Rapport PDS : Compilateur VSL+

Antoine POSNIC, Antoine LEVAL

December 12, 2017

Abstract

Vous pouvez retrouver la version finale du projet sur ce dépôt Github :
https://github.com/Viinyard/PDS_TP2

1 Introduction

Pour ce projet nous avons choisis d'utiliser Java/Antlr, trouvant cette approche objet pour l'ASD plus pratique et l'outil antlr très puissant.

Nous devions réaliser au cours de ce TP un compilateur pour le langage VSL+ (Very Simple Langage +). Nous devions comme au projet précédent générer l'ASD de ce langage, puis à partir de cet ASD réaliser la vérification de type puis la génération de code trois adresses LLVM.

2 VSL+ : spécifications

Le langage VSL+ ne comporte que des variable de type entier, et des chaîne de caractères. Il ne comporte pas de pointeur. Autorise uniquement les tableau à une dimension (d'entier). Autorise la redéfinition de variables dans des blocs différents, exemple :

```
FUNC VOID main() {
   INT y
   READ y
   { INT y
     READ y
     PRINT "y a l'interieur vaut ",y
   }
   PRINT ", mais a l'exterieur du bloc il vaut ",y
}
```

Un deuxième exemple de redéfinition de variable :

```
FUNC VOID main(f) {
   INT f
   f:=100
   PRINT f
}
```

Cette redéfinition de variable peu commune à d'autre langage nous a causé des problèmes par la suite ou nous avions dû revenir en arrière dans notre code pour implémenter la table de symbole de manière différente.

Le langage VSL+ ne comporte pas de terminaison d'instruction, nous pension au début devoir ordonner au compilateur d'interpréter le saut de ligne comme une fin d'instruction, mais nous avons voulu attendre d'être obligé de l'implémenter pour le faire, ce qui n'arriva jamais, les instructions se terminent par le début d'une autre sans aucun problème d'ambiguïté.

3 LLVM : fonctionnement

LLVM pour *Low Level Virtual Machine* est un langage intermédiaire permettant d'optimiser le code à la compilation, et de gérer l'éditeur de lien. (Exemple avec la fonction `printf`, et `scanf`.

LLVM permet donc de créer une machine virtuelle et se comporte donc comme Java/JGroovy, ou autre langage fonctionnant sur une machine virtuelle. LLVM propose un compilateur nommé clang qui compile le code trois adresses, le C le C++ et d'autres langage.

On peut facilement l'expérimenter via la commande : `clang -S -emit-llvm exemple.c` qui génère un fichier exemple.ll en code trois adresse LLVM-IR donc. Cette commande, en plus de la document LLVM officielle, de mieux comprendre le fonctionnement de sa représentation intermédiaire. Je me suis donc largement inspiré de la représentation intermédiaire trois adresses du langage C lorsque la documentation LLVM n'était pas clair sur le sujet, ou lorsque des choix de conception s'imposaient. Cette commande m'a aussi permis de comprendre certains bugs lors du développement du compilateur VSL+, en comparant ce que antlr produisait avec ce que clang produisait sur le même programme VSL traduit en C.

Pour ce qui est des variables temporaires la documentation LLVM précise à propos des identificateurs : *Unnamed temporaries are numbered sequentially (using a per-function incrementing counter, starting with 0). Note that basic blocks and unnamed function parameters are included in this numbering.*

Si on lis l'exemple de code compilé fournis dans le sujet et le code fournis pour commencer le projet on peut voir que toute les variables temporaires sont nommé comme ceci : `%tmp1, %tmp2, %tmp3` etc ..

Mais LLVM les nommes (par soucis d'optimisation) comme ceci `%0, %1, %2, %3` etc.. À noter que à chaque début de fonction se trouve un label d'entré, qu'on peut omettre puisqu'il y est par défaut, néanmoins il est possible de le placer, celui-ci est nommé `%0`, ainsi les variables temporaires dans la fonction commencerons à `%1`, néanmoins si l'on décide de nommer ce label d'entrée par exemple `%entry`, alors les variables commencerons cette fois par `%0`.

Ce nommage inclus donc le label d'entré de la fonction mais aussi (avant ce label), les arguments de la fonction. Ainsi pour une fonction avec deux arguments son premier sera nommé `%0`, son second `%1`, le label `%2`, et les variables temporaires suivante `%3, %4` etc..

4 Implémentation

4.1 Problèmes rencontrés

Comme énoncé plus haut la partie nous ayant causé le plus de difficultés fut la redéfinition de variable, je n'ai en effet jamais croisé ce genre de redéfinition avec d'autres langages. Certe le scope des variables, accessible dans un sous bloc mais non visible dans un bloc parent était déjà implémenté depuis le début, mais le fait de les nommer par exemple pour l'instruction `INT x : alloca i32 %x` fait que clang refusait que je renomme une autre variable temporaire avec le même identificateur.

Pour pallier à ce problème, nous avons choisis de ne pas nommer les variables locales avec leur nom, mais avec la convention de nommage d'LLVM sur les variables temporaires comme vu plus haut. Cependant si l'on adopte cette convention de nommage, clang nous génère une erreur à la compilation si l'indice des variables passe de 7 à 9 par exemple, ou si elles ne sont pas strictement ordonnée. On peut voir cela comme un compteur de programme. À cause de cela nous avons rencontré un autre problème par la suite. Voyons d'abord les choix d'implémentation qui nous ont mené à ce problème.

4.2 Choix d'implémentation

Le premier choix de conception s'est faite suite à une longue réflexion au sujet de *"où placer la table des symbole ?"* Sûrement pas du côté LLVM, mais il aurait très bien pu trouver sa place dans l'ASD ou dans la grammaire. Cependant la placer dans l'ASD aurait soit signifié *variable globale*, soit de la passer en paramètre de chaque objet de notre ASD, ce qui aurait été très laborieux. Placer la table de symbole en variable globale serait un peu comme tricher, ce n'est pas comme cela qu'on a pu le voir en TD, et c'était interdit lors du TP précédent de placer une variable globale.

Nous avons donc choisis de la placer la faire hériter dans la grammaire à partir du nœud de départ : `program`. Les instructions sont elles synthétisé évidement.

L'inconvénient de cette méthode est que nos variables temporaires qui doivent donc être créées dans le bon ordre pour être correctement ordonnée, sont créée au niveau de la grammaire. Sauf que pour certaines instructions il est nécessaire de créer d'autres variables temporaires, pour des

calculs intermédiaire. Pour un calcul par exemple. Celles-ci sont créées dans l'ASD, hors elles ne sont pas crées lors de la création de l'objet mais lors de l'appel de la méthode `toLlvm()`, qui se fait plus tard. Ce qui a pour cause de rompre l'ordonnancement des variables temporaires.

Pour pallier à ce problème, nous avons décidé de ne plus définir le nom des variables. En effet que ce soit en paramètre, nul besoin de garder le nom puisque c'est l'ordre qui compte (le premier argument est nommé `%0`), et la force de Java est que tout est objet, il nous suffit donc d'avoir un objet de type variable par exemple. A chaque fois que l'on doit instancier une nouvelle variable dans notre ASD on crée un objet de type variable, dans lequel on ne définis pas son nom, on fait de même lors des calculs intermédiaire. Pour expliquer la suite du déroulement voyons d'abord comment est implémenté l'objet variable :

```
public static class Variable extends Expression {
     public static final int local_scope = 0;
     public static final int global_scope = 1;
     public static final int no_scope = 2;
     private Llvm.Variable llvm_Variable;
     Type type;
     int scope;
     String value;
     public Variable(int scope, Type type) {
        this(scope, type, null);
     }
     @Override
     public boolean equals(Object other) {...}
     public Variable(int scope, Type type, String value) {
        this.type = type;
        this.scope = scope;
        this.value = value;
     }
     public Llvm.Variable toLlvm() {
        if (this.llvm_Variable == null) {
           this.llvm_Variable = new Llvm.Variable(this.scope, type.toLlvmType(),
               this.value);
        }
        return this.llvm_Variable;
     }
     @Override
     public String pp() {...}
     @Override
     public RetExpression toIR() throws TypeException {
        return new RetExpression(new Llvm.IR(), this);
     }
  }
```

On peut remarquer que dans la méthode `toLlvm()` se trouve une sorte de pattern singleton pour la variable `llvm_Variable`; car en effet, comme nous utilisons désormais un objet et plus une chaîne de caractère pour notre table des symbole, il faut veiller à ce que notre objet variable ne soit pas cloné, elle doit être conservée, pour chaque objet Variable de notre table de symbole doit correspondre un seul objet unique `Llvm.Variable`.

Deuxièmement on peut remarquer un attribut de type scope, qui nous sert à déterminer si la variable dans la LLVM doit être précédé d'un Les variables globales de types String, ont alors le scope `global_scope`, les autres variables locale le scope `local_scope`, et enfin les variables définit à la compilation comme les entiers dans les calculs le scope `no_scope`.

Afin de voir la suite du processus voyons maintenant l'objet `Llvm.Variable` :

```
public static class Variable extends Instruction {
     Type type;
     private String value;
     private int scope;
     private static final String[] scope_token = { "%", "@", "" };
     public Variable(int scope, Type type, String value) {
        super(null, 0);
```

```
        this.type = type;
        this.scope = scope;
        if(value != null) {
           this.value = Variable.scope_token[this.scope] + value;
        } else {
           this.value = value;
        }
     }
     public String getValue() {
        if(this.value == null) {
           this.value = Variable.scope_token[this.scope] + this.getCpt();
        }
        return this.value;
     }
       ...
  }
```

On vois que dans la méthode `getValue()` est implémenté token La méthode `getValue()` définit donc par récursion le nom des variables. La classe parent Instruction s'occupe de construire une liste chaînée d'instruction à partir d'un nœud `Fonction`.

```
public static abstract class Instruction {
     private Instruction pred = null;
     protected int level;
     int cpt;
     public Instruction(Variable result, int cpt) {
        this.level = Llvm.lvl;
        this.cpt = cpt;
        if(result != null) result.setParent(this);
     }
     public void setParent(Instruction parent) {
        this.pred = parent;
     }
     public int getCpt() {
        if(this.pred == null) {
           return -1;
        }
        return this.pred.getCpt() + this.cpt;
     }
     public abstract String toString();
  }
```

Ici la méthode `getCpt()` se charge d'obtenir le nom de la variable, à chaque instanciation de variable on appelle le super constructeur `Instruction` en lui fournissant une valeur `cpt`, 0 ou 1 car il y a en effet des instructions qui n'incrémentent pas le nom des variables, comme une référence à une variable, etc. On lui fournis également son parent pour permettre de faire le lien.

Une fois la compilation fini on print le langage LLVM IR, grâces aux méthodes toString des objets Instruction, quand le programme a besoin du nom d'une variable il appelle la méthode `getValue()` de l'objet Variable, qui appelle la méthode `getCpt()` de sa classe mère Instruction. La méthode getCpt() remonte jusqu'au nœud `Fonction`, qui vaut -1, donc l'instruction suivante (son premier argument vaut `%0` et ainsi de suite.

Cette façon de nommer les variables temporaire grâces aux références d'objet est très pratique et empêche l'apparition d'erreur car très robuste. Il est dit dans la documentation LLVM que le compilateur n'a que faire du nom des variables, on a donc choisis de faire de même, ne pas se soucier du nom des variables, on peut les simuler par références en Java.

4.3 Les tableaux

Pour implémenter les tableaux nous avons décidé de ne pas suivre le code proposé en exemple dans le sujet (allouer X fois la taille d'un entier sur un pointeur), mais de suivre cette syntaxe :

```
%2 = alloca [3 x i32]
```

```
%3 = getelementptr inbounds [3 x i32], [3 x i32]* %2, i32 0, i32 0
```

Nous avons fait ce choix afin de pouvoir utiliser le même code pour les tableaux que pour les chaînes de caractères, représenté en LLVM sous forme de constantes globales de tableau de caractères.

Ainsi la seule différences entre la référence d'un entier d'un tableau et une chaîne de caractère c'est l'indice. Pour référencer une chaîne on fournis toujours le premier caractère l'indice 0 du tableau de caractère.

Le seul inconvénient est lorsque l'on passe un tableau en paramètre de fonction, puisque la taille du tableau passé en paramètre n'est pas connu à la compilation il faut le passer sous forme de pointeur sur le premier élément du tableau, comme ceci :

```
%50 = load i32*, i32** %3
call void @naivesort(i32* %50, i32 %52)
```

```
define void @naivesort(i32*, i32) {
; <label>:2
  %3 = alloca i32*
   ...
   store i32* %0, i32** %3
   ...
   %12 = load i32*, i32** %3
  %13 = getelementptr inbounds i32, i32* %12, i32 %11
```

Ce qu'on peut observer ici, c'est que le fonctionnement reste tout à fait similaire mise à part le type qui est remplacé par un pointeur d'entier.

4.4 Les fonctions

Pour les fonctions nous avons de ne pas générer le code de retour `ret i32 %..` à chaque instruction de retour, mais d'instancier une variable au début de la fonction. Ainsi à chaque instruction de retour nous affectons à cette variable la valeur à retourner et nous faisons un saut vers le dernier label de la fonction (qui comporte l'instruction de retour), par exemple :

```
; <label>:28
  store i32 1, i32* %5
  br label %30
...
; <label>:30
  %31 = load i32, i32* %5
  ret i32 %31
}
```

Cela nous permet de générer une erreur à la compilation lorsqu'une instruction `RETURN` ... Voyons pour illustrer cela la grammaire définissant une fonction :

```
fonction [SymbolTable table] returns [ASD.Expression out]
  @init {
    SymbolTable funcTable = new SymbolTable(table);
  }
  : FUNC_TYPE type ident {
      ASD.Variable returnVariable = null;

      if(!($type.retType instanceof ASD.VoidType)) {
        returnVariable = new ASD.Variable(ASD.Variable.local_scope, new
            ASD.PointerType($type.retType));
        funcTable.add("RETURN", new SymbolTable.VariableSymbol(returnVariable));
      }
```

Si le type de retour est différent du type `VOID`, on place une variable `returnVariable`, avec la clé `"RETURN"` (cette clé est aussi un mot clé réservé du langage VSL+, aucun risque donc de déclarer

une autre variable avec ce même nom).

Ensuite lorsque l'on croise une instruction `RETURN` :

```
RETURN_STMNT expression[table] {
     SymbolTable.Symbol s = table.lookup("RETURN");
     if(s == null) { throw new IllegalExpressionException("Cannot return with the type
         void"); }
     expressions.add(new ASD.Affectation(s.variable, $expression.out));
   }
```

On vérifie si la clé `RETURN` est bien présente dans la table des symbole, si ce n'est pas le cas alors on ne peut pas retourner de valeur (type void par exemple), et gère également lors de l'affection dans l'ASD le conflit de type.

4.5 L'ASD

Pour la génération de l'ASD nous avons choisis de donner le super type `Expression` à toute les instructions. En effet toutes les instructions ont besoin de seulement deux méthodes : `toIR()` et `pp()`, mise à part les variables que nous n'avons pas besoin de traduire en code intermédiaire telle quel, car elles doivent faire parti d'une instructions, c'est la méthode `toIR()` qui s'occupe de les transformer en variable LLVM.

5 Conclusion

Lors de ce projet nous avons dû à de nombreuses reprise supprimer et recommencer du code existant, suite à des problèmes auxquels nous n'avions pas penser au début, comme par exemple le faire qu'on ne peut faire hériter quelque chose sur une grammaire récursive gauche.

En effet nous avions choisis afin de simplifier la compréhension et pour profiter de la puissance d'antlr qui s'occupe lui même de supprimer la réflexivité à gauche, de le laisser gérer lui même la priorité des expressions. En effet plus une instructions est placé en haut plus elle est prioritaire, ainsi nous avions géré la priorité des expressions ainsi :

```
expression returns [ASD.Expression out] :
    PO expressio PF {
      $out = $expression.out;
    }
    |
      l=expression op=(MUL | SDIV) r=expression {
        switch($op.getType()) {
        case MUL :
          $out = new ASD.MulExpression($l.out, $r.out);
          ..
        }
      }
    |
      l=expression op=(ADD | SUB) r=expression {..}
    | c=atome {..}
    ;
```

Mais lorsque nous avons dû implémenter les variables en atome, il nous a fallu faire hériter la tableau des symboles, ce qui n'est pas possible dans une grammaire récursive gauche.

Alors nous avons remplacé ce code par l'exemple du cours :

```
expression [SymbolTable table] returns [ASD.Expression out, ASD.Type retType] :
     l=multExpr[table] {...} (op=(ADD | SUB) r=multExpr[table] {...})* ;

multExpr [SymbolTable table] returns [ASD.Expression out, ASD.Type retType] :
    l=atome[table] {...} (op=(MUL | SDIV) r=atome[table] {...})* ;

atome [SymbolTable table] returns [ASD.Expression out, ASD.Type retType] :
     PO expression[table] PF {..}
```

```
    |
      ident {...}
... (tableau, appel de fonction, string, entier ..)
```

Un autre problème énoncé plus au a été que VSL+ autorise la redéfinition de variable, qui nous a fait changer la façon de nommer les variables temporaires, ensuite le fait que des variables temporaires étaient créé dans l'ASD puisse briser l'ordonnancement des variables, toute ces erreurs de parcours qui nous ont fait supprimer une bonne partie de notre code pour le ré-écrire.

Nous avons également rencontré une erreur pour compiler ce genre de boucles

```
WHILE n-1 DO {
   INT x
   x := 1
} DONE
```

En effet nous obtenions une erreur de type segmentation fault, car nous faisions plusieurs allocations sur une même adresse. Il nous a fallu changer le fonctionnement de la classe IR car nous utilisions déjà la liste `header` pour nos variables globales. Nous avons donc placé les variables globales dans une liste `global`, nos instanciation dans une liste `header`, et le reste dans une liste `code`.

5.1 Version finale

Après toute ces erreurs nous sommes néanmoins arrivé à une version totalement fonctionnelle, gérant 100Nous n'avons néanmoins pas implémenté les deux levelerror optionnel, comme la gestion d'appel de fonction prototypé mais non définis et les programme sans fonction main, pour plusieurs raisons. Afin de ne pas prendre le risque de casser notre code fonctionnel, et car c'est dommage pour un langage de forcer chaque fichier de code à posséder une fonction main ou de définir toute les fonction appelée, car cela restreint l'évolution du compilateur à l'implémentation des classes. En effet rien n'indique au compilateur que le main ou la définition de la fonction ne puisse se trouver dans une autre classe.