

Compiler project - phase 5

Code generation

Reyhaneh Derafshi - 610396098

introduction:

Input: intermediate representation (IR) and symbol table

Output: semantically equivalent target code for a given a machine architecture

Compiler: translate IR to assembly code

Assembler: generate binary code

Linker: link the binary code (assembler and linker as part of the compiler tool chain)

Three main tasks:

1. **Instruction selection:** choosing appropriate target machine instructions to implement the IR statements
2. **Register allocation and assignment:** deciding what values to keep in which registers (interface with the hardware)
3. **Instruction ordering:** deciding in what order to schedule the execution of instructions

Code design:

overview:

At a high level, the program does the following steps:

1. Generate an inheritance graph of the existing classes. As the program has passed the semantic analyzer stage, we know that the graph is valid. We create the graph to get the inheritance information, to find which class is a parent of which class.
2. Add the structure of each class as a struct in the IR, and add the global string constants in the program to the IR. We also compute the

memory that needs to be allocated when a new object of a class is created.

3. Traverse the AST, for each class one by one. For each AST node, depending on the type, IR is generated and sent to the output.
4. Add the constructors for all the classes and the methods defined for the default classes in COOL, i.e., Object, IO, and String.

Preparing the inheritance graph

The execution of the code generator starts by visiting the root of the AST, which is the node for the program. Here, first, it prepares the inheritance graph of the program. As the program has already passed semantic checks, it is known that the inheritance graph will be valid. Classes are added to the graph in their appropriate positions, such that the child nodes of a class represent the classes that inherit from it, and the parent node represents the class it inherits from. The purpose of creating this inheritance graph is to have this inheritance information, so that when class types are defined as structs in IR, the members of the ancestors can be included in an indirect way in the struct of the children. The inheritance graph also provides a helper function to get the join of two types, which then determines the return type of a conditional statement.

Declaring String constants

The string constants used in the COOL program are declared globally in the IR. To facilitate this, the following has been done:

1. Modify the visit() method for string constants in the semantic analyzer so that each string constant is stored in a map.
2. After the inheritance graph is generated in the code generator, in the program node itself, values from this map are read and declarations are added to the output.
3. For facilitating input and output of integers and strings, for using in the calls to printf() and scanf(), for specifying error messages, etc. we

declare a few more strings directly added to the map. At the end of this, all the string constants have been declared in the resultant IR.

Generating structs for each class and calculating their memory requirements

Next, the compiler generates a struct declaration in LLVM IR corresponding to every class. This is done as follows:

1. First, it declares an a struct for Object, which has a single member of the `i8*` type. This member is used to store the name of the type of the class of any object created. Storing here is useful because every class inherits from Object. This class has no other members.
2. Then, the inheritance graph is traversed in a depth first fashion. For each class, except Int, String, and Bool, a struct declaration is created based on the members of the class. For Int, `i32` is used, for Bool, `i8` is used, and for String, `i8*` is used. For other class objects, a pointer to the class struct type is created. The first member is always an object of the current class' parent's struct. In COOL, every class other than Object has a parent, so this is valid.
3. In the case of Int, String, and Bool, we use the primitive types, hence no struct is created for them.
4. Here, while creating the struct, we also create a map that maps the each class name to a variable name, which then maps to the index in the struct for that name in that class. This is useful when calling `getelementptr` in the IR, as it easily provides the index given the name.

Now, we also need the sizes of each class type, so that memory can appropriately be allocated on the heap when a new class object is created using `new`. This is done in the following way:

1. In the program, iterate over each class. In the class, iterate over all the attributes. For each attribute, the memory needed by it, in bytes, is added.

2. The memory needed by `Int`, stored as `i32`, is 4 bytes, by `Bool`, stored as `i8`, is 1 byte, and by `String`, stored as `i8*`, is 8 bytes, as it is a pointer type. For all other class members, as their pointers are stored, 8 bytes is allocated. Memory is also allocated for the parent's members.
3. The memory requirements computed for each class is stored in a map that maps the class name to the memory needed. For the in-built classes, values are directly added to the map. This map helps in quickly accessing the amount needed when calling `malloc()`.

Traversing the AST

The compiler now traverses the AST to generate IR for each input program construct. The previous steps took place in the program node of the AST. From here, we start traversing each class node of the AST, and recursively visit each node's children using the visitor pattern. For each node, based on the type of the node, IR is generated. In this recursive manner, IR for all the code in the input COOL program is generated. The `visit()` methods return values or registers when appropriate that build up the IR. The following is broadly done at each node:

1. `no_expr_class`

This node just returns null. This is used only when new attributes are created, and in the node for attributes, a check for null is used to check if no assignment has been made along with the declaration.

2. `assign_class`

This first recursively visits the expression on the right hand side of the assign. If the types do not match, a bitcast is performed. If the left hand side is a method parameter, the pointer to it, allocated at the start of the method, is used for storing. If not, the appropriate attribute from the class is fetched using a `getelementptr`. The result is then stored using a `store`, and the right hand side result is returned.

3. `static_dispatch_class`

First, the caller is visited. Then, creating a conditional structure, a check is made for null on the caller. If it is null, an error message is displayed and `abort()` is called. In-built functions are then separately handled. Then, the name of the function to be called and its class name is found, the caller is bitcasted if necessary, and a call instruction to the dispatch is created. The return value of the call is returned.

4. `cond_class`:

Labels for three new basic blocks are created. Memory is allocated to store the result. The label for the condition is added. First, the visitor visits the predicate. The result is used to create a `br` instruction to jump to the appropriate block. Then, the label for the body is added and the visitor visits it. The result is stored in the memory allocated, and a jump to the end is created. The same is done for the else block. The result type is determined by computing the join of the types of the individual blocks. If necessary, a bitcast and a load is performed, and the result returned.

5. `loop_class`:

As with conditionals, three blocks are created for the loop. First, the label for the condition is added and the predicate visited. Then, a conditional break to either the body or the end is created. A label for the body is added, and the body is visited. This is followed by a break back to the condition block, to evaluate the condition again. The label for the end is finally created and null of the IR is returned.

6. `block_class`:

Each expression in the block is visited. The result of visiting the last expression is returned.

7. `new__class`

For a primitive type, as memory is already allocated, this just reassigns the value to the default value. For other types, using the memory information stored in the map, a call to `malloc` of C is created to allocate the required memory. Then, the result is appropriately bitcasted, the constructor of the type called, and the resultant bitcasted pointer returned.

8. `isvoid_class`

This visits the expression, and then performs an equality check with null of the IR. The result of this is returned.

9. `plus_class`, `sub_class`, `mul_class`

Each of the operands are visited. Then, the appropriate instruction is created in the IR and the result returned.

10. `divide_class`

This is similar to the previous case. Here, however, we also have a special check for if the division is being done by 0.

11. `Comp_class`, `lt_class`, `leq_class`, `eq_class`

This is very similar to `AST.add`. Here, the appropriate icmp instructions are called and a boolean type is returned.

12. `neg_class`

The expression is first visited. Then, the result is created by subtracting it from zero and is returned.

13. `object_class`

If the identifier is a method parameter, it is simply loaded and returned. If not, it must be an attribute, and then, a `getelementptr` is called to fetch it from the struct. It is then loaded and returned.

14. `Int_const_class`, `bool_const_class`

This returns the constant converted to a string.

15. `String_const_class`

This fetches the string from the global string constants stored using a `getelementptr`, and returns the result. The other nodes in the AST just return null because as per the assumptions, the constructs they represent would not be present in the input program and are not handled.

Adding the constructors and default methods

After all the IR for the input program is generated, the constructors for each class and "default" methods, i.e., methods in in-built classes, are defined. The constructors are defined as follows:

1. Starting from the root, the constructor for each class is generated. The traversal of classes is done in a depth first fashion using the inheritance graph.
2. For each class, a method is defined by generating a mangled name with the same name as the class name. It first creates a call in the IR for the constructor of its parent. Then, it traverses and visits each attribute node of the class, adding the IR as per the input given. The return type of the constructor is void.