière étape de l'optimisation sur le contrôle de flot consiste à supprimer le code rendu inaccessible. cela permet de réduire la quantité de mémoire occupée par le résultat. Il s'agit de tester les conditions sur les blocs possédant plusieurs arcs sortants. Si la condition peut être évaluée sans exécuter le programme, seule la branche prise sera conservé. On obtient alors :



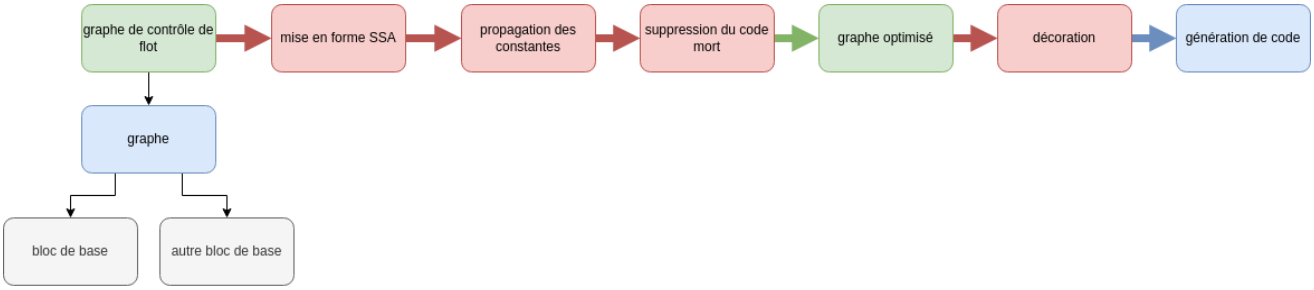
En réalité il suffit de supprimer l'arc sortant vers la mauvaise branche pour la supprimer, puisque la génération de code se fait en parcourant le graphe. La branche non prise est alors inaccessible. Le graphe produit par **-O2g** intègre de base cette suppression de code mort, donc il affichera :



Architecture logicielle

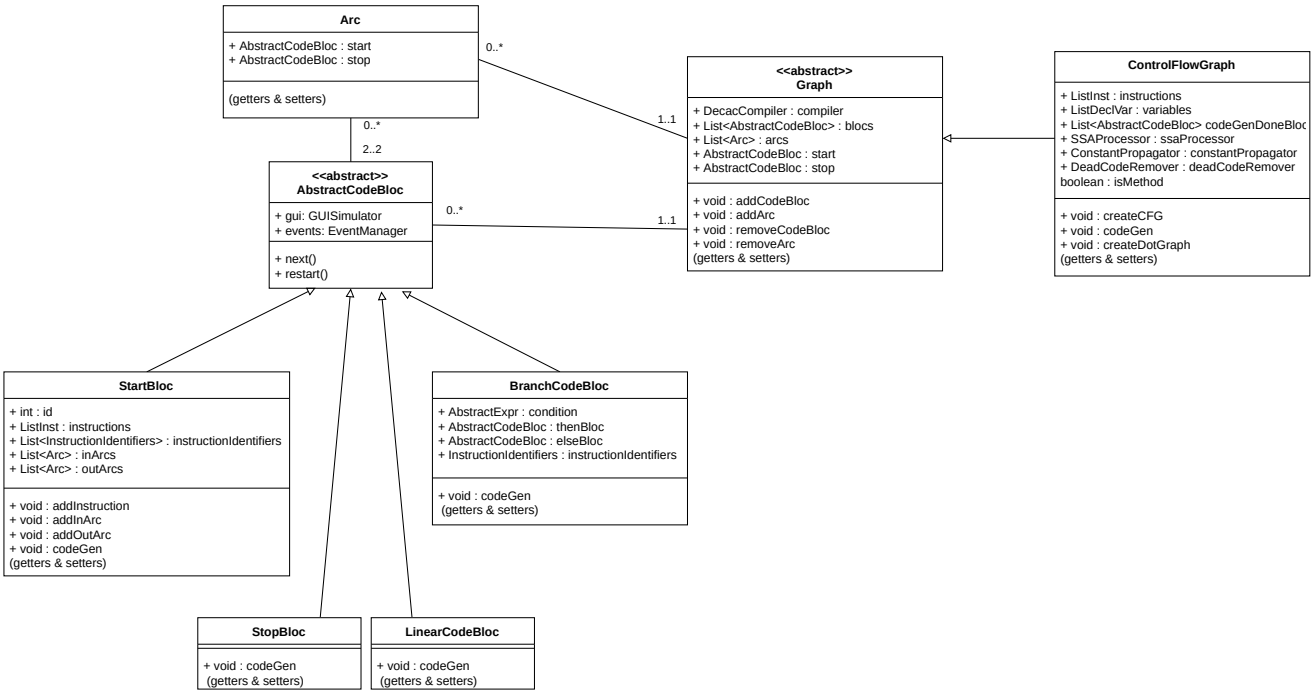
Les deux premiers types d'optimisation présentés plus haut ont leurs implémentations directement intégrées dans la génération de code. La gestion des registres est notamment présentée dans la **documentation de conception, section génération de code**. Une exemple de règle de réécriture et le calcul d'expression constante est aussi donnée dans la sections **Deep dive** de cette documentation.

En ce qui concerne l'optimisation par contrôle de flot, un package nommé **opti** a été ajouté aux sources de Decac. Étant donné que l'enchaînement des différentes étapes est linéaire, il est possible d'obtenir une vue globale des principaux composants sous forme de pipeline.



1. Graphe de contrôle de flot

Un graphe de contrôle de flot étant un graphe orienté, il y a par conséquence une classe **Arc**, une classe **Graphe** au sein du package. Les blocs peuvent alors 4 types (bloc de début, bloc de fin, bloc linéaire, bloc de branchement) qui héritent tous de la même classe **AbstractCodeBloc**. Le graphe de contrôle de flot hérite de la classe **Graphe** et ajoute de nombreux éléments spécifiques au contrôle de flot.

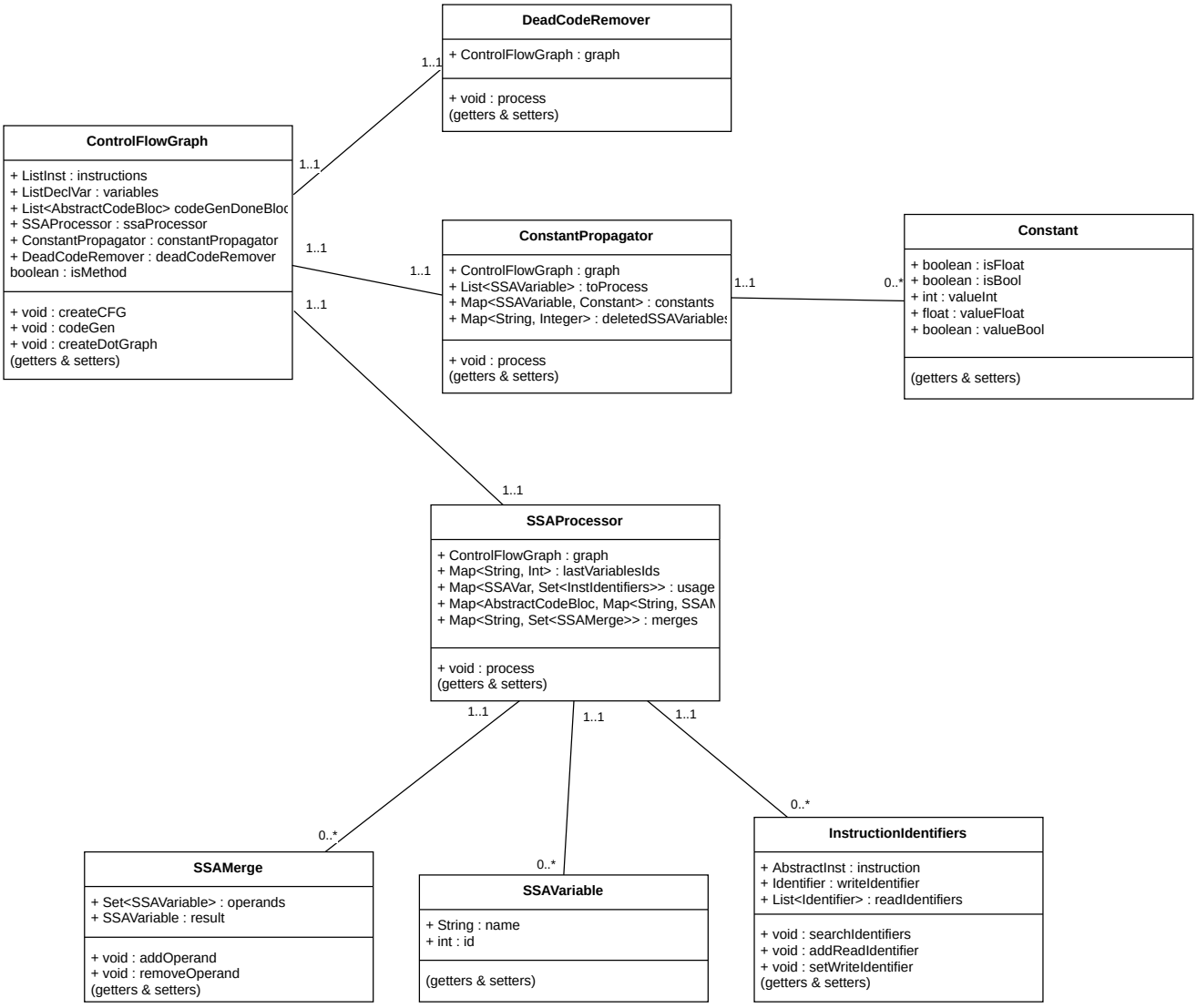


2. Traitement du graphe

Les différents traitements à faire sur le graphe (mise en forme SSA, propagation des constantes, suppression du code mort) sont effectuées par des instances des classes correspondantes aux différents traitement. Il y a donc une classe par traitement, qui sont créer et le traitement fait dans le constructeur de **ControlFlowGraph**. Elles portent ainsi les noms :

- SSAProcessor
- ConstantPropagator
- DeadCodeRemover

Les classes **Constant, SSAVariable, SSAMerge, InstructionIdentifiers** ont été créés pour décorer le graphe ou faciliter les différents traitements.



Deep dive

Alors que la section précédente se concentrais sur l'architecture des différentes optimisations mise en œuvre, il semble intéressant d'en expliciter certains points clés, sur lesquels repose l'implémentation réalisée.

- **Suppression des LOAD inutiles**

  Afin d'implémenter cette optimisation détaillée dans la section **Concepts**, et connaissant le fonctionnement de la gestion des registres détaillé dans la **documentation de conception**, il a été nécessaire d'ajouter deux champs dans la classe **ContextManager** :

  ```
  private VirtualRegister lastStoreRegister = null;
  private RegisterOffset lastStoreOffset = null;
  ```

  Les valeur des ces champs sont modifiés à travers des setters par la classe **AssignOperation** juste après avoir généré l'instruction **STORE** d'une affectation. Ainsi, lorsque la méthode **freePhysicalRegister** de **ContextManager** est appelée, ce registre sera gardé et ne sera pas détruit. Puis, dès lors qu'une nouveau registre est demandé, il y a deux cas possible. Soit la donnée correspond à la donnée qui vient d'être stockée en mémoire, et il est alors possible de reprendre le registre en attente sans effectuer de **LOAD**, soit ce n'est pas le cas et le registre en attente est réellement détruit avant de créer le nouveau.

- **Calcul d'expression constante**

  Cette optimisation est utilisée à deux moments. Le premier est lors de la propagation de constantes, le second est lors de la génération finale du code. Dans les deux cas, la méthode ***Constant getConstant(DecacCompiler compiler)*** est utilisé pour cela. Présente dans toutes les classes héritant de **AbstractOperation** et **AbstractExpr** sur l'arbre, elle permet, comme la génération de code, de procéder récursivement sur les nœuds et jusqu'aux feuilles de l'arbre représentant l'expression à essayer de calculer. Pour les fonctions feuilles, la valeur d'un littéral remonte ainsi, et c'est aussi le cas pour les identifiant de variable ayant été marqué par la propagation de constante comme constant avec la méthode **setConstant**. Puis, dans le cas des opérations arithmétiques, booléennes, ou de comparaison, le calcul est directement effectuée dans le code Java, et le résultat remonte alors. Dans le cas où il n'est pas possible de calculer de constante, par exemple lorsqu'il y a des variables non connues ou une lecture d'entrée utilisateur, la valeur retournée est **null**.

  L'appel à la méthode **getConstant**, lors de la génération finale du code, est fait dans les méthodes **doOperation** des classes héritant de **AbstractOperation**. Si le resultat est différent de **null**, la constante est alors mise dans la pile d'opérations du **ContextManager**. Dans le cas contraire, le calcul est effectuée de façon classique.

- **Création du graphe de contrôle de flot**

  La création du graphe de contrôle de flot est faite en appelant le constructeur de **ControlFlowGraph** dans la méthode responsable de la génération de code du **main** ou d'une **méthode**. De plus, il est nécessaire que l'option **-O2**. L'option **-O2g** va, en plus de créer le graphe, produire un fichier **.dot** en appelant méthode **createDotGraph**. La construction du graphe, à partir de la liste d'instruction, est faite à travers la méthode **createCFG** appelé par le constructeur de **ControlFlowGraph**. La liste d'instruction est alors découpée récursivement à chaque qu'un branchement est à ajouté, en utilisant la méthode **CFGRecursion**.

- **Mise en forme SSA**

  Cette étape est réalisée entièrement par la classe **SSAProcessor**. L'appel au constructeur initialise les structures internes utilisées par l'algorithme principal. Ce dernier est exécuté lors de l'appel à la méthode **process**. Les structures à connaître sont les suivantes :

  - ```
    private final Map<String,Integer> lastVariablesIds;
    ```

    il s'agit d'une table permettant, pour une variable locale (les champs sont exclus et ne sont pas traités) identifiée par la chaîne de caractère représentant son nom, de retrouver la dernier *ID* attribué à une variable en forme *SSA*. Ce sont simplement des compteurs qui sont incrémentés lorsqu'une variable est à nouveau affectée et qu'un *ID* est demander pour la variable *SSA* produite.

  - ```
    private final Map<SSAVariable, Set<InstructionIdentifiers>> usages;
    ```

    Il s'agit d'une table permettant, pour une variable en forme *SSA*, de récupérer l'ensemble des **InstructionIdentifiers** contenant cette variable en particulier. Comme chaque **InstructionIdentifiers** est associé à une instruction, cela permet de savoir quelles instructions utilisent, aussi bien en lecture qu'en écriture, cette variable.

  - ```
    private final Map<AbstractCodeBloc, Map<String,SSAMerge>> waitingFusionCodeBlocs;
    ```

    Il s'agit d'une table permettant de stocker temporairement la liste des blocs à fusionner. Elle permet alors, pour un bloc en attente de fusion, de récupérer la table des fusions en cours associées à chaque variable du programme, identifiée par la chaîne de caractère représentant son nom.

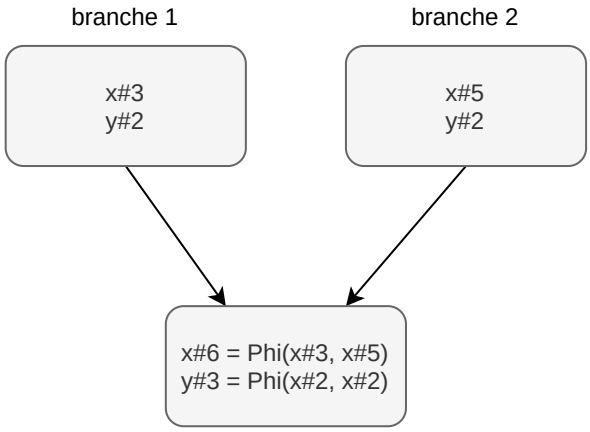
  - ```
    private final Map<String, Set<SSAMerge>> merges;
    ```

    Il s'agit d'une table permettant, pour chaque variable du programme identifiée par une chaîne de caractère représentant son nom, d'obtenir l'ensemble des fusions qui lui sont associées.

  - ```
    Map<String,SSAVariable> localSSA
    ```
    ```

Il s'agit d'une structure local aux méthodes utilisées par l'algorithme principale. Étant donné que le graphe est parcouru de proche en proche, cette table est copiée et chaque copie est transmise aux nœuds enfants, qui vont la muter, et ainsi de suite. Elle permet, pour chaque variable du programme, de récupérer la variable en forme *SSA* courante. Celle-ci est utilisée lorsque qu'une lecture est trouvée, et changer si une nouvelle affectation de la variable est détectée.

L'algorithme principal, dans la méthode **process**, initialise **localSSA** en appelant **processDeclVar**. Cela permet d'associer une première variable **SSA** à chaque variable déclarée. Puis tous les blocs sont traités de proche en proche grâce à la méthode **processBloc**. Il est important de bien copier entièrement **localSSA** si un nœud possède deux branches sortantes, pour rendre les copies indépendantes. La méthode **processBloc** a la charge de repérer les bloc avec plusieurs arcs entrants, qu'il faut donc fusionner. Lorsque c'est le cas, un opérateur souvent nommée ϕ dans la littérature est à appliqué. Il symbolise la fusion des variables **SSA** lorsque ce ne sont pas les mêmes qui viennent des branches entrantes. Ici, ϕ est représenté par la classe **SSAMerge**. Le rôle des méthode **startMerge, continueMerge, endMerge** appelées par **processBloc** lorsqu'une fusion est détectée permet que créer, enrichir, ou terminer les instances de **SSAMerge** nécessaires pour représenter toutes les fusions à faire. Par soucis de simplification des algorithme, une instance de **SSAMerge** est créée pour chaque variable, même s'il n'y a pas de conflit sur les deux branches. Cela évite de traiter des cas particuliers. Pour un exemple simple, on obtient :



En réalité, ces opérateurs ϕ ne possèdent pas d'implémentation lors de la génération finale du code. Les étapes d'optimisation suivantes permettent d'en supprimer une partie, lorsque les variables qu'ils contiennent auront été remplacées par des constantes. Ceux qui resteront seront traité en ne gardant qu'une case mémoire unique pour tous les opérandes. Ainsi, quelque soit la branche prise, la valeur dans cette case sera la bonne. Cela peut impacter la performances des processeurs de PC modernes, qui font de l'exécution spéculative, mais ce n'est pas le cas d'IMA.

Enfin, pour chacune des instructions du bloc en cours de traitement, la méthode **processInstructionIdentifiers** est appelée avec en argument l'instance de classe **InstructionIdentifiers** associée à l'instruction, afin de traiter tous les usages de variables au sein de cette dernière, et ce sans avoir à reparcourir l'arbre.

- **Propagation des constantes**

  L'étape de propagation des constantes est faite dans la classe **ConstantPropagator**. le constructeur initialise les structures internes et la méthode **process** permet de traiter le graphe et propager les constantes. Une mise en forme **SSA** du graphe doit être faite précédemment, car l'algorithme utilisé repose sur le fait que cette propriété soit vérifiée. Les structures à connaître sont les suivantes :

  - ```
    private final List<SSAVariable> toProcess;
    ```

    Il s'agit d'une list, permettant de stocker temporairement toutes les variables en forme *SSA* qu'il reste à traiter.

  - ```
    private final Map<SSAVariable, Constant> constants;
    ```

    Il s'agit d'une table utilisée pour connaître la constante associée à chaque variable en forme *SSA* lorsque c'est les cas. Cette structure est utilisée de manière conjointe avec la structure précédente. En effet, si une variable est détectée comme constante, la constante est mise dans la table et la variable dans la liste des variables à traiter.

- ```
  private final Map<String, Integer> deletedSSAVariables;
  ```

  Il s'agit d'une table permettant, pour chaque variable du programme original identifiée par la chaîne de caractère représentant son nom, de retrouver le nombre de variables en forme *SSA* associées qui ont été détruites et transformées en constantes. Si ce nombre est égal à **lastVariablesIds** de la classe **SSAProcessor**, alors toutes les variables *SSA* ont été simplifiées, donc la déclaration et la case mémoire associées à la variable, peuvent êtres supprimées.

L'algorithme se décompose en trois parties principales. Les deux premières ont pour objectif de repérer les expressions constantes, respectivement dans les déclaration de variables et les affectations. Cela est fait en utilisant la méthode **getConstant** de **AbstractExpr** et en effectuant de nombreuses vérifications d'usages, relatives à la portées des variables par exemples. Les deux méthodes associées à cela sont **foldDeclVar** et **foldAssign**. Puis la troisième partie, directement codée dans la méthode **process** a pour rôle de propager les constantes précédemment calculées, de recalculer les expressions une fois la propagation faite, et de recommencer tant qu'il reste des variables *SSA* à propager dans la liste **toProcess**. Une gestion particulière des **SSAMerge** est aussi nécessaire. La technique utilisée consiste à supprimer chaque opérande détecté comme constante des instances de **SSAMerge** associées, puis de regarder le nombre d'opérandes restants. S'il est égal à 0 ou 1, alors le résultat des totalement déterminé et la fusion peut être simplifiée. Enfin, lorsque la propagation est terminée, une dernière étape est faite dans la méthode **postProcess**, dont l'objectif est de supprimer les déclarations de variables devenues inutiles.

- **Suppression du code mort**

  La suppression de code mort, deuxième et dernière optimisation implémentée, est entièrement faite dans la classe **DeadCodeRemover**. Plus simple que les traitements précédents, celui-ci ne possède qu'une méthode importante, *process*, qui doit être appelée après avoir instancié la classe. De plus, il n'y a pas de structures particulière à ce traitement.

  L'algorithme consiste à parcourir le graphe de proche en proche pour chercher les blocs de branchement. Lorsqu'un est détectée, une évaluation constante de la condition est tentée, avec la méthode **getConstant** de **AbstractExpr**. En cas de succès, la branche à prendre est connue, donc il est possible transformer le bloc en un bloc linéaire et supprimer l'arc vers la branche inutile. Devenue non accessible, aucun code ne sera généré pour cette branche. L'algorithme s'arrête lorsque tout le graphe a été parcouru.

Benchmark

Bibliographie
  ```