Intermediate Code Generation

Sungbin Jo (2021-13630), Kijun Shin (2021-15391)

SwiftSnuPL IR implementation notes

Due to time constraints, we have implemented the IR only on the Swift port of the compiler.

The Swift `Generator` converts a parsed module with resolver information to the IR, which is basically two dictionaries; a symbol-to-instructions mapping that represents the TEXT section, and a symbol-to-strings mapping that represents the DATA section.

Honestly the IR is almost the same with the provided CppSnuPL with the biggest differences being the names and how the code is being shared. Most notable differences are from how we represent operands and memory-related instructions.

`Operand` represents 64-bit scalar values that can either be a constant, a register value, or a value on the stack; pointer operations like array accessing and assignment are handled by memory-related instructions directly. This is less of a design decision, but more of something was forced due to time constraints; we wanted a separate `Address` type that explicitly represents memory addresses. Unfortuately, we did not have the time to design it, and thankfully the SnuPL language is light in pointers; two specialized instructions for loading and storing data from the memory was enough to produce an IR.

Another point that stems from this difference is that the store, load instructions do not have memory access size in the instruction; we are planning for the memory layout of the array to have its size of the item at the head, so that the runtime can dynamically check the item size and invoke the appropriate instruction. We are planning to implement this in the assembly backend, but we might re-add the memory access size if we decide to not implement this.

The function call instruction also contains argument operands, which is also different from CppSnuPL where there is a separate instruction for passing arguments; this is so that the assembly backend doesn't need to care about the registers being overwritten while passing arguments.

All-in-all, there isn't really a lot to explain on the Swift IR implementation; most ideas are a straightforward port of the CppSnuPL TAC, optimized for implementation speed. A simple overview of each instructions are below; consult the code on the specific details on how they are implemented.

```
enum Operand {
    case constant(Int64) // a constant with the value
    case temporary(name: String) // a location for temporary scratch values
    case string(name: String) // a constant with the address of a string literal
    case symbol(Resolver.Symbol) // a location of the variable with the symbol
}

enum Instruction {
    case move(destination: Operand, source: Operand)
    case unary(op: UnaryOp, destination: Operand, source: Operand)
    case binary(op: BinaryOp, destination: Operand, source1: Operand, source2: Operand)
    case parameter(destination: Operand, index: Int)  // always 64-bit

    case jump(destination: String)
    case branch(destination: String, source1: Operand, source2: Operand)  // branch if equa
    case call(destination: Operand?, symbol: Resolver.Symbol, arguments: [Operand])
    case `return`(value: Operand?)

    case load(destination: Operand, symbol: Resolver.Symbol, indices: [Operand])
    case store(symbol: Resolver.Symbol, indices: [Operand], source: Operand)

    case label(name: String)
}
```