Code generation - Mini Assignment Report

Prateek Kumar
Roll No. : CS15BTECH11031

Vaibhav Sinha
Roll No. CS15BTECH11034

October 16, 2017

1 Instroduction and Overall Structure

We create two new classes ClassInfo and ClassInfoTable similar to semantic assignment. This helps in saving and handy access to the information of the class. For ecah class ClassInfo stores the following information:

1. Parent
2. Depth in the tree
3. A list of the name of all the attributes
4. A map of all names of attributes to the actual attributes
5. A map of all names of methods to the actual methods
6. A map of the actual names of all methods to the names that appear in the IR.

Very few modifications have been made in Semantic.java. Apart from these the bulk of the code rests within Codegen.java.

2 Brief Overview of Codegen

2.1 Constant prints

Some boilerplate IR code needs to be printed in all cases and these are handled by simple function calls that print out this constant IR. These functions are involved in printing the IR corresponding to the functions of IO, Object and String and Header information printed at start and function declarations. Also they print the main function.

2.2 Printing the classes

First a graph is created of all the classes within the program. Upon this graph a breadth first traversal is done and simultaneously the classes are added to the classTable. Such a BFS traversal is necessary because we may not insert child classes into the classTable before their parents, because we need parents data for child. Along with the traversal we go on printing the class declarations.

2.3 Printing the individual class

For each class we begin by printing the constructor that initializes the attributes created in the class. After that we print the IR for each method of the class. The method printing is handled by printExpr function which generates the IR for each kind of expression.

3 Features of our Codegen

3.1 Class declaration

Each class is declared in IR as %class.classname = type . The class declaration is done keeping in mind inheritance. So first we declare Object Class as %class.Object = type i32, [1024 x i8] * . Note that i32 and [1024 x i8]* are needed to maintain the class name which will be used by type_name() and size the class (memory needed to be allocated to its object). For inheritance the child class always creates the first attribute as its parent. So the classes like Main, A and IO (in factorial.cl file) have first attribute as %class.Object. Apart from that, type consists of the attributes defined in the function. Observe that class C inherits from B and hence has %class.B as its first attribute. If a class has an object of other class as its attribute then it appears as a pointer to that class in the class declaration in IR. Int and Bool appear as i32, while string as [1024 x i8]*.

3.2 Handling static strings

A program typically has some strings that are used for printing to IO etc. These strings get globally defined in the IR, as
`@.str<n> = private unnamed_addr constant [<strlen> x i8] c"<str>\00", align 1`
Whenever used we need to bitcast the string to [1024 x i8]* by
`bitcast [<string_length> x i8]* @.str<n> to [1024 x i8]*`

3.3 Constructor

For each class a constructor is defined, which initializes the attributes with appropriate values. For Int, String and Bool if the initial values are not provided 0, false and "" are initialized. For Objects, either the objects are initialized as declared by new or if not then null is assigned. The constructor of the Main class

gets called by the main function (define i32 @main) which is the starting point of execution of an IR. It is called as : `call i32 @_ZN4Main8__cons__(%class.Main* %0)` Note that all classes constructors have `__cons__` in their names and return i32 0. Whenever an object is created its constructor gets called.

3.4 Methods

A sample declaration of method is as follows :
`define <ret_type> <fun_name>(%class.<class_name>* %self [, type %formal_name]*){...}`
A concrete example is :
`define i32 @_ZN1A4fact( %class.A* %self, i32 %n ){ }` The first argument in a method is always self pointer of its own class. After that the formals follow. The function name is declared as
`@_ZN<Class_name_length>Class_name<Function_name_length>Function_name.`
Each function begins with the `entry :` block.

3.5 Divide by zero error

In case of division we use sdiv. But to handle division by zero we must check if the denominator is 0. To achieve this we have an if condition. In case the denominator is zero we call exit with message defined by Abortdivby0 global variable. Otherwise we continue with normal program flow.

3.6 Dispatch to void error

When we call a function we create an if construct, in which we call exit with message defined by Abortdispvoid global variable upon a dispatch to void, which aborts the program.

3.7 Phi nodes required by IF condition

We have used phi nodes for getting values return values from body of if and else statement. It decides which value to take depending on the branch we have come from.

3.8 Handling specific expressions

- Object : We check if it is a formal or an attribute. If formal, then we only need to carefully load it otherwise if it is attribute, then we need to first get the element pointer and then load it.
- Complement : We return 1 - x, because this converts true to false and vice versa.
- Equal : Use icmp eq operator
- Less than equals : Use icmp sle operator

- Less than : Use icmp slt operator
- Negation : We return 0 - x, for x.
- Divide : Division handling is covered in Divide by Zero Error.
- Multiplication : Use mul operator.
- Subtraction : Use sub operator.
- Addition : Use addition operator.
- Is void : Simply icmp eq with null.
- New : First we allocate memory for the object using malloc with size as obtained from class table. Then we bitcast it to proper type. Then we call the constructor of the class.
- Block : Simply convert all the expressions within the block to IR.
- Loop : Create 3 new blocks : loop_cond, loop_body and loop_end . The condition expression of the loop goes to loop_cond which branches to either end or body, and the body expression of the loop goes to loop_body.
- Static Dispatch : First we must check for dispatch to void. Refer to its section for more details. Once it is confirmed that the object being called upon is proper, first create the string that will hold the parameters to the call. We need to use get element pointer in case an attribute is being passed. After that we simply use the method call as described earlier.
- Condition : To handle if-else construct we create if.then, if.else and if.end. Eah block handles its corresponding expressions. At the end we need to add the phi instruction as needed. Refer to its section for more details.
- Assign : Assignment can be done to an attribute or a formal. To handle attribute we need to have an additional get element pointer. We use store for assignment, but we the type being assigned to is watched as a parent class object can be assigned to a child classs. According to need we have to do bitcast.

4 Test cases

The testcase factorial.cl provided covers many cases, in particular :

- Classes and Inheritance
- Constructors
- Looping and If construct
- Nesting constructs with if

- PHI cases
- Methods and Static Dispatch
- Uses of IO methods
- Recursive methods
- Arithmetic instructions
- Comparisons
- Assign
- Block construct
- Objects
- And of course int and string constants

The following test cases handle the run time error cases :
Div by 0 : div0.cl
Static dispatch to void : dispatchvoid.cl

The provided test cases pretty much cover all the constructs asked to be converted to IR by the assignment.