

Projet Génie Logiciel

Documentation de l'extension

ENSIMAG 2ème année - 2022-2023

Groupe 1 - équipe 3

Jorge Luri Vañó
Nils Depuille
Vianney Vouters
Virgile Henry
Logan Willem

I. Sommaire

[I. Sommaire](#)

[II. Introduction](#)

[III. Fonctionnement de l'extension](#)

[VI. Élimination des variables inutiles](#)

- 1) [Repérer les variables utilisées depuis le main](#)
- 2) [Repérer les variables utilisées depuis les méthodes](#)
- 3) [Repérer les champs "override" de champs utiles](#)
- 4) [Élimination des variables inutiles](#)
- 5) [Remise à zéro de l'attribut utilisé](#)

[VI. Résolution des calculs évidents](#)

[VII. Factorisation des calculs](#)

[VIII. Élimination des calculs redondants](#)


[IX. Optimisation Post Génération](#)

- 1) [Description de l'optimisation](#)
- 2) [Architecture utilisée](#)
- 3) [Traitement des boucles](#)
- 4) [Traitement des ifs](#)
- 5) [État actuel de l'optimisation](#)

[X. Ressources](#)

II. Introduction

L'objectif principal du projet était de réaliser un compilateur pour le langage déca. En plus de cela, nous avons le choix entre un certain nombre "d'extensions" pour explorer d'autres enjeux du compilateur. D'un commun accord, nous avons fait le choix de travailler sur l'optimisation du compilateur pour améliorer les performances du code assembleur produit. Avec ce choix, nous avons voulu découvrir les différentes techniques utilisées par les compilateur, en comprendre les enjeux, les possibilités comme les limites. A l'heure où le numérique se doit d'être plus sobre c'est une dimension qu'il est nécessaire de prendre en compte dans notre métier d'ingénieur.



Cette optimisation s'est articulée autour de 2 axes : le nombre de cycles à l'exécution et la taille du code assembleur. Nous sommes en fait parti du principe que le code compilé pouvait être utilisé dans de l'électronique embarqué ou pouvait être déployé à grande échelle ce qui rend pertinent de faire davantage de calculs à la compilation pour pouvoir en faire moins ensuite à l'exécution ou pour pouvoir déployer le code compilé sur des machines aux ressources plus limitées. Toutefois, nous pensons que cette optimisation code doit rester optionnelle pour ne pas accroître le nombre de calculs inutiles pour des applications qui n'en valent pas la peine. Nous laissons donc à l'utilisateur le choix d'utiliser l'optimisation grâce à l'option "-o" de la commande decac.

Suite à nos recherches, nous avons en tête 6 techniques d'optimisation à explorer :

- Fenêtre "peephole"
- Elimination des variables inutiles
- Résolution des calculs évidents
- Factorisation des calculs
- Elimination des calculs redondants
- LLVM


Après négociation avec le client, il a été convenu d'implémenter les 4 premières techniques pour le rendu du 23 janvier.

III. Fonctionnement de l'extension

Comme indiqué précédemment, l'utilisateur a le choix d'utiliser ou non l'optimisation du code grâce à l'option "-o" de la commande decac. Cette option utilise la totalité des techniques présentées ci-dessous.

L'optimisation du code consiste en deux temps :

- Tout d'abord, on effectue plusieurs passes d'optimisation sur l'arbre généré à la manière des passes de la vérification contextuelle. Les passes d'optimisation sont réalisées les unes à la suite des autres tant que des optimisations sont réalisées. Nous avons en fait considéré le fait que la réalisation d'une technique d'optimisation permettra à une autre d'avoir à nouveau des optimisations à réaliser, et donc qu'il était possible de continuer d'optimiser du code tant qu'une optimisation effectuait encore des changements.
- Ensuite, une fois l'arbre optimisé et la génération du code effectué, il est possible de repasser sur le code assembleur généré pour l'optimiser lui aussi. En effet, la génération du code effectué depuis l'arbre est relativement abstraite, et peut générer du code non



optimal. Par exemple, une simple assignation va d'abord charger un immédiat dans un registre, puis charger ce registre dans la variable à assigner, alors qu'il est parfois possible de directement charger un immédiat dans une variable sans passer par un registre intermédiaire.

IV. Résolution des calculs littéraux

Le principe de résolution des calculs littéraux est de calculer à la compilation toutes les occurrences de calculs dont on connaît le résultat par leur valeur. Cela permet de ne pas avoir à effectuer les calculs lors de l'exécution du programme.

Pour cela, on a créé des méthodes nommées “collapse” qui permettent de réduire les calculs évidents. L'idée est que chaque instruction ou expression peut se simplifier en quelque chose de plus rapide à exécuter, si on connaît certains résultats à l'avance. Le principe de base de ces méthodes est d'être récursives et d'appliquer ainsi la simplification sur tout l'arbre, puis de remonter les résultats souhaités. Ainsi, si un appel à une fonction collapse remonte l'information qu'une expression ou instruction peut se simplifier, le parent qui a appelé la méthode sur ses enfants peut remplacer le nœud par un littéral.

Pour écrire cette fonctionnalité de la façon la plus simple et claire possible, on a créé une classe générique, “CollapseResult<T>”, qui contient deux informations :

- un booléen qui permet de savoir si à un moment, on a effectué une simplification. Cela permet de faire remonter tout changement jusqu'à l'appel principal de cette optimisation, et ainsi continuer d'optimiser comme expliqué précédemment.
- Un générique qui est ce en quoi l'objet sur lequel on a appelé notre méthode peut se simplifier. Par exemple, une opération entière peut se simplifier en entier, une instruction peut se simplifier en liste d'instructions (un “if” est une seule instruction, mais si on arrive à connaître sa condition à l'avance, il se simplifie en sa liste d'instruction “then” ou “else”).

Finalement, tous les nœuds non terminaux vont appeler “collapse” sur leur enfants. Les terminaux littéraux vont dire qu'ils sont en effet simplifiable, et les non terminaux, selon si leur enfants sont simplifiables ou non, vont également se dire simplifiable. De plus, lorsqu'un nœud voit un de ses enfants comme simplifiable, il le remplace par la valeur remontée par le “CollapseResult<T>”.

V. Élimination des variables inutiles

L'idée de cette optimisation est d'éliminer du programme les variables, les méthodes, les champs et les classes qui ne sont pas utilisés pour éviter des calculs inutiles et libérer de la place sur la pile. Dans la suite, si rien n'est précisé, nous regroupons sous le terme variables les variables, méthodes, fields et classes. Cette optimisation est réalisée directement sur l'arbre abstrait à la suite de la vérification contextuelle et de la décoration de l'arbre. Elle se décompose en 2 étapes principales : repérer les variables utilisées puis éliminer celles qui ne le sont pas. La première étape se fait en 3 passes: une à partir du "Main", une pour le traitement du corps des méthodes utiles et des méthodes "Override" et une dernière pour les champs "Override". La deuxième étape se fait quant à elle en une seule passe.

1) Repérer les variables utilisées depuis le main

Les variables (variables, méthodes, champs et classes) sont représentées dans le programme par une instance "identifier" dans l'arbre abstrait. Cet "identifier" possède comme attribut une instance "définition" auquel on ajoute un attribut boolean "spotted" qui permet de savoir si "l'identifier" est utilisé dans le programme. On considère comme utilisées les variables qui sont utilisées pour un "print()", un "MethodCall", une condition de "IfThenElse" ou de "While", une "Initialization" et un "Return". Par ailleurs, si une classe est utilisée alors ses superclasses le seront aussi. Il est important de noter que nous explorons que l'initialisation des déclarations de variable et uniquement la partie droite des assignations. En ce qui concerne les "instance of", nous n'explorons que l'expression et pas le type. L'objectif est d'ensuite pouvoir évaluer à faux l'expression si le type n'est jamais utilisé (lors d'un "new" par exemple).

Pour repérer ces variables, une première passe est réalisée récursivement depuis le nœud "Main" de l'arbre abstrait "Program" grâce aux méthodes "spotUsedVariables()". On positionne ainsi à "true" l'attribut "spotted" de l'ensemble des variables utilisées directement dans le main.

2) Repérer les variables utilisées depuis les méthodes

Une fois que les variables utilisées dans le "Main" ont été repérées, nous réalisons une passe supplémentaire pour repérer les variables utilisées dans le corps de méthodes déjà repérées. On établit donc un "Set" des méthodes à étudier que l'on va parcourir en boucle tant que de nouvelles variables ont été repérées. Lorsqu'une méthode est étudiée, son corps est exploré si elle a été "spotted" ou qu'elle est un "override" d'une méthode utilisée et que sa classe est utilisée. Elle est ensuite retirée du "Set".

Le cas des méthodes non repérées mais qui sont des "override" de méthode utile est assez vicieux. Une telle méthode peut en fait devenir utile à l'exécution si l'objet qui appelle la méthode est une instance d'une sous-classe de la classe statique. Il faut donc repérer ces méthodes lorsqu'elles appartiennent à des classes qui ont été repérées comme utiles par un "New" ou un "Cast". On repère si une méthode est

un “override” à partir de sa définition en remontant la hiérarchie de classes avec une table associant à chaque définition de classe un “Set” des indices des méthodes qui sont utilisées.

Nous aurions pu réaliser la passe de repérage des méthodes utiles statiquement (non “Override”) en même temps que la passe du “Main” en passant en paramètre de “spotUsedVar()” une table associant à chaque définition de méthode “l’AbstractMethodBody” associé. Il aurait aussi été possible de donner un attribut “DeclMethod” aux définitions de méthodes mais nous avons préféré garder une distinction claire entre les méthodes et leur définitions. Finalement, nous avons fait le choix expliqué plus haut pour faciliter le traitement des méthodes “override” de méthodes utiles. Nous avons ainsi un “Set” de méthodes à explorer pour éviter de parcourir tout l’arbre des méthodes en boucle. Toutefois, nous ne profitons pas assez des relations entre les différentes classe de noeuds de l’arbre abstrait et il aurait été préférable de parcourir l’arbre récursivement plutôt que de boucler sur les classes et les méthodes à partir de la classe “Program”. Par soucis de temps, nous ne sommes pas revenu sur cette architecture.

3) Repérer les champs “override” de champs utiles

Pour ce qui est des champs, ils sont repérés lors du parcours du main et du corps des méthodes. Toutefois, de la même manière que pour les méthodes “override”, il est possible que des champs qui n’ont pas été repérés deviennent utiles dans un contexte dynamique. En deca, il n’est pas possible de “override” des champs mais si une sous classe possède un champ de même symbole qu’un champ de sa mère alors c’est bien ce champ qui sera utilisé dynamiquement si le type dynamique est celui de la sous-classe.

Contrairement au repérage des méthodes redéfinies où leur corps de méthode devait être exploré, il suffit pour les champs de parcourir 2 fois la liste de classes du programme sans boucler. La première passe permet de construire une table associant à chaque symbole de champs utilisé un “set” des classes qui l’utilise. Lors de la seconde passe, on repère l’ensemble des champs dont le symbole est présent dans la table et possède dans le set associé une super classe de du champ en question.

La construction du set aurait pu se faire lors des 2 premières étapes mais nous avons préféré distinguer clairement ces étapes pour une meilleure maintenabilité du code.

4) Elimination des variables inutiles

Une fois le repérage des variables utilisées effectué, on réalise une unique passe de l’ensemble de l’arbre pour l’élaguer et retirer les variables qui n’ont pas été repérées comme utiles. On retire ainsi de l’arbre toutes les classes, les champs et les méthodes qui ne sont pas utilisés ainsi que les déclarations de variable inutiles.

Cette étape se fait récursivement à l’aide des méthodes “removeUnusedVar()” qui renvoient “null” ou une instance de “Tree” pour faire remonter l’arbre modifié. Par ailleurs, ces méthodes

prennent en paramètre l'instance du programme afin de positionner à vrai son attribut booléen "varRemoved" quand une variable au moins est retirée. Cet attribut permet d'optimiser le programme tant que des optimisations sont réalisées.

En même temps qu'on retire les déclarations des variables, champs, méthodes et classes inutiles, on retire aussi du main et des blocs des méthodes les instructions inutiles. Pour ce faire, il nous est parfois nécessaire de savoir si une expression contient des expressions qu'il ne faut pas éliminer comme des "Read", des "MethodCall" (ils pourraient changer l'état d'un objet ou appeler d'autres méthodes) ou des assignations. On utilise donc la méthode "getUnremovableExpr()" qui permet d'obtenir récursivement et dans l'ordre d'apparition la liste de ces expressions à ne pas retirer grâce à la méthode "addUnremovableExpr()" qui ajoute une expression à la liste.

Plus concrètement, voici les simplifications importantes effectuées lors du parcours récursif :

DeclClass, DeclMethod, DeclField, DeclVar : Ces déclarations sont retirées si les variables ne sont pas utilisées. Toutefois, la déclaration d'une variable ou d'un champ n'est pas retirée si son initialisation contient des expressions à ne pas retirer ("Read", appel de méthode ou assignation). On pourrait être tenté de ne garder que ces expressions à ne pas retirer comme expliqué plus bas pour la simplification de "Expression" mais cela interdit d'avoir une instruction parmi les déclarations. Une amélioration possible serait de repérer à la génération du code l'inutilité de la variable déclarée et de ne réaliser les calculs que des expressions à ne pas retirer. Une autre amélioration possible serait de ne conserver que la somme des sous expressions à ne pas retirer (dans le cas d'une expression arithmétique et non booléenne) plutôt que l'entièreté de l'expression.

Assignation : On ne garde que la partie droite si la variable concernée est inutile, que l'assignation soit dans une expression ou non

Expression : Une expression seule comme instruction est supprimée et on n'ajoute à la suite et dans l'ordre les sous expressions qu'on ne peut pas retirer obtenues via "getUnremovableExpr()". Toutefois, dans le cas particulier d'une expression booléenne, on la conserve telle quelle car il est possible que seule une partie de l'expression soit évaluée à l'exécution.

NoOperation : On élimine les instructions sans effets codées par une suite de ";;".

IfThenElse : On simplifie la condition, le bloc "alors" et le bloc "sinon" et si ces derniers sont tous les deux vides alors on remplace l'instruction "IfThenElse" par sa condition.

While : Le traitement est similaire au "IfThenElse" à la différence qu'on ne remplace l'instruction par sa condition que si celle-ci ne contient pas de sous expression à ne pas retirer ("Read", appel de méthode ou assignation). Une contrepartie est que l'optimisation peut retirer une boucle

infinie si le bloc d'instructions est vide, que la condition est vraie et qu'elle ne contient pas de sous expression à ne pas retirer.

InstanceOf : L'expression est remplacée par faux si le type testé n'est jamais utilisé (pas de "New" avec cette classe ou une sous classe).

Main : Un "main" sans déclaration ni instruction est transformé en un "EmptyMain"

Une amélioration possible de cette simplification serait de repérer les méthodes qui n'induisent pas de "Read", de "Print" ou de changement d'état d'un objet (directement ou non). Cela permettrait de simplifier plus finement les expressions.

Finalement, il faut bien avoir en tête qu'une passe de simplification peut créer d'autres possibilités d'optimisation ce qui explique pourquoi on optimise en boucle jusqu'à ce que l'on ne puisse plus optimiser.

5) Remise à zéro de l'attribut utilisé

Etant donné qu'on optimise en boucle avec les différentes techniques d'optimisation, il est nécessaire de pouvoir remettre à zéro les attributs "used" des définitions des variables utilisées. Pour ce faire, on parcourt l'entièreté de l'arbre "Program" récursivement à l'aide des fonctions "iter()" données avec le squelette de code et on remet à zéro les définitions lorsque l'on atteint une feuille "Identifier".

6) Nota bene

Il est important de noter qu'en optimisant le programme ainsi, il est possible que des instructions qui auraient mené à une erreur à l'exécution soient retirées dans le code assembleur optimisé. L'utilisateur ne doit donc pas s'attendre à observer les erreurs qu'il attendait. C'est par exemple le cas pour une division par 0 dans un calcul inutile.

Par ailleurs, on peut noter que des méthodes peuvent être supprimées mais qu'on ne touche à aucun moment aux indices des méthodes. Le code assembleur généré va donc réserver de l'espace mémoire sur la pile pour la table des méthodes de chacune des classes utiles (et uniquement les classes utiles) mais ne va pas utiliser la totalité de cet espace réservé si des méthodes ont été éliminées. Ce choix, bien plus facile à implémenter, se justifie par le fait que cet espace inutilisé reste négligeable en comparaison à la taille totale de la pile. Néanmoins, on réduit significativement la taille du code assembleur en éliminant les classes inutiles et le corps des méthodes inutiles.

VI. Fonctions inline

L'idée de cette optimisation est d'optimiser les changements de contexte lors des appels de méthodes car c'est une opération coûteuse pour le compilateur. En effet, nous avons tout d'abord les instructions "BSR" et "RTS" qui prennent presque 10 cycles chacune pour aller lire le code de la méthode et revenir au programme principal. Ensuite, rentrer dans le corps d'une méthode demande de sauvegarder des registres, vérifier la taille maximale de la pile, et vérifier que la méthode renvoie bien quelque chose. A cela s'ajoute la vérification d'erreur lors de l'exécution. Tout cela peut être évité avec les méthodes "inline".

On qualifie de "inline" les méthodes qui n'ont aucune déclaration de variable et une seule instruction, un "Return". Ces méthodes calculent donc simplement une expression à partir de littéraux et de paramètres donnés en entrée et ne font pas intervenir de champs. Un exemple typique de méthode "inline" est une méthode qui prend en entrée un entier et retourne en sortie le carré de l'entier.

L'optimisation consiste donc à remplacer dans le code les appels à ces méthodes par leur corps de méthode, l'expression du "Return" en substituant les paramètres par les variables et littéraux donnée en entrée. L'expression du "Return" ne doit donc pas contenir d'assignation pour ne pas changer les valeurs des variables passées en entrée. Par ailleurs, il est possible qu'un paramètre apparaisse plusieurs fois dans l'expression. Il est donc indispensable qu'il ne soit passé en paramètre d'entrée que des littéraux ou des variables (possiblement "castée") mais pas d'expression plus complexes pour ne démultiplier les calculs. On vérifie cette condition en appelant la fonction "isAtomic()" sur chacun des paramètres d'entrée.

Plus concrètement, cette optimisation est réalisée en 2 passes: une première pour repérer les méthodes "inline" et une seconde pour les remplacer quand c'est possible.

1) Repérage des méthodes "inline"

On repère les méthodes "inline" en parcourant récursivement l'ensemble des méthodes depuis le nœud "Program" avec les méthodes "spotInlineMethod()". On lui passe en paramètre une table associant à chaque définition de méthode "inline" sa déclaration pour pouvoir récupérer facilement le corps des méthodes "inline" lors de la substitution. Pour chaque méthode, on vérifie si elle respecte les caractéristiques des méthodes "inline" et on l'ajoute à la table si c'est le cas. Cette vérification se fait rapidement en parcourant récursivement l'expression avec "containsField()" et "getUnremovableExpression()" pour s'assurer que l'expression ne contient pas de champ ou d'assignation.

2) Repérage des appels aux méthodes “inline”

Une fois les méthodes “inline” repérées, on parcourt récursivement depuis “Program” le corps du “Main” et le corps des méthodes avec la méthode “doSubstituteInlineMethods()” à la recherche d’appels de méthodes. A chaque appel de méthode rencontré, on regarde si la méthode est dans la table des méthodes “inline” passée en paramètre. Si c’est le cas alors on regarde si les paramètres passés en entrée sont bien “atomiques” à l’aide des méthodes “isAtomic()” puis on réalise la substitution. La méthode “doSubstituteInlineMethods()” retourne un arbre abstrait “Tree” pour remplacer effectivement l’appel de méthode par un nouvel arbre de substitution.

3) Remplacement des appels aux méthodes “inline”

Une fois qu’un appel de méthode “inline” remplaçable par son expression a été trouvé, on récupère l’expression à substituer à l’aide de la table des méthodes “inline” et on substitue les paramètres à l’aide de la méthode “getSubstitution()”. Cette méthode crée une table de substitution en associant à un paramètre de la méthode “inline” le paramètre donné en entrée. Elle parcourt ensuite récursivement l’expression à l’aide de “substitute()” en construisant à chaque nœud un nouvel arbre de même classe, de même type et de même localisation qu’elle fait ensuite remonter. A chaque fois que “substitute()” rencontre un paramètre de la méthode “inline”, elle fait remonter le paramètre d’entrée associé grâce à la table de substitution passée en paramètre.

4) Nota bene

Il aurait été intéressant dans notre implémentation de passer en plus en paramètre l’instance de du “Program” pour lui indiquer quand un appel de méthode a été substitué. Cela permettrait de boucler plus finement sur les différentes techniques d’optimisation.

Enfin, il est important de noter que nous ne vérifions pas lors de la substitution que l’objet appelant est bien instancié. Un tel appel lance normalement une erreur à la compilation mais il est possible que l’appel soit substitué par l’expression de la méthode “inline” lors de l’optimisation et qu’aucune erreur ne soit lancée à l’exécution. Ceci est un comportement que l’utilisateur doit avoir en tête en utilisant l’option d’optimisation. Nous n’avons pas cherché à pallier ce problème car il introduit une très grande complexité et que ce cas particulier ne limite que très peu l’utilisation du compilateur. En effet, une méthode “inline” n’a, par définition, aucun champ qui apparaît dans son expression. L’état de l’objet appelant n’a donc en fait aucune importance.

VII. Factorisation des calculs

L'idée de cette optimisation est principalement basée sur l'optimisation du nombre de cycles. On part de la connaissance que l'instruction "MUL" prend 20 cycles à être effectuée, alors que les instructions de décalage "SHL" et "SHR" ainsi que l'addition "ADD" prennent 2 cycles. Cette optimisation, bien qu'appelée de factorisation, peut être décomposée en deux étapes que sont l'étape de factorisation et celle de développement.

1) Étape de factorisation

L'étape de factorisation consiste comme son nom l'indique à factoriser les expressions de calculs. Ainsi, pour chaque expression de chaque instruction, une fonction est chargée de calculer le nombre d'occurrences d'un terme. Ici, nous ne comptons que le nombre d'occurrences des *Identifieur* et non des expressions plus complètes. Il nous est donc impossible de factoriser une expression par exemple par $(a+b)$. Il serait donc favorable par la suite de pouvoir compter le nombre d'occurrences des expressions pour pouvoir factoriser de manière plus générale le code.

Une fois le nombre d'occurrences établi, un noeud de multiplication est alors créé entre l'*Identifieur* et son nombre d'occurrence. Ceci ne devrait être fait qu'à partir d'un certain nombre, mais actuellement, si une variable n'apparaît qu'une seule fois, elle sera tout de même multipliée par 1.

2. Étape de développement

Cette étape se base sur la technique de multiplication égyptienne. Il s'agit de décomposer une multiplication en somme de produit du terme et d'une puissance de deux en l'occurrence.

Avant d'être développée, la multiplication est envoyée dans la fonction *isSplitable* permettant de savoir si la décomposition de la multiplication est avantageuse ou non en terme de nombre de cycles à l'exécution. Une multiplication prenant 20 cycles, il s'agit de décomposer cette même multiplication en un maximum de 9 opérations d'additions et de décalages. Ainsi une variable multipliée par 63 ne sera pas décomposée qui une fois décomposée donne 5 additions et 15 décalages, tandis que 15 le sera avec ses 3 additions et 6 décalages.

Si l'expression est décomposable, on la décompose alors effectivement en somme de multiplication avec des puissances de 2 et on met à jour pour chacune de ces multiplications une variable propre au noeud en appelant la fonction *setShiftReplacable* qui spécifie que l'expression en question doit utiliser un décalage et non pas une multiplication lors de la génération du code assembleur.

3. État actuelle de l'optimisation

À ce jour, cette optimisation est celle ayant été la moins bien implémentée en partie dû au peu de temps que nous lui avons accordé. Beaucoup de problèmes ont été relevés, certains faciles à résoudre, d'autres devant reprendre l'intégralité du code de factorisation. Parmi les problèmes relevés on peut noter les suivantes:

- Utilisation inutiles de la fonction *isLiteral*, qui de plus n'a pas été supprimée dans le fichier *AbstractExpr.java* générant généralement un problème.
- Si un *Identifieur* est un nombre flottant, il sera lors de la factorisation multiplié par un entier sans créer de noeud *ConvFloat*. Cette étape d'optimisation ne peut donc pas être considérée avec des flottants.
- Comme expliqué plus haut, une variable n'apparaissant qu'une seule fois sera à la fin multipliée par 1. Même si cette multiplication est censée être supprimée par l'étape d'optimisation de calculs évidents, ce n'est pas un comportement souhaitable pour une fonctionnalité d'optimisation.
- Il subsiste également un problème lorsque l'on essaie de factoriser des expressions comportant des additions et des soustractions. Cette erreur n'apparaît pas dans chaque expression mêlant ces deux opérations, mais cela reste quand même une bonne limitation de l'optimisation.

VIII. Élimination des calculs redondants

Il a été convenu avec le client de ne pas implémenter cette partie pour le 23 janvier mais nos équipes travaillent activement sur le sujet pour la prochaine version du compilateur. En voici une description du travail effectué.

1) Description de l'optimisation

L'objectif de l'élimination des calculs redondants est d'éliminer les accès pas nécessaires aux calculs réalisés au préalable ainsi que d'éviter les calculs qui ont déjà été faits avant. Ainsi, si par exemple dans une somme nous avons des variables qui ont déjà été définies au préalable, ces variables seront changées par l'expression correspondante à la variable définie au préalable. L'objectif est de réduire le nombre d'accès mémoire. Ainsi, une élimination réitérée de tous les calculs redondants implique que les opérations en dehors des boucles et des ifs seront progressivement changées par des opérations de littéraux, ce qui servirait aux autres extensions pour pré-calculer le résultat de l'opération durant la compilation.

2) Architecture utilisée

Pour pouvoir accéder aux valeurs rapidement dans le compilateur, nous avons implémenté un dictionnaire *currentValues* qu'à chaque définition et redéfinition d'une variable du code, on associe une expression associée. Ainsi, dès qu'une variable est appelée, nous pouvons substituer la variable par l'expression définie.


Pour appliquer la substitution, nous avons implémenté la fonction *irrelevant* qui a été redéfinie dans la majorité des classes de l'arbre. La fonction renvoie un booléen indiquant si les fils de la classe sont des variables définies au préalable. Si oui, la fonction réalisera les changements nécessaires afin de changer les variables par leurs expressions de définition. La fonction *irrelevant* renvoie *true* directement dans une classe Identifier, pour indiquer que nous avons bien une variable à changer. De fait, le compilateur parcourt l'arbre en recherche des variables à changer, à chaque passage il effectue les modifications nécessaires et remonte à la racine l'information de s'il reste des transformations à effectuer. Si des transformations peuvent encore être faites, le compilateur re-parcourt l'arbre en recherche de nouvelles variables à changer.

Pour ce qui concerne les classes, le compilateur réalise les modifications aussi à l'intérieur de chaque méthode de manière indépendante afin de pouvoir éliminer les calculs redondants aussi à l'intérieur des classes. Puis, pour chaque classe, nous avons un tas de variables définies (attributs) avec ses expressions associées. Nous avons donc définie un dictionnaire de dictionnaires qui, à chaque classe, associe le dictionnaire des variables définies.

Ainsi, lorsque dans une méthode ou dans un *main*, nous appelons à une classe (à travers un *New*), le compilateur prend en compte la nouvelle classe en associant à la variable contenant la classe les variables définies comme attributs dans la classe. De même, dans le cas d'une variable définie dans une classe extérieure qui porterait le même nom que dans une méthode ou le *main*, le compilateur peut distinguer les deux variables avec le préfixe *this* : si le *this* est présent, il faut chercher dans les *currentValues*, et sinon dans la définition des classes externes.

3) Traitement des boucles

Un des problèmes qui est traité par les chercheurs en matière d'optimisation est justement l'optimisation des boucles *while*. Une boucle *while* est compliquée à optimiser car nous ignorons au préalable si la boucle finira ou pas. Ceci provoque qu'il est difficile d'appliquer l'optimisation dans un *while*, car si une variable est réécrite à l'intérieur de la boucle, sa valeur changera jusqu'à la fin de la boucle et nous ne pouvons pas alors prédire la valeur finale de celle-là. En tout, une boucle *while* symbolise beaucoup de problèmes dans l'optimisation et rend fausse l'élimination des calculs redondants. Ainsi, nous avons implémenté que, si une variable est redéfinie à l'intérieur d'une boucle *while*, celle-ci serait éliminée de *currentValues* afin de ne pas



être prise en compte dans le reste du code et ainsi éviter de changer les valeurs de la variable si elle est appelée après. Si cette limitation n'est pas réalisée, cela peut impliquer une fausse modification des valeurs de la variable.

4) Traitement des ifs

Dans le cas des ifs, de la même manière que pour les boucles, nous ne pouvons pas savoir forcément si nous allons rentrer ou pas dans le if ou si le programme partira dans le else. Ainsi, ça serait faux de changer les variables qui ont été redéfinies à l'intérieur d'un if, car la valeur est inconnue (on ignore si elle a été redéfinie ou pas, comme on ignore si on rentre ou pas dans le if).

Nous avons alors implémenté que, à l'intérieur d'un if/else, une copie des *currentValues* est créée afin de pouvoir continuer l'optimisation au sein de chaque partie if ou else de manière indépendante l'une de l'autre. Néanmoins, toutes les redéfinitions effectuées entraînent une élimination de la variable de *currentValues* comme avec les boucles. Ceci permet de ne pas changer les variables dans le futur si on ne connaît pas certainement sa valeur.

5) État actuel de l'optimisation

Actuellement, l'optimisation est fonctionnelle et réalise bien les échanges désirés. Nous avons réalisés plusieurs tests unitaires s'arrêtant à la décompilation qui prouvent que l'arbre décoré après passage par cette optimisation est bien l'arbre réduit que nous souhaitons avoir. Néanmoins, il y a deux problèmes qui ont été la source de ne pas avoir voulu les inclure dans le rendu final du 23 janvier :

Premièrement, la non-possibilité de lancer la programmation parallèle. La structure interne des dictionnaires se réalise à travers des *static*, ce qui entraîne un partage global des dictionnaires dans l'entièreté du code. Ainsi, si on lance plusieurs programmes en parallèles, le dictionnaire sera partagé par les différentes partitions parallèles et mènera alors à des informations qui sont erronées (par exemple : si on a un test 1 et un test 2 qui définissent chacune une variable a de valeurs entières différentes, si le test 1 définit a mais que le test 2 définit a après, le test 1 prendra en compte la valeur de a du dictionnaire, soit la valeur du test 2. Ceci est faux). Ainsi, l'optimisation ne fonctionne pas avec l'option parallèle.

De plus, l'optimisation utilise une grande quantité de ressources (dictionnaires, dictionnaires, copie de dictionnaires de dictionnaires...). Ceci étant très lourde comme architecture, nous avons préféré l'enlever afin de l'améliorer en interne avant la sortie définitive de l'optimisation.

IX. Optimisation Post Génération

Nous avons plusieurs techniques d'optimisation après du code, mais la plupart ont été enlevées à la fin puisqu'elles nécessitaient plus de travail pour les ajouter à la compilation objet, et que nous avons manqué de temps pour les implémenter proprement. Cependant, nous allons quand même parler de ces optimisations, de leur fonctionnement et implémentation.

Pour organiser ces différentes optimisations de façon propre, on a d'abord créé une classe "AssemblyOptimizer" qui ne contient que des fonctions statiques qui optimisent un programme IMA. Chaque optimisation est effectuée par une fonction. De la même façon que sur l'arbre, ces optimisation s'exécutent à la chaîne, et donc on continuera d'optimiser le programme tant qu'on effectue des changements. Les différentes fonctions d'optimisation vont donc renvoyer un booléen, qui indique si le programme a été modifié, et tant qu'au moins une de ces fonctions optimisera le programme, on continuera de les appeler.

1) Remplacement par immédiats

En parcourant le programme IMA, on va construire une table des différents registres. A chaque fois que l'on va charger un immédiat dans un registre, on va alors connaître la valeur de ce registre, et remplacer chaque occurrence de ce registre par l'immédiat correspondant. Si on charge quelque chose d'autre dans ce registre, que l'on ne connaît pas encore, alors on enlève de la table la valeur courante du registre. Cela permet de libérer l'utilisation des registres, et d'éviter des chargements inutiles.

2) Suppressions des écritures inutiles

Cette fois, on va parcourir le programme IMA à l'envers. On garde en mémoire tous les registres qui ont été lus ou utilisés, dans une table de booléens, donc on initialise cette table à faux pour chaque registre, car aucun ne va être utilisé lorsqu'on arrive à la fin du programme. Ensuite, en parcourant le programme à l'envers, chaque registre utilisé va être marqué comme tel, puis lorsqu'on écrit dans ces registres, on les note comme non utilisés. Ainsi, lorsqu'on écrit dans un registre non utilisé, on peut supprimer l'instruction, puisque ce registre ne va pas être lu avant d'être réécrit. L'instruction était donc inutile. On peut trouver ce genre d'instructions inutiles après le remplacement des immédiats, car l'utilisation de certains registre ont été changés en immédiats.

3) Optimisation des conditions

Cette optimisation a été prévue, mais n'a pas pu être implémentée dû à sa complexité par rapport au temps fourni. L'idée était que dans le code assembleur généré, on se retrouvait

souvent avec plusieurs instructions conditionnelles d'affilée (Scc et Bcc). En voyant cela, on a remarqué qu'il était possible de combiner ces tests conditionnels, en changeant les conditions.

4) Utilisation des registres

Après avoir généré et optimisé le code assembleur une première fois, on peut avoir des registres encore disponibles, selon le nombre de registres utilisable. Sachant que la lecture et l'écriture sur un registre ne coûte rien (0 cycles), on peut les utiliser à notre avantage. On construit donc la liste des registres disponibles et jamais utilisés lors du programme, et on regarde les variables ou immédiats les plus utilisés. On peut alors allouer entièrement des registres pour ces variables. Si par exemple, dans le programme on utilise souvent l'immédiat 0, on le met dans un registre, et on économise tous les chargements de l'immédiat 0. De même, les variables principales les plus utilisées peuvent être stockées dans des registres au lieu de la pile. Cela permet de s'épargner tous les chargement et enregistrement sur cette variable.

5) Remplacement par instructions moins chère

Certaines instructions prennent beaucoup de cycles et donc de temps d'exécution, et peuvent parfois être remplacées par des instructions plus rapides. L'exemple le plus parlant est la multiplication ou la division entière, qui peuvent être remplacé par des décalages lorsque un des opérandes est un immédiat qui est une puissance de deux. De plus, on peut également remplacer les chargement de l'immédiat 0 par une soustraction d'un registre à lui-même, mais seulement s' il contenait déjà une valeur entière ou flottante (pas une adresse). Concrètement, "SUB R2 R2" ne prend que deux cycles, alors que "LOAD #0 R2" en prend quatre.

X. Ressources

[Les optimisations des compilateurs • Tutoriels • Zeste de Savoir](#)

[Les optimisations des compilateurs](#)

[Optimisation des compilateurs](#)

[Optimisation des compilateurs/Les optimisations des expressions et calculs — Wikilivres](#)

[Optimizing compiler - Wikipedia](#)