Output C Code

Computer Langage Processing '16 Final Report

Cédric 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 make object-oriented code into C code which does not have built in data types to support class, as it is an imperative programming language .

2. Examples

The example below presents how a class and the inheritance could be depicted in C programming language.

```
class Animal{
  var isAPet: Bool;
  var speed: Int;
  def sleep(): String = {
    return "zzz...";
  }
}

class Dog extends Animal{
  def bark(): String = {
    return "Woof!";
  }

  //override
  def sleep(): String = {
    return "zzz... wouaf... zzz...";
  }
}
```

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

```
struct Animal{
   int isAPet;
   int speed;
  char * (*sleep)(void *);
};

struct Dog{
  int isAPet;
  int speed;
  char * (*sleep)(void *);
  char * (*bark)(void *);
};

char * Animal_sleep (void * this) {
  // some code that will be depicted in the
      implementation part
}

char * Dog_sleep (void * this) {
  // some code that will be depicted in the
      implementation part
}

char * Dog_bark (void * this) {
  // some code that will be depicted in the
      implementation part
}
```

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.
The variables of the class "Animal", they also find themselves in the "Dog" structure because the class "Dog" inherits from the class "Animal".
Contrariwise to Tool the methods of the class are declared outside of the structure. In C, a structure can't have a function declaration as a member, thus 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 "Dog" class is suggested, and this allows overriding (a same function with two implementation

according to the class).
Therefore this is how the C code mimics the behavior of an object-oriented class.
There exists other dilemma like method chaining, dynamic dispatch,. . . , which will be discussed in the implementation section.

3. Implementation

3.1 Theoretical Background

The only background needed for this extension is knowing the tool's AST and the C language. However since we are trying to emulate an object-oriented language into one that is not, the book "Object-oriented programming with ANSCI-C"[**?**] has been useful for some concepts and it will be referenced throughout this report.

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 classes to implement the two kinds of structure's members, which are a variable or a pointer to a function (as explained in section 2. Examples). The pointer to a function holds two fields, one for an internal representation of a C function pointer (the class "FunctionPtr"), and one for the AST version of the method declaration that should be linked to the pointer.

```
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
}
```

In this file is also defined the CType abstract class. This allows us to represent C version of the Tool types. We only have 4 object that extends this abstract class : CInt, CIntArray, CStruct and CString. There is not an equivalent of the type Bool since boolean values are treated as int in C language. All classes of Tool are represented by the CStruct type, since struct are used to emulate class in our C code printed.

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 output is two files, a .h and .c, which encloses all the StringBuilders concatened.

Overview The main method of the object `COutputGeneration` is `def run(ctx: Context)(prog: Program)`, which takes in parameter the AST of the code.
The method starts by creating a `StringBuilder` for the preprocessor directives. It includes the headers `stdio.h, string.h and stdlib.h`. It contains `#define INT_MAX_LENGTH 12`, which is useful in order to have correct Int to string of characters concatenation (12 being the number of digit of -2^{32}, the most wide number and int can hold.). It also encloses the definition of default constructor macros for every classes of the program. We will get back on this point later in the section implementation

```
#define nAnimal 0
#define nDog 1
```

Listing 1. In the case of the example used in section 2, this would be the macros produced

Finally, a `StringBuilder` contains the inclusion of the file's header, which is directly created.
Afterward a method takes care of generating the structures corresponding to the Tool's classes:
`def genStructDef(ct: ClassDecl): StructDef`. The method takes as only parameter a `ClassDecl` and returns a corresponding `StructDef`, which was presented in section 3.2.1 Data Type. Then the `StructDef`'s method `toStringRepr` is called to return the structures

as String to be printed.
Thereafter the method manages the program's methods.

```
def genMethods(ct: ClassDecl): StringBuilder =
       (for(mt <- ct.methods)yield(cGenMethod(ct, mt))).foldLeft(new StringBuilder())((a,b) => a append b)
```

The method `cGenMethod` traverses the AST node `MethodDecl` to translate the statements and the expressions in C programming language, using the methods `cGenStats` and `cGenExpr` like in the Code Generation lab. The method returns a `StringBuilder` representing a function, that corresponds to a Tool's class method.
Then the Tool's main method is translated into C programming language.

```
def genMainMethod(main: MainObject): StringBuilder = {
  val mainMethod = new StringBuilder("int main(void){\n")
  main.stats.foldLeft(mainMethod)((sB, stmt) => sB append(cGenStat(stmt)(1, None)))
  return mainMethod.append("\treturn 0;\n}")
    }
```

The `genMainMethod` method receives as parameter the AST node `MainObject`. It is converted as the `main` function in C programming language.

```
int main(void){
  return 0;
}
```

`MainObject` contains a list of `StatTree`. Each `StatTree` is evaluated by the method `cGenStat` and the returned `StringBuilder(s)` is the body of the C `main` function.
In the `run` method, there exists an object `defaultConstructor`. It represents the construction of the default constructor. At the end of the run method, it will produce an helper C function to be printed with every program. The function is called `new`. It takes as only parameter an integer. It is one of the defined integer for each Tool's class, that we have seen before in Listing 1.
Subsequently a method `addStructConstructor` takes care of adding a `case` element to the `new` function for a correct initialization and allocation of a Tool's object in C programming language.

```
void * new(int type){
  void * object;
  switch(type){
    case nAnimal:
      object = malloc(sizeof(struct Animal));
      ((struct Animal *) object)->sleep = Animal_sleep;
      break;
    case nDog:
      object = malloc(sizeof(struct Dog));
      ((struct Dog *) object)->bark = Dog_bark;
      ((struct Dog *) object)->sleep = Dog_sleep;
    default:
      return NULL;
  }
  return object;
}
```

Listing 2. In the case of the example used in the section 2

Since this `new` function will be called each time a `new` was written in Tool, class methods needs to have it visible. Because `new` references all the function defined, it can't be put before them in our outputted C program, so its prototype is included in the .h file. And it is the sole purpose of the .h.

At the end, the functions `void helper_reverse_plus(char str[], int len)`, `char* itoa(int num)` and `int * arrayAlloc(int size)` are constructed. Their usefulness will be explained in the subsections to come.
Finally all these snippet of C code are concatenated and written in a .c file.

Dynamic dispatch Dynamic dispatch is accomplished during the default construction of a new Instance of a class. In Tool, there is only on type of constructor, the default one. One idea we borrowed from our reference book[1] was to have a default method be outputed called `new` which would take care or creating a new Instance of a struct and returning it as a generic pointer `void *`. This function was already presented in the overview. `void *` is the type used in C to represent class, getting class member or function is then accomplished by casting this generic pointer to the corresponding struct.

And according to the overriding of the methods or not, this method also takes care of setting the value of the struct's function pointers to the corresponding function. To be sure that calling `sleep` on an instance

[1] www.cs.rit.edu/ ats/books/ooc.pdf at page 11, si jamais faudra mettre ca dans la bibliography la fin scuse

of Animal will get the correct pointer, for inherited structs, we make sure they hold the same fields in the same place than their parent struct. If we look at the struct definitions of example 2, we clearly see that the function pointer `sleep` is held in the same place in both structs. Therefore, if we consider this Tool code snippet :

```
program Sleep{
  println(new Farm().getPet(0).sleep());
  //prints ``zzz...''
  println(new Farm().getPet(1).sleep());
  //prints ``zzz... wouaf... zzz...''
}

class Farm{
  def getPet(sel: Int): Animal = {
    if(sel == 0){
      return new Animal();
    }else{
      return new Dog();
    }
  }
}
```

Running it will result with the sleep message of animal printed and followed by the sleep message of a dog. This is an exemple of dynamic dispatch. Using casting, then the same result can be obtained in C :

```
int main(void){
  void * farm0 = new(nFarm)
  void * animal0 = ((struct Farm
      *)farm0)->getPet(0);
  printf("\%s\\n", ((struct Animal *)
      animal1)->sleep())
  //prints ``zzz...''
  void * farm1 = new(nFarm)
  void * animal1 = ((struct Farm
      *)farm1)->getPet(1);
  printf("\%s\\n", ((struct Animal *)
      animal1)->sleep())
  //prints ``zzz... wouaf... zzz...''
}
```

Casting to Animal is made in both case because `getPet` returns an instance of Animal so the compiler treat the value returned to be of type "Animal". Even if a Dog is returned it will use the `sleep` function of the dog since the pointer to `sleep` in Dog's struct is in the same place as the one of Animal's struct.

Length of Arrays Another problem encountered was retrieving an array's length. In Tool, getting it is made in the same way as Java, by accessing the field `length` of the array. However this cannot be translated easily in C if the array is not statically allocated. The way we treat arrays of ints in our extension is by defining them as a pointer to a value of type int. Then creating a new array of n elements is made by using calloc with n and `sizeof(int)` as arguments. The pointer returned points now to a region of the memory that can hold n ints. We will call this pointer "x". And since our array is juste a pointer, trying the trick of `sizeof(x)/ sizeof(int)` to get its length will either be equal to 1 or 2 (depending on the system word size) because `sizeof(x)` in this case return the size of the pointer, not the size of the memory this pointer was allocated to.
To resolve this problem, we defined an helper function called `arrayAlloc` printed with every program.

```
int * arrayAlloc(int size){
  int * smrtArray = calloc(size + 1, sizeof(int));
  smrtArray[0] = size;
  return (smrtArray + 1);
}
```

What this function does is pretty simple but ingenious. The array is callocated for one more element than the size required, and then the length of this array is stocked at the first place of the callocated array. We return then the value of the pointer incremented by one. Thus, when we need to get the length of the array "x", we can get it from the value hold at the place "-1" of the pointer "x"

```
int * x = arrayAlloc(42);
int xLength = *(x - 1) //xLength is equal to 42.
```

Concatenation The AST node "Plus" asked for some attention. The Tool language allows to apply the "+" operator on Int and String operands.
The situation, where the two operands are Int, is straightforward to emulate in C programming language. It's also an addition.
The event where the two operands are String is a little bit more tricky. It is necessary to allocate memory before concatenate the two string of characters.

```
strcpy(malloc(strlen(lhsString) +
    strlen(rhsString) + 1), lhsString, rhsString)
```

Finally when the two operands are different, we were obliged to create two C functions:
`void helper_reverse_plus(char str[], int len)` and `char* itoa(int num)`. These functions transform

an integer into a string of characters and return it. These two functions are written in every .c file generated by the compiler.

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 with the model we choosed to make object oriented possible. Our "methods" need to have as a first argument a pointer to the struct calling, so to make a method call on a struct we would need to write : `void * a = new(nA); ((struct A *)a)->foo(a)`. Our variable `a` needs to be referenced two times, first to get the good function pointer and then as the first argument if `foo` needs to access fields defined in A. Therefore chained method calls need intermediate variables to work correctly. To tackle this problem, we created a simple object: "tmpVarGen".

```
object tmpVarGen{
  private var counter = 0
  private var lastSuffix = ""
  def getFreshVar(suffix: Option[String]): String = {
    counter += 1
    val sffx = suffix match{
      case Some(s) => s
      case None => ""
    }
    lastSuffix = sffx
    return "tmp"+sffx+counter
  }
  def getLastVar: String = "tmp"+lastSuffix+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 with a suffix given as argument to make the temporary variable hold information of which expression it represents. Now at each evaluation of an AST node, 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 intermediate 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 node: 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.
The same procedure has been used for all other expressions and also for statements because they contain expression(s).

4. Possible Extensions

The aim of the project, to output C, has been reached. The C programs outputed can be compiled on gcc and running them will output the same result than if the original Tool was runned on the JVM.
However enhancements could be done to produce better program. Firstly, the problem we encountered about method chaining forced us to print intermediate variable for each expression. Even though it doesn't change the functionnality of the program, it makes the output more difficult to be read by humans. A new model for object-oriented in C could be conceived to make method chaining more straight-forward. The struct methods could not have to hold a pointer to the struct itself as first argument, and instead accessing a registry of all classes/struct produced to get the reference to their class and access the fields.
Another enhancement would be to include a garbage collector[2] to our program. In our implementation, instances of classe and array are allocated but never freed, thus making memory leaks. A simple garbage collector mechanic could be printed with the program to deallocate the pointers when reaching the end of the scope.

[2] ref necessaire peut łtre