Milestone 3

Motivation

We have chosen to implement Milestone 4 using low-level code, specifically, LLVM IR. More specifically, we plan to first convert Golite to a simpler custom IR before converting it to the LLVM IR using the LLVM C API.

We plan to use LLVM because as a backend, as it's used by several popular languages like Rust, Swift, C, C++, D, etc. It can compile to many different targets and offers optimization passed for free. LLVM IR is comparatively easy to work with, and due to its popularity it's well documented and standardized, meaning that if we get stuck somewhere we have documentation to look at to help us.

Also it has a proper API to generate it using objects, which is a lot nicer than trying to print a language like C. With it we can manipulate the IR more freely. Since we're working with the AST directly, things like names don't matter. If there's a conflict, LLVM will take care of generating new names when printing.

It has some disadvantages in that it is comparatively more difficult to implement than higher level languages like C/C++, but we wanted to take on this challenging task and in the process learn more about LLVM.

Using LLVM allows us to get experience with using a powerful framework and toolchain, that powers a vast number of compilers used in research

and enterprise.

Implementation

After type-checking, we run the `SemanticData` produced by our type-checker through our `IrConverter`. The `IrConverter` visits AST nodes and converts them to our custom node types (found in `golite.ir.node`) to facilitate conversion to LLVM IR. The `Program` class contains all the relevant information required for code generation: the package name, global variables, and functions (through which enclosed statements can be accessed). The conversion allows for complicated statements to be split up into multiple simple statements.

We have so far implemented the conversion for declarations and most of the simple statements (like `print`). Some of the expressions are converted, mostly the complicated ones (like `append` and struct equality). Most of the work left is the control flow statements, and unary and binary operators.

As an example of our custom IR, consider this simple Golite program:

```
package main

type Person struct {
    name string
    age int
}

var arr1 [16]Person
var arr2 [16]Person
```

```
func main() {
    println(arr1 != arr2)
}
```

It is converted to the follow custom IR:

```
package main

var arr1 [16]struct {name string; age int}
var arr2 [16]struct {name string; age int}

func main() {
    var arrLeft [16]struct {name string; age int}
    assign(arrLeft, arr1)
    var arrRight [16]struct {name string; age int}
    assign(arrRight, arr2)
    var arrEq bool
    assign(arrEq, true)
    var arrIdx int
    assign(arrIdx, 0)

startLoop:
    jump(endLoop, ((arrIdx >= 16) || !arrEq))
    var structLeft struct {name string; age int}
    assign(structLeft, arrLeft[arrIdx])
    var structRight struct {name string; age int}
    assign(structRight, arrRight[arrIdx])
    var structEq bool
    assign(structEq, true)
    jump(structNeq, !(structLeft.name ==$ structRight.name))
    jump(structNeq, !(structLeft.age == structRight.age))
    jump(endStructEq, true)

structNeq:
    assign(structEq, false)
```

```
endStructEq:
    assign(arrEq, structEq)
    assign(arrIdx, (arrIdx + 1))
    jump(startLoop, true)

endLoop:
    printBool(!arrEq)
    printString("\a")
    return
}

init {
    memset(arr1, 0)
    memset(arr2, 0)
    return
}
```

As you can see, a lot of the implicit semantics are made explicit (such as zero values). The equality operator is also expanded into a loop to compare the arrays of structs. The last function is a special one, which is called before `main`, and is used to initialize the global variables.

The `CodeGenerator` takes in the `Program` generated by the conversion phase and uses LLVM C API calls to create the object structure representing the IR. The remaining implementation work is for unary and binary expressions, as well as the jumps and labels.

Afterwards the IR is optimized with a few built-in simplification passes (such as memory to register), then compile to and object file.

Some of the functionality is implemented in a C runtime (such as appending slices), which is also compiled to an object file. To obtain an

executable, we simply need to link the two objects using a C compiler.

Test Programs

MultiAssignSwap.go

It's a good edge case for Code Generation. `a, b = b, a` should swap variables, which won't work unless you generate intermediate variables for the values.

switch_codegen.go

This checks if the generated code is able to handle the comparison condition in a case conditional. Many languages like C don't support these kind of case conditions by default, so it is an interesting case.

switch_codegen_booleans.go

This again checks if the generated code is able to handle two booleans in a case conditional. Many languages like C don't support these kind of case conditions by default, so it is an interesting case as well.

array_default.go

It's a good test case to check if an array is initialized during the codegen by default. We used a multi-dimensional float array to check this.

scope_var.go

This program is to ensure that the scopes are handled correctly by the codegen. The value of the variable outside the function scope should be replaced by the newly declared in-scope value.