Design Document for Milestone 3

Lore, J., Lougheed D., Wang A.

March 27, 2019

Contents

This document is for explaining the design decisions we had to make whilst implementing the components for milestone 3.

1 JVM Bytecode for Code Generation

We decided on targeting JVM bytecode for our compiler, through the Krakatau bytecode assembler. Krakatau bytecode syntax is derived from Jasmin, but with a more modern codebase (written in Python) and some additional features.

1.1 Advantages

The primary advantages of targeting JVM bytecode are: portability, execution speed, and its focus on stack manipulation, as opposed to a higher level language, which aids in overcoming some of the common pain points of GoLite code generation:

1.1.1 Portability

The JVM has been ported to many common platforms, meaning code written in GoLite, when compiled with our compiler, will be able to run on any platform the JVM can run on.

1.1.2 Execution Speed

Although Java is often considered slow as opposed to ahead-of-time compiled languages such as C and C++ due to its garbage collection and non-native compiled code, most implementations of the JVM provide JIT compilation. By targeting JVM bytecode, we can take advantage of this, and our generated code will likely be faster than if we generated code in an interpreted language such as Python. Sometimes, the JVM can be faster than even ahead-of-time compiled programs, since run-time information is available for optimization purposes.

1.1.3 Stack Based/Low Level

The fact that JVM bytecode is fairly low level gives us lots of granular control for changing the behavior of constructs, especially when dealing with odd `GoLite` / `GoLang` behavior that doesn't map perfectly to more common programming languages.

Operating on a stack makes some of the operations mentioned in class as 'difficult to implement' surprisingly easy. In particular, swapping variables (e.g. `a, b = b, a`) is fairly straightforward. The right-hand side must be evaluated before assignment, which can be done by pushing and evaluating all RHS terms, left to right, onto the stack; followed by popping each value in turn and loading them into corresponding locals. Compared to temporary variable or register allocation, this is a very natural way to implement this construct.

1.2 Disadvantages

The main disadvantages of generating JVM bytecode are its low-level semantics and its (in general) slightly slow speeds versus an ahead-of-time compiled language.

The low level of JVM bytecode is particularly frustrating to deal with when it comes to types that are not representable by an integer or floating point number and are thus classes which must be instantiated; in the JVM, this includes strings and ~struct~s. This means that operations such as comparisons and string concatenations go from being a few bytecode instructions to signficantly longer patterns.

2 Semantics

2.1 Scoping Rules

2.1.1 Go Semantics

In `GoLite`, new scopes are opened for block statements, `for` loops, `if` / `else` statements and function declarations (for the parameters and the function body). A new scope separates identifiers (which are associated with type maps, variables, functions and the constants `true` and `false`) from the other scopes' identifiers. Whenever we refer to an identifier, it references the identifier declared in the closest scope.

There is nothing very special about scoping in `GoLite`. The main notable thing is that something like `var a = a` will refer to `a` in a previous scope, not the current `a` that was just declared, unlike languages like `C`. On the other hand, recursive types such as `type b b` fail as expected, and do not reference a type from higher scopes.

2.1.2 Mapping Strategy

JVM bytecode only has "scoping" for `methods`, as they have their own locals and stack. Block statements do not exist per se, and statements except called method bodies are scope-less. In higher level languages, we could just append the scope depth to all identifier names to keep them unique, also eliminating the need for separate scopes, as we already typecheck the correct use of identifiers. In our case, we have scoped identifiers in our newly generated typechecked AST which we convert to offsets (for locals).

These offsets can be used in our intermediate representation, where the offsets are unique for each variable declaration. Variables with the same scoped identifier will be given the same offset, and we can optimize our stack limit by reusing offsets when two variables can never occur at the same time due to branching.

2.2 Switch Statements

2.2.1 Go Semantics

In `GoLite`, `switch` statements consist of an optional simple statement, an optional expression and a list of case statements. Case statements are either a case with a non-empty list of expressions, or a default case with no additional expression. Each case statement also contains a block statement, containing code to execute upon match. This makes them structurally different when compared to Java or `C` / `C++`, where:

- Simple statements don't exist.
- Expressions aren't optional.
- Case statements don't match on a list of expressions.

The simple statement is executed before the `case` checking. After that, the optional expression is compared with each `case` statement, evaluating and comparing expression lists from left to right. The first match enters that case's body, automatically breaking at the end of it. This makes cases significantly different semantically:

- Cases automatically break.
- Each `case` or `default` block defines its own scope for declarations.
- Case statement expressions do not need to be a constant expression.

2.2.2 Mapping Strategy

For the structural differences:

- Simple statements can be modeled and executed as the first statement in the new "scope".
- Missing expressions can be converted to the constant literal 'true'.
- For a list of expressions that is of length greater than one, we can compare each element from the list one at a time, duplicating the value we're comparing to before each comparison (as otherwise we'll lose it during the stack operation). In other terms, the value we compare to during each `case` block is stored on the stack until the `switch` statement is done. Semantically:

- After `case` expression comparisons, we will either jump to the case body or keep going to the next `case` comparison or `default` block.
- To automatically break, for each case statement, we add a `goto` to a label at the end of the switch statement.
- `default` statement jumps should be placed after all jumps to case bodies as the fall-through case, when no other jumps are followed.
- Simulating new scopes is easy because of how our scoping works; the variable names will already be resolved to their correct local's index.
- Expressions in `case` blocks not being constants does not matter too much for us, as we will compare each expression normally (we are simulating switch statements using `goto` and comparisons, and aren't limited by any language-native `switch` statement definitions).

2.3 Assignments

2.3.1 Go Semantics

In `GoLite`, assignments are either an assignment operator with a single LHS expression and a RHS expression, or two non-empty expression lists (LHS and RHS) of equal length. This makes them structurally different (for the two non-empty list case) from 'classic' assignments, which typically only allow one l-value. This structural difference is a lot more significant than it seems at first glance, because the assignments are done in a "simultaneous" way, that is `a, b = b, a` will swap the values of `a` and `b`. If the assignments were done sequentially, `a` and `b` would be the original value of `b` and wouldn't be swapped.

2.3.2 Mapping Strategy

There are two tricky things about assignments:

- Assignment operators. We cannot just convert `e += e2` to `e = e + e2`, where `e` is an expression, because `e` might contain a function call with side-effects, which we do not want to call twice (note that in some cases, the assignment operator has an equivalent bytecode instruction, i.e. incrementing and decrementing using `iinc`. However, we generalize in this discussion as most operators do not have an equivalent instruction to operate and assign at the same time). There are thus several cases for `e`:

- `e` is just an identifier. Then, we can just convert `e += e2` to `e = e + e2`, as there will be no side effects.
- `e` is a selector. If `e` is an addressable selector, then it is not operating on the direct/anonymous return value of a function call and so re-evaluating `e` will not produce any side effects. Thus we can do `e = e + e2` again.
- `e` is an index, say `e3[e4]`. In this case, `e3` can be an anonymous `slice` from a function return, and `e4` could also be an anonymous `int` from a function return. In order to avoid duplicate side effects, we should resolve `e3[e4]`, including any function calls, to some base addressable expression, storing the result on the stack. Then, we can operate on the stack, adding `e2` and assigning the result to whatever the stack value references.
- The other cases for `e` are not `l-values`, and shouldn't happen in the type-checked AST.

- Assignment of multiple expressions. As mentioned earlier, we cannot do the assignments sequentially. Thus, we should evaluate the entire RHS, pushing each result onto the stack and then assigning each stack element one by one to their respective LHS expression l-value. This way, `a, b = b, a` will not overwrite or interfere with any values used on the RHS. This is one of the advantages of using a stack-based language, as values on the stack implicitly act like temporary variables, so we don't need to allocate other temporary resources for simultaneous assignment.

3 Currently Implemented: Intermediate Representation

The main feature that was worked on during this milestone was the creation of our intermediate representation, and the conversion of the typechecked AST to said IR.

We decided on creating an IR for bytecode in order to make conversion easier from the AST, and enforce some degree of correctness using Haskell's type system. The IR is also stack-based, and to a large extent is functionally identical to JVM bytecode, modeled in Haskell. We represent classes and methods as Haskell records. Method bodies are a list of what we call `IRItems`, which are either stack instructions or labels.

Available stack instructions, as of this milestone, include `Add` and other binary operations, `Dup`, `Load` and `Store`, `InvokeVirtual/InvokeSpecial`, some integer-specific operations, and `Return`. Instead of specifically representing equivalents of `iadd/fadd`, `iload/aload/...`, etc., we define an `IRType` data type which can either be a bytecode primitive (integer or float) or an object reference. In this way, the IR definition is kept short and similar instructions can be combined into a single Haskell constructor model. Other Haskell types are used to model method/class specifiers, Jasmin-style parameter and return types, and loadable values (ints, floats, and strings).

Eventually, our goal is to then convert this IR into Krakatau bytecode syntax, which should be very straightforward given that the IR is so close to bytecode already.