Rapport de Projet - UE Compilation

10/01/2022

Compilateur pour le langage *DECAF*

HAMMER Tom JANATI Siham WEREY Laurent XU Chenglin

Sommaire

I. Options d'exécution du programme

Sommaire

./decaf <source> -o <dest> [-tos] [-i] [-W] [-Caffeine] [-Caffeine+]
./decaf -version

<source>	Fichier source à compiler
-o <dest>	Spécifie le nom du fichier produit en assembleur MIPS *(stdout pour sortie standard)*
-tos	Affiche la table des symboles
-i	Affiche le code intermédiaire
-version	Affiche les membres du groupe
-W	Affiche des warnings sans erreur
-Caffeine	Optimisation du code intermédiaire (à consommer avec modération)
-Caffeine+	Optimisation avec affichage des Blocs de Bases

***-W** est encore très limité. Pour le moment, cette option vérifie uniquement que toutes les fonctions de type non void effectuent bien un retour. Les options Caffeine ne sont qu'un début modeste d'optimisation détaillé plus bas.*

II. Capacités du compilateur

Sommaire

Toutes les fonctionnalités demandées dans le sujet ont été implémentées. Un début d'optimisation modeste a aussi été réalisé et sera détaillé plus bas, ainsi que deux fonctions de librairie additionnelles.

III. Choix d'implémentations

Sommaire

1. Table des symboles et calcul d'offsets

Nous avons implémenté une table de hachage vu que c'est la méthode la moins coûteuse en temps d'accès, les collisions sont traitées avec des listes chaînées et la fonction de hash a été choisie pour limiter les collisions (K&R v2).

Dès le début du projet, nous nous sommes basés sur un modèle de conservation des offsets dans les quadop et de suppression définitive des TOS lors du dépilement du contexte. Nous avons eu des problèmes peu de temps avant le rendu dans certaines situations particulières

où les offsets n'étaient pas correctement calculés lors de l'exécution, principalement à cause du fait que la déclaration des temporaires se faisaient au fur et à mesure, il n'y avait pas de réservation d'espace complet en entrant dans un contexte nouveau. C'est une erreur qui a été corrigée beaucoup plus facilement en changeant de modèle et en gardant un pointeur vers la table des symboles dans les quad de début de contexte, vu que la taille du contexte est conservée dans celle-ci.

Dans la fonction de traduction MIPS, nous avons un pointeur sur le contexte courant qui est mis à jour lors du quad d'empilement de contexte, et est modifié par l'élément suivant à chaque quad de dépilement de contexte, vu que le chaînage est conservé. Le calcul des offsets se fait désormais lors de la traduction MIPS et les quadop ne conservent plus que les noms des variables et leurs types.

2. Booléens

Sommaire

Les booléens doivent pouvoir être représentés de deux manières différentes selon le besoin, soit un couple de True/False Lists pour les structures de contrôles, ou bien une valeur booléenne. Pour celà, nous avons créé deux fonctions ***'goto_to_val'*** et ***'val_to_goto'*** qui permettent de passer d'une représentation sous forme de couple true/false lists en valeur stockée dans une temporaire, et inversement. La première est utilisée dans les tests d'égalités, les appels de méthodes et les affectations. La deuxième pour l'opérateur de négation '***!***', les opérateurs '||' et '***&&***', et au sein des clauses ***'If'***.

3. Break et Continue

Sommaire

Comme nous effectuons des dépilements à la sortie de chaque bloc, il faut prendre cela en compte pour les ***Break*** et les ***Continue*** car ces derniers peuvent sauter le quad de dépilement de contexte qui a été créé pour ***Next***.

S'ils existent, les listes ***Break*** et ***Continue*** sont donc complétées à la sortie du bloc par un quad de dépilement de contexte (uniquement si des variables ont été déclarées dans ce bloc), suivi d'une création d'une nouvelle liste avec un ***Goto*** incomplet. Pour le Return, ceci n'est pas nécessaire car on utilise **$fp** en fin de méthode.
De plus, le quad de dépilement du bloc d'une méthode est **remplacé** par un quad de fin de méthode vu qu'à ce stade, on utilise **$fp** pour dépiler.

4. Vérifications dynamiques

Sommaire

a. Return

Lors de l'appel d'une méthode, le quad créé ***Q_METHODCALL*** enregistre dans un ***quadop*** si l'appel attend une valeur de retour, c'est simplement le type de la méthode qu'on récupère depuis la table de symboles. La règle ***Statement : Return return_val*** crée aussi un quad de return indiquant le type de retour.

En MIPS, c'est le registre ***$v1*** qui sera utilisé pour indiquer si un retour a été effectué. La traduction du quad d'appel de méthode commence par initialiser le registre ***$v1*** à 0, alors que la traduction du ***quad*** de ***Return*** le met à 1. Si lors de l'exécution, le contrôle passe par les instructions du ***quad Return***, le registre ***$v1*** sera mis à 1. Après le ***jal,*** l'appelant vérifie la valeur du registre et quitte le programme si nécessaire.

b. Accès Tableau

Lors de la déclaration d'un tableau, sa taille est aussi stockée dans un label portant le nom du tableau suivi de **_SIZE**. Il ne peut y avoir de conflits grâce aux namespace expliqués plus bas. Lorsqu'on rencontre un quad d'accès ou d'écriture tableau, on vérifie que l'index est bien compris entre 0 et **_SIZE** et on quitte le programme si nécessaire.

5. Règles intermédiaires

Sommaire

a. Arguments Booléens

Pour pouvoir passer à une méthode des arguments booléens qui sont représentés par un couple de Goto, il faut d'abord faire une réification pour les transformer en valeur. Pour cela, il est nécessaire de créer des règles intermédiaires avant d'évaluer les arguments suivants.

b. Appels Récursifs et paramètres de méthodes

Pour permettre des appels récursifs et pouvoir reconnaître les paramètres d'une méthode, il faut que ces derniers existent dans la table des symboles avant d'entrer dans le bloc, une règle intermédiaire est donc nécessaire.

c. Copie de la valeur maximale du compteur de la boucle For

Un nouveau token ***'Max'*** de type ***intval*** a été créé pour récupérer cette valeur. Ce token est ***%empty*** et ne produit aucune règle. Il ne sert que comme 'variable' vu que l'on ne peut pas récupérer des variables créées au sein d'une règle intermédiaire.

d. Evaluation de l'expression au sein d'un 'If'

Si l'expression au sein d'un ***'if'*** est représentée par une valeur, il faut la transformer en un couple true/false list avant d'entrer dans le bloc. Ceci nécessite la création d'une règle intermédiaire et une factorisation de la grammaire pour éviter les conflits entre 'If' et 'If Else'.

6. Vérification des types de retour

Sommaire

La règle ***Statement : Return return_val;*** crée un quad ***Q_RETURN*** avec un quadop qui stocke le type de retour, une liste est créée pour l'attribut return de ***Statement*** qui pointe vers ce quad. Les attributs return sont concaténés dans tous les blocs et complétés dans la règle de déclaration de la méthode par le quad de fin de méthode. C'est au sein de cette dernière règle que le type de retour est vérifié en bouclant sur toutes les adresses contenues dans la liste *Return* pour accéder au quad correspondant et vérifier que le type des quadop correspond bien au type de la méthode.

7. Tableaux

Sommaire

Pour les accès tableau, nous avons un ID qui est une variable globale, et une expression à évaluer représentant l'index auquel on veut accéder. Cette expression peut aussi être un autre ID et sera donc dans ce cas transformée en temporaire. On génère ensuite un quad ***Q_ACESSTAB*** où les opérandes sont l'ID du tableau, une temporaire/constante pour l'index et une création d'une nouvelle temporaire pour stocker la valeur de la case du tableau.

Pour l'écriture tableau, il n'y a pas une grande différence, l'ordre des opérandes est inversée, et le même quad d'affectation pour les scalaires est utilisé pour les tableaux. La distinction se fait lors de la traduction MIPS.

8. Namespace

Sommaire

Les mots suivant sont des mots réservés du langage qui ne peuvent être utilisé qu'au sein d'un contexte local:

__glob_ Spim ne permet pas de définir des variables globales dont le nom est un opcode en MIPS ('b' par exemple). Le namespace **__glob_** est donc ajouté systématiquement, lors de la traduction en MIPS, aux débuts des noms de toutes les variables globales.

__str_ sert à définir les labels des chaînes de caractères passées comme arguments à WriteString. Il est toujours ajouté au début du label d'une chaîne de caractère.

__end__ sert à définir le label de fin d'une fonction. Il est suivi du nom de la fonction.

9. Fonctions de librairie

Sommaire

a. Quelques précisions concernant Read_Int

Nous avons ajouté un passage par adresse pour les appels de méthodes. Il faut ajouter un '***&***' avant l'identificateur comme pour la fonction ***scanf***.
Une autre solution que nous avons trouvée moins élégante aurait été d'avoir deux règles de grammaire séparées pour deux types de fonctions, un type qui ne prend que des ***location*** en argument et un autre type qui ne prend que des ***expr***. Nous avons préféré une solution plus ouverte qui nous permettrait d'améliorer notre compilateur par la suite pour prendre en compte les pointeurs et à la fois les passages par adresses et par copie dans toutes les fonctions.

La structure ***param*** permet de récupérer les paramètres définies d'une fonction, et est aussi utilisée pour récupérer les arguments d'un appel. Celle-ci contient un attribut ***by_Address*** qui permet de faire cette distinction lors de la vérification des types.

Le passage par adresse ne fonctionne que pour Read_Int, vu que nous n'avons pas eu assez de temps pour ajouter les pointeurs au langage et permettre leurs définitions dans les paramètres.

b. Fonctions supplémentaires

1. **Int Random (int *upper_bound*)** Retourne une valeur entière aléatoire entre 0 inclu et upper_bound exclu.
   Cette fonction ne peut être exécutée que sur le simulateur Mars vu que Spim a un ensemble de syscall très limité.

2. **Void Exit (int *val*)** C'est la fonction standard pour quitter un programme avec une valeur de retour.

IV. Débuts d'optimisation

Sommaire

1. Débuts CFG

Nous avons commencé par implémenter une structure de graphe dont les éléments sont les blocs de bases du programme. Vu la contrainte de temps, nous n'avons pas pu l'utiliser pour optimiser le code intermédiaire. Le programme se limite à la définition des blocs de bases qu'il est possible d'afficher grâce à l'option ***-Caffeine+***.
La seule 'optimisation' qui agit sur le code intermédiaire est une suppression de certains Goto inutiles. Il arrive que certains Goto soient complétés par l'instruction qui suit directement le jump et que ceux-ci ne soient pas un label, il est donc possible de les supprimer. Les options **-Caffeine** et ***-i*** ou l'option ***-Caffeine+*** seule permettent d'afficher les lignes supprimés dans le tableau. (Si des gotos inutiles existent).

Certains fichiers de tests dans le dossier UseCase, notamment 'BasicBlocks.decaf' ont été réalisés de manière à avoir plus d'instructions élémentaires et limiter le nombre de jump, pour mieux voir le résultat de la définition des blocs de base.

2. AST abandonné

Afin de réduire le nombre de variables temporaires utilisées, nous avions pour projet d'intégrer une résolution des expressions mathématiques à l'aide d'un AST. Les autres parties de la grammaire auraient gardé leur fonctionnement en générant immédiatement le code intermédiaire. Mais, dès l'encontre d'une expression, un AST serait construit pour l'expression puis le code correspondant généré.

Une telle implémentation aurait permis la réduction du nombre d'Ershov (i.e le nombre de temporaires) en effectuant un parcours en profondeur de l'arbre. Le fonctionnement de base avec uniquement les règles grammaticales s'apparentant plus à un parcours en largeur, donc coûteux en mémoire.

En raison de la complexité de cette implémentation, nous avons décidé de la reléguer à la partie "optimisations" de notre planning prévisionnel. Nous n'avons pas eu le temps de terminer la fonctionnalité mais il est possible de retrouver le squelette des structures de données utilisées dans les fichiers *ast.c* et *ast.h* du répertoire *old*.

V. Jeu de Tests

Sommaire

Le jeu de test fourni présente plusieurs catégories de tests, permettant de tester séparément les fonctionnalités du compilateur, de tester son utilisation dans des use-cases réels, et de tester que des entrées erronées amènent bien à une erreur de compilation.

Les premiers, fournis dans le dossier «*Should_succeed*», constituent ce qui pourrait s'apparenter le plus à des tests unitaires. Les prédicats de tests sont les suivants:

– Un test pourra uniquement utiliser des fonctionnalités déjà vérifiées dans un test précédent. Ce qui veut dire que l'ordre de ceux-ci est important.
– Un test vérifiera une seule et unique fonctionnalité supplémentaire qui pourra, à son tour, être utilisée de manière sûre dans les tests suivants.

Afin de faciliter l'implémentation de ces tests, les premières fonctionnalités qui seront testées seront les fonctions d'entrée/sortie du compilateur. Celles-ci ont été implémentées à la main et ne dépendent donc pas d'autres fonctionnalités du compilateur. Une fois ces fonctions testées, nous les utiliserons pour vérifier les valeurs de différentes variables dans le cadre de nos tests en les exécutant à l'aide de SPIM.
L'ordre de ces tests correspond à l'ordre d'implémentation des diverses fonctionnalités du compilateur et est le suivant:

1. Fonctions I/O
2. Déclarations et affectations de variables
3. Opérations sur les variables
4. Structures de contrôle
5. Fonctions et méthodes
6. Contextes et masquage

Nous pouvons ainsi garantir l'intégrité de toutes les fonctionnalités implémentées une par une et déterminer l'origine de tout problème dans le code de sortie aisément.

Les seconds, fournis dans le dossier « *Should_fail* », permettent de vérifier le bon renvoi d'erreur de compilation dans les cas énumérés par le sujet. Ils sont répartis dans les mêmes catégories que les tests précédents.

Les troisièmes, fournis dans le dossier *Runtime_error*, sont présents afin de tester le bon fonctionnement des renvois d'erreurs dynamiques (i.e à l'exécution). Il permettent de vérifier que le code assembleur renvoie bien un message d'erreur dans les cas prévus par le sujet.

Les derniers, fournis dans le dossier *Use_case*, constituent une collection d'exemples utiles pour visualiser l'exécution d'un programme type en Decaf. Il est à noter que l'exemple *PauseCafe.decaf* fonctionnera uniquement sur Mars, étant donné son utilisation de la fonction *Random*, décrite plus haut.

Exécution des tests

Sommaire

Chacun des dossiers de catégories de tests présente un script *test.sh* qu'il est possible d'exécuter afin de lancer les tests de la catégorie en affichant une vue détaillée ou non de la réussite de chaque test.

Un script *test.sh* se trouve également à la racine du répertoire de tests et peut être lancé afin d'exécuter l'intégralité des tests et d'afficher une vue globale, par catégorie, de ceux-ci. Le script peut être lancé avec l'option *-v* pour afficher la sortie détaillée de tous les tests.