PROJET GÉNIE LOGICIEL

2023/2024

Documentation de Conception du Compilateur Deca

Antoine Borget
Pierre Lin
Anass Haydar
Yassine Safraoui
Jihane Zemmoura

Groupe GL-47

Table des matières

1 Introduction

Le document suivant explore en détail l'implémentation d'un compilateur, en se concentrant sur divers aspects clés du processus. Une partie initiale aborde l'organisation de l'architecture des classes et leurs interdépendances. La section suivante examine les algorithmes et structures de données employés, mettant en évidence l'utilisation d'environnements pour les expressions et les types. L'approche pour la gestion des types, y compris les opérations arithmétiques, est également décrite, faisant appel à des structures comme EnvironmentType. La troisième section se penche sur l'étape C, mettant en évidence le code associé à la génération de code pour la machine IMA. Différentes classes utilitaires, telles que RegManager, StackManager, CondManager, ErrorManager, VTableManager, VTable et VMethodInfo, sont présentées pour expliquer leurs rôles respectifs dans le processus global de compilation. Ces éléments constituent une base cruciale pour la compréhension du fonctionnement du compilateur, sans nécessairement se référer aux numéros spécifiques de sections.

2 Architecture des classes et leurs dépendances

Le code utilisé pour générer le compilateur est disponible dans le dossier **src/main/**.

Le lexeur et le parseur sont générés grâce à l'outil **ANTLR4**, les fichiers concernés sont dans le dossier **src/main/antlr4/**, ainsi que dans le sous-répertoire **syntax/** du dossier **src/main/-java/**.

Les classes nécessaires à la vérification contextuelle d'un programme Deca sont dans le sous-dossier **context/** du dossier **src/main/java/**.

De même, les classes nécessaires à la génération de code d'un programme Deca sont dans le sous-dossier **codegen/** du dossier **src/main/java/**.

Enfin, le sous-répertoire **tree/** du dossier **src/main/java/** contient les classes utilisées par le parseur pour construire l'arbre du programme Deca. Dans ces classes, les méthodes **verifyX** correspondent aux vérifications contextuelles du programme Deca, ainsi qu'à la décoration de l'arbre. Puis les méthodes **codeGenX** correspondent à la génération de code assembleur à partir de l'arbre décoré par les méthodes **verifyX**.

3 Algorithmes et structures de données employés

3.1 Étape B

3.1.1 Environnements

Environnement des expressions

On a implementé cet environnement avec une pile de *HashMap*, cela permet de mettre en œuvre simplement l'opération notée '/' dans le sujet qui rajoute des définitions plus "locales" qui écrasent temporairement les définitions potentiellement déjà existantes d'un symbole, pour la remplacer par sa nouvelle définition dans un bloc par exemple.

En effet il suffit alors d'empiler l'environnement 2 sur l'environnement 1 :

```
public static EnvironmentExp empile(EnvironmentExp env1,
        EnvironmentExp env2) {
    EnvironmentExp copyEnv1 = env1.copy();
    EnvironmentExp copyEnv2 = env2.copy();
    EnvironmentExp dernier = copyEnv1;
    while (dernier.parentEnvironment != null) {
        dernier = dernier.parentEnvironment;
    }
    dernier.parentEnvironment = copyEnv2;
    return copyEnv1;
}
```

Et l'algorithme de recherche dans cet environnement est tout simplement :

```
public ExpDefinition get(Symbol key) {
    if (env.containsKey(key)) return env.get(key);

    if (parentEnvironment == null) return null;

    return parentEnvironment.get(key);
}
```

En effet, on cherche dans les dernières couches en priorité, pour accéder aux définitions les plus "locales".

Et pour déclarer, il suffit de *put* dans le dernier dictionnaire de la pile.

Nous avons également créé une opération d'union disjointe entre deux environnements. Celle-ci renvoie le premier symbole qu'elle trouve présent dans les deux environnements s'il y en a, et null sinon.

```
public Symbol disjointUnion(EnvironmentExp envExp) {
    for (Symbol s1 : envExp.getKeys()) {
        if (this.get(s1) != null) return s1;
        ExpDefinition def = envExp.env.get(s1);
        this.env.put(s1, def);
    }

    return null;
}
```

Environnement des types

Les fonctions partielles **type_unary_op**, **type_arith_op** et **type_binary_op** du sujet, qui à une opération donnée, et un ou deux types pour ses arguments, associent les types de retour, ont été implementées avec des *HashMap* dans la classe **EnvironmentType**. De cette manière on a crée une classe de clé pour chaque dictionnaire, et en instanciant cette classe avec une opération et un ou deux type, on peut savoir si les types sont compatible avec l'opération (si le dictionnaire contient la clé) et savoir le type de retour.

```
public Type getTypeUnaryOp(String op, Type type) {
    KeyTypeUnaryOp key = new KeyTypeUnaryOp(op, type);
    return typeUnaryOp.get(key);
}

public Type getTypeBinaryOp(String op, Type type1, Type type2) {
    KeyTypeBinaryOp key = new KeyTypeBinaryOp(op, type1, type2);
    return typeBinaryOp.get(key);
}
```

Le dictionnaire pour les opération arithmétiques a simplement aidé a remplir celui pour les opération binaires.

Pour ce qui est de la relationd de sous-typage, elle a également été implementée dans **EnvironmentType**, par une méthode suivant strictement les conditions du sujet :

```
public boolean subtype(Type type1, Type type2) {
    if (type1.equals(type2)) return true;
    if (type1.isClass()) {
```

```
            ClassType classType = (ClassType) type1;
            if (!type2.isClass()) return false;
            ClassType classType2 = (ClassType) type2;
            return classType.isSubClassOf(classType2);
        }
        return (type1.isNull() && type2.isClass());
    }
```

Quant à **assign_compatible**, on a fait quelque chose d'assez particulier pour faciliter les conversions *int* vers *float* :

On a implementé cela avec une méthode, toujours dans **EnvironmentType**, qui prend en argument le type de l'expression de gauche, et l'expression de droite.

```
    public AbstractExpr assignCompatible(Type type1,
                                         AbstractExpr expr2) {
        Type type2 = expr2.getType();
        if (type1.isFloat() && type2.isInt()) {
            return new ConvFloat(expr2);
        }
        if (type1.equals(type2) ||
                compiler.environmentType.subtype(type2, type1)) {
            return expr2;
        }
        return null;
```

On a besoin de l'éxpression de droite car si jamais on doit faire le cast *int* vers *float*, on va directement le créer dans cette méthode, et le renvoyer. S'il n'y a pas de cast à faire, on va renvoyer l'expression passée en paramètre, et si l'assignation n'est pas compatible, on renvoie *null*.

Ainsi quand on l'utilise on fait par exemple :

```
AbstractExpr expr =
    comp.environmentType.assignCompatible(expectedType, this);

    if (expr == null) {
        throw new ContextualError(...);
        // En disant que les types sont incomptables
    }
    if (expr != this) { // On test s'il y a eu un cast
        expr.verifyExpr(compiler, localEnv, currentClass);
    }
```

3.2 Étape C

Tout le code utilisé pour faire de la génération de code pour la machine IMA est présent dans les fonctions **codeGenX**. Dans ces fonctions, La façon de générer le code respecte toujours ce qui est décrit dans la partie [GenCode] (Page 209) du polycopié. Néanmoins, nous avons fait quelques optimisations (sur le code généré) que nous avons déjà décrites dans la documentation énergétique de notre projet.

Nous allons désormais présenter toutes les classes utilitaires (dans le dossier **codegen/**) utilisées dans les fonctions **codeGenX** afin de générer du code pour la machine IMA. Comme il n'y avait que le *compiler* en argument de ces fonctions dans le code fourni, nous avons décidé de stocker une instance de chacune de ces classes utilitaires dans le *compiler*, on peut alors accéder à ces classes via un *getter*. Il n'était pas possible de faire des classes en utilisant des attributs et méthodes statiques car cela pose problème lorsque l'on utilise l'option *-P* (pour compiler tous les programmes en parallèle).

3.2.1 RegManager

La classe **RegManager** concerne la manipulation des registres. Sa fonction principale est de contenir la liste des registres disponibles pour utilisation. La structure de données utilisée pour stocker les registres est une liste chaînée. Ceci permet de prendre un registre libre ou de libérer un registre en $\mathcal{O}(1)$ via les méthodes **removeFirst** et **addFirst**. Par exemple, pour générer le code pour $x + y$, on ferait :
- Prendre un registre libre dans la liste via **removeFirst** et générer le code pour y stocker x.
- Prendre un autre registre libre dans la liste et générer le code pour y stocker y.
- Générer l'instruction pour l'addition des deux registres.
- Libérer les deux registres via **addFirst**.

En plus de sa simplicité et de son efficacité, grâce à la liste chainée, nous avons pu implémenter l'option *-r X* très facilement. En effet, il suffit de ne pas ajouter les registres interdits dans la liste chaînée lors de son initialisation.

De plus, la classe contient un attribut *lastImm* qui est utilisé dans un but d'optimisation du code généré. En effet, cet attribut permet de regarder si le dernier code que l'on a généré via une fonction **codeGenX** était un immédiat ou non. Cela permet donc d'éviter certains *load* inutiles d'immédiat dans un registre.

Enfin, nous avons l'attribut *usedRegsStack* qui est utilisé pour sauvegarder tous les registres utilisés lors de la génération du code d'une méthode. Cet attribut est une pile de tableau de booléen car il faut gérer le cas des méthodes imbriquées. Après avoir généré le code pour une méthode, il y a des méthodes telles que **addSaveRegsInsts** qui permettent de générer le code nécessaire en fonction des registres utilisés.

3.2.2 StackManager

La classe **StackManager** concerne tout ce qui est gestion de la pile. Sa fonction principale est de stocker des compteurs du nombre de variables (globales ou locales à une fonction), du nombre de valeurs *push* temporairement dans la pile etc...

En utilisant ces attributs, il est aisé de générer le code pour la gestion de la pile (*tsto*, *addsp*...), que ce soit pour le programme principal, ou les méthodes.

De plus, la classe permet de facilement trouver la prochaine adresse libre dans la pile pour y stocker une variable globale ou locale à une fonction, notamment via **getOffsetAddr**.

3.2.3 CondManager

La classe **CondManager** a été créée pour la gestion des structures de conditions, mais elle est aussi utile pour les évaluations paresseuses des expressions booléennes.

Tout d'abord, elle permet d'avoir un label unique du type **LX** où **X** est un nombre augmentant à chaque utilisateur de la méthode **getUniqueLabel**. Bien que des labels ayant un nom du type **LX** rendent le débogage plus difficile, il était important pour nous d'avoir des noms le plus court possible, afin de limiter l'espace mémoire que prend le fichier assembleur généré. Nous avons bien-sûr fait cette modification de nom après avoir fait la validation de tout ce qui concerne les conditions dans notre compilateur.

De plus, il y a l'attribut *doingCondCount* qui est utile pour les évaluations paresseuses des expressions booléennes. En effet, grâce à elle, on peut savoir si nous sommes dans une structure conditionnelle (*if/else*, *while*) ou non, et générer du code en fonction de cela. Par exemple, si l'on est dans un *if/else*, il ne faut pas refaire la génération des labels qui sont déjà présents.

Enfin, toujours concernant les évaluations paresseuses des expressions booléennes, il faut noter que nous avons ajouté les attributs *isInTrue* et *branchLabel* dans la classe **AbstractExpr**. Cela nous a permis d'implémenter ces évaluations paresseuses comme décrit dans le polycopié.

3.2.4 ErrorManager

La classe **ErrorManager** concerne tout ce qui est gestion des erreurs à l'exécution. Pour chaque erreur possible, il y a un attribut de type *Label* qui lui est associée. De plus, il y a une *HashMap errMsgs*, qui pour chaque erreur, associe son label à un message d'erreur.

Ensuite, nous avons une autre *HashMap errMap*, qui pour chaque erreur, associe son label à un booléen représentant si l'erreur a été utilisée ou non. En effet, pour chaque erreur, il y a une fonction du type **getErrorNameLabel** qui retourne le label de l'erreur concernée, et met la valeur *true* dans *errMap* dans la case qui lui est associée.

Enfin, grâce à *errMap*, on peut facilement générer seulement le code des erreurs utilisées via la fonction **codeGenAllErrors**.

3.2.5 VTableManager, VTable & VMethodInfo

La classe **VTableManager** permet de stocker l'ensemble des tables des méthodes du programme Deca compilé. Cela est stockée sous la forme d'une *HashMap*, qui pour chaque classe, associe son nom à sa table des méthodes (**VTable**). Il y a ainsi une multitude de méthode donnant des informations sur la table des méthodes d'une classe, tout en $\mathcal{O}(1)$.

La classe **VTable** représente la table des méthodes d'une classe. On y stocke :

- Le nom de sa classe mère (*Object* par défaut).
- L'adresse de la table dans la pile.
- Les informations des méthodes de la classe avec une *HashMap*, qui pour chaque méthode, associe son nom à ses informations (**VMethodInfo**).
- Les informations sur les attributs de la classe avec une *HashMap*, qui pour chaque attribut, associe son nom à l'index de sa déclaration dans la classe.

De même, il y a des méthodes pour avoir des informations sur les méthodes et attributs de la classe en $\mathcal{O}(1)$.

La classe **VMethodInfo** permet de représenter une méthode d'une classe, on y stocke :

- Le nom de la classe **où la méthode est déclarée**.
- L'index de la méthode.
- Les informations des paramètres de la méthode avec une *HashMap*, qui pour chaque paramètre, associe son nom à son index.

Encore une fois, on peut accéder à toutes les informations en $\mathcal{O}(1)$.

3.2.6 Autres Classes

La classe **CodeGenUtils** contient une seule méthode **extractAddrFromIdent** qui permet d'avoir l'adresse d'une variable. Par exemple, considérons la méthode :

```
void f() {
    println(x);
}
```

Il est difficile de savoir si x est une variable locale de la méthode, un argument de la méthode, ou un attribut de la classe de la méthode. La méthode **extractAddrFromIdent** sert justement à trouver quel est ce x et renvoyer son adresse en utilisant l'ensemble des tables des méthodes.

La classe **LabelUtils** contient des méthodes pour avoir par exemple, le label d'une méthode. Elle sert essentiellement à éviter les bugs si on répète plusieurs fois le calcul d'un label dans différentes parties du code.

Enfin, il y a les classes **GameBoyManager** et **GameBoyUtils** qui concernent tous deux la génération de code pour notre extension GameBoy.