Output C Code

Compiler Construction '16 Final Report

Cdric Viaccoz Bastian Nanchen
EPFL
{cedric.viaccoz}@epfl.ch

1. Introduction

The first part of the lab was to implement a compiler for Toy Object-Oriented Language (Tool), which is a small object-oriented programming language.
This has led us to conceive a Lexer, a $LL(1)$ grammar, a Parser, a Type Checking system and to generate binary code for the Java Virtual Machine.
The aim of this project is to replace code generation for the Java Virtual Machine with a pretty printer, which outputs C code.
The difficulty is to write C code, which is an imperative programming language, from an object-oriented programming language.

2. Examples

First it is important to see the representation of an object in C.

```
class Rectangle {
  var width: Int;
  var height: Int;

  def area(): Int = {
    return width * height;
  }
}
```

This Tool code will be pretty-printed in C like this:

```
struct Rectangle{
  int width;
  int heigth;
  int (*area)(struct Rectangle*);
}

int Rectangle_area(struct Rectangle* this) {
  return this->width * this->height;
}
```

The class are represented in C by a structure. The variables of the class are members of the structure. They are declared in the same fashion.
Contrariwise the method of the class is declared outside of the structure. In C, a structure can't have a function declaration as a member. The function shall be declared outside. But the structure has a pointer to the function.
The function is declared with the name of the class as prefix followed by an underscore. Thus the belonging to the "Rectangle" class is suggested.
Therefore this is how the C code mimics the behavior of an object-oriented class.
There exists other dilemma like inheritance, dynamic dispatch,..., which be discussed in the implementation section.

3. Implementation

This is a very important section, you explain to us how you made it work.

3.1 Theoretical Background

3.2 Implementation Details

Two files were created to implement this project.

3.2.1 Data Type

The first one is "CDataType.scala". It contains a class that depicts the C's structure.

```
class StructDef(val name: String,
  val membersList: ListBuffer[StructMember])
```

A trait "StructMember" is extended by two class to implement the two sort of structure's members, which are a variable or a pointer to a function (as explained in section 2. Examples). Without forgetting to define a class "FunctionPtr". It defines the fact that each func-

tion have as first parameter a generic pointer to the calling structure.

```
class StructVar(val name: String,
  val tpe: CType) extends StructMember {
  // some code
}

class StructFunctionPtr(val ptr: FunctionPtr,
  var mtDcl: MethodDecl) extends StructMember {
  // some code
}

class FunctionPtr(val name: String,
  val retType: CType, val args: List[CType]) {
  // some code
}
```

3.2.2 Code Generation

The second file is "COutputGeneration.scala". Its skeleton and behavior is based on the "CodeGeneration.scala" completed to output binary code for the Java Virtual Machine. The program uses the same architecture of methods like: "cGenMethod", "cGenStats", "cGenExpr". Each of this method returns a "StringBuilder". The final is a .h and .c, which encloses all the StringBuilders concatened.

Inheritance bonjour

Dynamic dispatch

Method chaining This problem was one, that we had not seen coming.
In Tool, there is the possibility to write multiple call to a method on one line. It works because each method returns an object. We can't directly translate it in C programming language. To tackle this problem, we created a simple object: "tmpVarGen".

```
object tmpVarGen{
  private var counter = 0
    def getFreshVar: String = {
      counter += 1
      return "tmp"+counter
    }
    def getLastVar: String = "tmp"+counter
}
```

It is used to have a unique variable name. It has two methods. "getLastVar" returns the last created variable name. "getFreshVar" return a new variable name.
Now at each evaluation of an AST element, which is an expression, it is important first to evaluate the expressions and assign to intermediate variables using the object "tmpVarGen". Next the expression is written using the variables.

```
case Equals(lhs: ExprTree, rhs: ExprTree) =>
  val lhsString = cGenExpr(lhs)
  val lhsLastVar = tmpVarGen.getLastVar
  val rhsString = cGenExpr(rhs)
  val rhsLastVar = tmpVarGen.getLastVar
  val andExprResultVar = genTabulation(indentLvl)+
    CInt.toString()+" "+tmpVarGen.getFreshVar+" = "+
    lhsLastVar+" == "+rhsLastVar+";\n"
  return lhsString.append(rhsString).append(andExprResultVar)
```

Here it is an example with the evaluation of an expression AST element: Equals. The left-hand side and right-hand side expressions are first evaulated using the "cGenExpr()" method and are stored in intermediate variable using the "tmpVarGen" object. Afterward the equal expression as known in C: "lhs == rhs" is written using the intermediate variable. At the end we append all element together. Thus one Tool line to write an equality is three lines in C programming language.

4. Possible Extensions

If you did not finish what you had planned, explain here what's missing.

In any case, describe how you could further extend your compiler in the direction you chose. This section

should convince us that you understand the challenges of writing a good compiler for high-level programming languages.

References