Machine Virtuelle pour Réprésentation Intermédiaire

Wilfried de Denterghem & Hugues Delorme

1. Introduction

Dans le cadre du projet Tiger visant à implémenter un compilateur pour le langage Tiger, on passe par plusieurs phases menant du source au binaire exécutable. L'une de ces phases consiste à transformer l'arbre de syntaxe abstraite en arbre de représentation intermédiaire (IR) indépendant de toute architecture matérielle. Les étapes suivantes constituent le back-end du compilateur qui à partir de cette IR va générer, au travers de nombreuses sous-étapes, le code objet de l'exécutable cible. Pour aider à la conception et la réalisation du compilateur, il serait intéressant de pouvoir disposer d'un moyen d'exécuter le programme source dés la phase de représentation intermédiaire afin de valider les étapes précédentes et aussi de pouvoir rapidement tester un programme Tiger.

2. Sujet

Le but du projet est de réaliser une machine virtuelle pouvant à partir de la représentation textuelle d'une IR exécuter le programme Tiger correspondant.

3. Concepts généraux

3.1. Description de l'IR

Voir :
- Modern compiler implemantation in *, Andrew W. Appel : chapitre "Translation to Intermediate Code"
- http://www.lrde.epita.fr/people/akim/compil/compil.html#Tiger%20Intermediate%20Representation

3.2 Canonisation

*3.*2.1 Buts

Après la traduction de l'arbre syntaxique en code intermédiaire, on cherche à rendre ce code exempt de certaines expressions gênantes pour les phases qui suivent.

Par exemple, il est parfois nécessaire de pouvoir évaluer les sous expressions d'une expression dans n'importe quel ordre. Les noeuds de type ESEQ ou CALL rendent ceci impossible.

De plus les noeuds de type SEQ n'apportent aucune information particulière, et il peuvent être arrangés de telle façon qu'ils peuvent être supprimés.

Cette phase de canonisation se déroule en 3 étapes :

- L'arbre de code intermédiaire est ré-écrit en une liste d'arbres canoniques, dont la définition est donnée ci-dessous.
- Cette liste est réarrangée en blocs fondamentaux (expliqués plus loin).
- Ces blocs sont ordonnés en traces(cf ...).

3.2.2 Les arbres canoniques

Les arbres canoniques ont les propriétés suivantes :

- Aucun noeud de type ESEQ ou SEQ
- L'expression parente de chaque noeud de type CALL est soit de type SXP ou MOVE(TEMP t, ...)

Ces arbres sont produits grâce à des règles de ré-écriture.

3.2.3 Les branchements conditionnels

Afin de rendre la traduction en instructions machine plus facile, il faut arranger les CJUMP.
Chaque CJUMP(condition, exp1, exp2, true_label, false_label) doit ainsi être suivi par LABEL(false_label).

Cette transformation s'effectue en 2 étapes : transformation des arbres canoniques en blocs fondamentaux, enfin transformation de ces blocs en traces.

3.2.4 Blocs fondamentaux

Un bloc fondamental est une séquence d'expressions qui se décomposent comme ceci :

- Un LABEL
- Des expressions...
- La dernière expression est un JUMP ou un CJUMP
- Il n'y a pas d'autres LABEL, JUMP ou CJUMP entre le début et la fin du bloc

3.2.5 Traces

Une trace est une séquence d'expressions qui peuvent être exécutées consécutivement(lors de l'exécution du programme). Dans le but d'arranger les CJUMP et ses "label faux", il faut produire un ensemble de traces qui couvrent le programme de façon exacte : chaque bloc doit se trouver dans une et une seule trace.

4 Spécifications de la machine virtuelle

4.1 Définition du format de fichier d'entrée

La machine virtuelle prendra en entrée un fichier texte généré par tc qui contiendra la représentation de tree qui pourra être de deux natures différentes : soit une IR "de base" telle que donnée par l'option -display-ir de tc ou bien encore une IR canonisée (voir 3.2) telle que donnée par l'option -display-canon de tc. La machine virtuelle devra détecter les éventuelles incohérences dans le fichier fourni en entrée en fonction du type d'IR choisi par l'utilisateur, par exemple la présence de SEQ alors qu'on utilise une IR canonisée. Afin de normaliser l'entrée, on conviendra que l'indentation éventuelle de sous-niveaux sera donnée par un tab (\t) à chaque fois.

Il est fortement conseillé d'implémenter la phase de lecture du fichier à l'aide d'outils spécialisés tels que Flex/Bison ou ANTLR.

4.2 Gestion de la mémoire

La gestion de la mémoire est un point très important de la machine virtuelle.
Ici sont décrites les principales fonctionnalités attendues.

- Les tableaux et les structures Tiger sont des pointeurs sur des données allouées
dans le tas. Pour accéder à un élément, le calcul de l'offset est basé sur la taille naturelle d'un mot de l'architecture cible. Il faudra définir ce genre d'informations nécessaires à la machine virtuelle.

- On a vu précédemment que l'opérateur MEM(e) permet de faire référence au contenu de la mémoire débutant à l'adresse e. A l'exécution, la machine virtuelle doit être capable de protéger l'accès et l'écriture d'une case mémoire, et de signaler toute opération eronnée sans que cela mette fin à l'éxecution de la machine virtuelle elle-même.

- Pour la gestion des frames, la machine virtuelle dispose de 3 registres spéciaux :
    - Le registre FP(Frame Pointer)
    - Le registre SP(Stack Pointer)
    - Le registre RV(Return Value), dédié au stockage du résultat d'une fonction.

4.3 Gestion des variables

Nous travaillons encore avec une représentation abstraite, indépendante de toute plateforme cible. De plus, les opérations consistant à évaluer le cycle de vie des variables (variable liveness) n'ont pas encore été effectuées, il est donc impossible de pouvoir associer les variables à des registres ou à des cases mémoires prédéfinies, on travaille donc exclusivement avec des temporaires. De même, ces variables ne sont jamais désallouées que l'on soit dans leur scope ou non vu que l'on ne dispose pas d'assez d'informations pour leur survie.

4.4 Gestion des appels de fonction

Le code intermédiaire d'entrée est censé contenir des instructions en rapport avec la gestion des frames. De par sa nature une frame est très dépendante de l'architecure visée. Notamment les appels de fonctions externes(malloc, init_array, les fonctions traitant les chaînes de caractères, ...) et les ajouts de prologue /épilogue dépendent de l'architecture matérielle. Il sera donc nécessaire d'implémenter un environnement pour les frames adapté à l'éxécution de la machine virtuelle.

4.5 Runtime

La machine virtuelle doit implémenter la runtime dans sa totalité. Il faut ainsi pouvoir non seulement allouer des données de type string ou array mais aussi gérer les entrées-sorties. La runtime sera implémentée comme partie intégrée de la VM et sera appelée à partir de celle-ci lorsqu'un appel à une fonction externe sera détecté grâce à la technique employée pour l'appel de fonctions. Le code emulé au travers de la VM sera donc branché sur du code built-in écrit dans la VM.

5 **Considérations supplémentaires**

5.1 Langage de développement

Au choix : Java ou C++ ou bien encore un dérivé de ML.

5.2 Contraintes d'environnement d'exécution

Le projet devra être installé et s'exécuter à partir de la plateforme informatique de l'EPITA.

5.3 Outils fournis

- Binaire tc qui génére une IR canonisée
- Jeu de tests

5.4 Partie supplémentaire : le debugger

Le debugger doit pouvoir se brancher sur la VM existante sans modification fondamentale de sa structure et fournir une exécution pas à pas ainsi qu'un système de breakpoint et d'affichage des variables au fur et à mesure de l'exécution. Il doit de plus posséder une interface homme-machine décente et ergonomique implémentée, selon le langage choisi, avec un framework standardisé (tel que swing ou QT).

5.5 Livrables attendus

- Procédure d'installation
- Machine virtuelle

- Manuel utilisateur du debugger
- Sources du projet convenablement commentées

5.6 Equipe de travail

Ce projet sera réalisé par une équipe de 2 à 3 Ingé1 et il sera mené en collaboration avec 2 Ingé2 SIGL (Wilfried de Denterghem & Hugues Delorme) qui fourniront une conception générale du problème à l'aide d'UML ainsi que des solutions aux divers problèmes de conception pouvant être recontrés.