CS3423 COMPILERS-II

# Code Generation

OM SITAPARA : CS16BTECH11036

HARSHIT PATEL : CS16BTECH11017

### Input - Annotated Syntax tree by Semantic Analyzer

### Output - LLVM-IR file.

Check for runtime exceptions as division by zero or dispatch on void.

Assumption - Input programs do not have any dispatch, let, typecase, branch.

# Design Decisions

## Inheritance Graph :

The Input program is our annotated AST. We traverse through the whole program and create a inheritance graph from it. Inheritance graph defines the structure by which different classes in a program inherit data from their super classes.

### Structure of Inheritance Graph :

Our inheritance graph is an ArrayList of graphNode.

We have defined a separate class for graphNode which provides our nodes with basic features like its parent, children list, index and its AST class.

Our Inheritance graph class contains several utilities like -

***classNameToIndex*** map for mapping each class name to an integer

Checking if our graph contains a main or not.

It also contains restricted types like - INT, BOOL, STRING and IO which cannot be used as class names.

***restrictedInheritanceType*** - BOOL, INT, STRING i.e. defining types which cannot be inherited from any class.

***classWithNoMethodType*** - BOOL, INT class which do not have predefined methods.

Inheritance graph cannot contain any cycle as there cannot be 2 classes such that they both inherit from each other. Hence we have DFS(depth-first search) for detecting cycles in our program(which is described later).

### Using our inheritance graph :

In Visitor.java we create an object of Inheritance Graph.

On creating its object its constructor sets its hasMain to false which is turned true only if we traverse a main class. The Basic Classes OBJECT, IO,STRING, INT and BOOL are added to our inheritance graph using the function ***addBasicClass()***.

The OBJECT class is the root of our inheritance graph which is added along with its predefined methods : abort, type_name and copy.

The IO class is added along with its methods : out_string, in_string, out_in, in_int.

STRING class with its methods : concat, substr, length

INT and BOOL class are also added.

While adding each class we map the class name to an integer for checking class name later on.

After the basic classes are added, we traverse through all the other classes, adding them to our inheritance graph using the function - ***addNewClass()*** . If there are no errors we add our class to inheritance graph and map it to an integer adding it to **classNametoIndex** map.

Then we traverse through all the classes and check if any of the classes are inheriting from restricted inheritance types by checking its parent.

The links between each class, its child and its parent are added by invoking ***connectGraph()*** function i.e. each class sets its parent class and adds its child to its list of children. Each class should posses a parent otherwise error message is generated.

## Code Structure and Overview of Classes :

We have tried to use the OOP approach heavily, to make the code more structured and understandable.

### Classes :

- ***Constants.java*** - Storing different types of constants present in COOL

- ***GlobalVariables.java*** - Contains global variables which are defined "public static" and can be accessed across the cool package

- ***GraphNode.java*** - Defines basic structure of our node of our inheritance Graph.

- ***InheritanceGraph.java*** - Our inheritance graph is declared here, and contains modules implementing basic features of our graph.

- ***VisitorCodgenImpl.java*** - Constructs our inheritance graph and implements visitor traversal mechanism by traversing over each node of our program.

- ***Codegen.Java*** - Starting point of the program

- ***UtilFunctionsIR.java*** - This file contains the functions required for generating the proper IR for the LLVM files.

# AST Traversal and Annotation:

For the traversal of the AST we use visitor mechanism. So what is visitor mechanism??

### Visitor Mechanism :

The visitor pattern is typically used to traverse a hierarchy of heterogeneous objects (inheriting a same abstract object) and dissociate the processing of these objects from the data within them. So here to implement the visitor mechanism we have created a class name Visitor which has has all the functions named traverse for each kind of node deriving from ASTnode. As we know that the expression class consists of different type of expression nodes we call method independently.

### Visitor Implementation :

Here the visitor mechanism is implemented in a DFS manner. First the semantic class creates a object of Visitor class and calls the traverse function on the AST.program. Now when this traverse is called the method of the parent node calls the traverse method on all its children node recursively so before traversing the parent all its children are traversed first and then the parent is traversed. Thus in this recursive fashion the complete AST is traversed. So how the code is generated ??

### CodeGeneration:

Now the traverse function is called on the program and it goes to expression by expression and generates the required code for that expression. We have handled the dispatch on null and division by 0 as a runtime errors.

How are runtime errors handled?

Every method on entry generates four blocks : entry, static.void, division.0, method.body. Now when a function is called if the dispatch is on null then we jump to static.void else we jump to normal branches. Same is for division by zero.

## AST.prog:

### Capturing Global Strings:

- We have a **GlobalStringsToIRMap** in which the string is mapped to the register name used for it. We have a static int **GlobalStringCounter** which counts the number of global strings present. For capturing the string constants instead of wasting a whole pass we modified the semantic analyzer to detect the string constants and put the strings to our map. Also we have inserted some default C strings as : %s, %d, \n etc for the printf statements.

### Generating class declaration:

- After the declaration of the Global string constants we declare the classes. the classes are declared as types where the first pointer is the pointer to its parent class and then the following attributes of the class listed. so for declaring the classes we travel inheritance graph in a dfs manner and generate the IR for each of the classes. during this traversal we maintain a map for the attributes in the each class mapped to the corresponding index in that class which is used for the for the get element pointer instructions. also while travelling these classes we maintain a map of the class name to its size which is useful for the malloc instructions for creating new objects.

### Generating IR for Constructors :

- While traversing for AST.program, after printing all the class declarations, we generate IR for each constructor. For generating IR, we start at the root node and generate IR for it and for other classes by traversing the inheritance graph in a DFS manner.
- For each constructor :
- Do not generate IR for Int, Bool, String i.e. primitive types.
- Call the constructor for parent class (Convert Instruction may be required for bitcasting it to desired type.)

- Traverse for all the features, if we encounter an attribute, we insert it into the scope table and for a method, we traverse on it.
- Then the children of this node are traversed in a DFS manner and corresponding IR for constructor is generated for them.

### Generating IR for Default Methods :

- After generating IR for constructors , we generate IR for default methods:
- Using the default C functions :
- Generate IR for default IO functions : out_int, out_string, in_int, out_int
- Generate IR for default string functions : length, concat, copy
- Generate IR for default object functions : type_name, abort
- Generate IR for C main method, void Dispatch error method and div by zero method.

### Generating the main function for IR:

- As we know the main function in C should return an integer but the main method in the main class of cool language can return anything so we need to generate the main method of C considering the return type of the main method of cool. so our main method returns 0 and has a proper call for the main method of the cool main class.

## AST.class :

- We traverse for each feature and if the feature is a attribute, we insert it into the scopeTable and if it is a method we traverse over it.

## AST.method:

- Method name is mangled formed by mangling the class name and function name.
- The first argument is a pointer of the class type i.e. this pointer, which is the pointer of the object calling the method.
- We traverse for each formals in the formal list and add it to list of formal parameters.

- An entry block is created and for each parameters present in the method, we allocate memory for them using alloca instructions. The allocated memory is on the stack.
- On traversing the method body, if the declared type does not match the return type for method.body, bitcast operation is performed.

## AST.formal:

- We add the formal name to a global formal list corresponding to method, so that it can be differentiated from attribute defined inside the method body. In the IR, we print as the typeid of formal followed by the formal name.

## AST.attr:

- We traverse for attributes when we generate IR for constructors.
- If assignment is done before, a store instruction is created to store in the attribute.
- If the type is primitive type, the default values for Int, Bool, String are stored, if no assignment is made.
- For other types, if no assignment is done, object pointer is used to store null.
- Bitcast may be performed if the types are not same as attribute type.

## AST.static_dispatch:

- Only static dispatches are handled.
    - First if the dispatch is for default methods, we generate IR for them.
- If not, we traverse the caller. If the caller is not of primitive type,  we generate a conditional branch instruction.
- If the caller is null, we call print_dispatch_on_void_error() and otherwise we jump to a new branch.
- In this branch, we lookup for the nearest class which contains method to be invoked.
- If the method is not present in the same class, bitcast is performed.
- Then we traverse through the actuals i.e and print the corresponding IR.
- FInally a invoke is done to the corresponding method.

## AST.assign:

- First we traverse the expression on the right side of assignment operator.
- If types mismatch occur either a new object is created using AST.new whose type is ROOT_TYPE or bitcast is performed , depending on if the expression type is primitive type or not.Then we determine the type of expression to be defined.
- If left side is a parameter defined inside the method, we assign value to it using the pointer stored, when we traversed the method.
- If not, then we use **getelementpointer** in our present class to get its pointer or we search for the expression name in the global formal list, and then a store operation is done for assigning the value.

## AST.new:

- If the new operation is done for primitive types, we return the default value (reassigns the value to the default value), as the memory is already allocated for them.
- Then a lookup is performed for the size of the object in the **mapClassSIze** and a malloc instruction to allocate the corresponding size.
- As the malloc returns a pointer to i8*, it is bitcasted to the type required.
- Then finally the constructor of the class is called whose object is to created.

## AST.cond:

- If the types of if.body and if.else do not match, **getJoinOf** is called to determine their lowest common ancestor .
- The predicate is traversed and according to it, we according to it we create a branch instruction.
- Label for cond.true is created and the if body is traversed, generating its corresponding IR and then branch to branch.normal.
- Similarly, Label for cond.false is created and the else body is traversed, generating its corresponding IR and then branch to branch.normal.

**AST.loop:**

- We add the loop condition, and the then we traverse for the loop predicate.
- A conditional break instruction is generated which determines the branch to loop.body or loop.exit.
- Then we traverse the loop body, generating its IR and finally a branch is created to the loop condition.

# LLVM IR Instructions :

### Load Instruction :

❖This instruction loads value from a memory location.

Prototype : <result> = load <ty> , <ty>* addr

E.g. %1 = load i32, i32* %0

### Store Instruction :

❖This instruction stores value to a memory location.

Prototype : store <ty> <defValue> , <ty>* addr

E.g. store i32 0, i32* %4

### Double Pointer Store Instruction :

❖This instruction stores value to a memory location, similiar to store instructions but the types here are non primitive types.

❖Useful when traversing for AST.attribute.

Prototype : store <ty>* <defValue> , <ty>** addr

E.g. store %class.Object* null, %class.Object** %5

## Binary Instruction :

❖ Operation on two operands of the same type

❖ Produce a result of the same type as operands

Prototype : <result> = add <ty> <op1>, <op2>

: <result> = add nsw <ty> <op1>, <op2>

nsw - no signed wrap

E.g. %2 = add nsw i64 %0, %1

## Conversion Instructions :

❖ Convert one type to another. E.g. bitcast, trunc, zext

Trunc :

❖ Truncate higher order bits

❖ Both types must be of integer types or vector of integers

❖ sizeof(ty) > sizeof(ty2)

Prototype : <result> = trunc <ty> <value> to <ty2>

E.g. %3 = trunc i8 %2 to i1

Zext:

❖ Zero extension of higher order bits

❖ Both types must be of integer types or vector of integers

❖ sizeof(ty) < sizeof(ty2)

Prototype : <result> = zext <ty> <value> to <ty2>

E.g. %0 = zext i32 %len to i64

Bitcast:

❖ Both operand & type must be non-aggregate first class types

❖ sizeof(value) == sizeof(ty2)

❖ Pointer can be bitcasted to pointer(of some other type) only

Prototype : <result> = bitcast <ty> <value> to <ty2>

E.g. %0 = bitcast %class.Main* %this to %class.Object*

## Branch Instructions :

❖This instruction jumps to a particular branch.

❖It is either conditional or unconditional.

❖Every basic block ends with a terminator instruction

Prototype : Unconditional : br label <dest>

Prototype : Conditional : br i1 <cond>, label <iftrue>, label <iffalse>

E.g : Unconditional : br label %method.body

E.g : Conditional : br i1 %4, label %static.void, label %branch.normal.1

## Void Call Instruction :

❖ Invoke void function like constructors.

Prototype : call void @ Name ( Argument List )

Useful for calling print_dispatch_on_void_error() and print_div_by_zero_err_msg() and invoking constructors of classes.

## Call Instruction :

❖ Invoking function

Prototype : <result> = call <ty> @ Name ( Argument List )

## Malloc Instruction :

❖ It allocates dynamic memory on the heap corresponding to the input bits.

❖ Returns a pointer

Prototype : <result> = call noalias i8* @malloc(i64 <input bits> )

E.g : %1 = call noalias i8* @malloc(i64 41)

Allocates 41 bits of memory.

## GEP(GetElementPtr) Instruction :

❖ Compute address offset of a field of an aggregate type

❖ Does not accesses memory

❖ First argument is the aggregate type

❖ Second argument is a pointer (vector of pointers) to the aggregate value

❖ Remaining are indices

Class attribute GEP : type(<ty>) is name of class.

Type name GEP : type(<ty>) is the ROOT Type.

Prototype : <result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, <ty> <idx>}*

E.g : %1 = getelementptr inbounds %class.Main, %class.Main* %this,  i32 0, i32 1

### Alloca Instruction :

❖ Allocates memory on the stack frame of the function

❖ Returns a pointer

❖ Automatically deallocated on return

Prototype : <result> = alloca [inalloca] <type> [, <ty> <NumElements>][, align <alignment>]

E.g : %0 = alloca i32, align 8

### Constructor Call Instruction :

❖ Invokes constructor of a type

E.g : call void @_CObject6_FObject6_(%class.Object* %0)

❖ Invokes object constructor

## Running the CodeGenerator:

Run the following commands:

- Make
- ./codegen filename.cl
- A file named filename.ll will be generated to run : clang filename.ll
- ./a.out

## Function not handled properly:

- ***Type_name*** : instead of printing the type of the object it returns the string "Type name function called"