6.035 Project 2

Kyle Miller, Alec Thomson, Patrick Hulin, Steven Valdez

March 20, 2012

1 Overview

The Purpose of this project was to extend the semantic checker produced at the end of the last project to actually generate correct x86 assembly code.

Our Code Generator follows a four step process. The first step is to translate the Hybrid AST and Symbol Table produced as the output of the previous project into a Mid-Level IR that represents the code as a graph of Basic Blocks. The second step is to translate the mid IR into a Low-Level IR that translates the higher level instructions used in the mid IR to low level assembly instructions. The low IR initially makes use of a number of "Symbolic Registers", so the next step of code generation is to use a basic Register Allocator to replace these symbolic registers with real registers and memory locations. Finally, the low IR with all symbolic registers allocated is translated directly into assembly language.

The rest of this document outlines each of these steps in detail. The mid level IR is described in Section 4. The low level IR is described in Section 5. The register allocator is described in Section 6. The assembly generator is described in Section 7.

2 Division of Work

The work was divided as follows. Kyle Miller designed and wrote the mid IR and the low IR and wrote all of the translation functions between them. Kyle also implemented several optimizations. Alec Thomson wrote an initial assembly template that provided some skeleton code for eventual assembly generation. Alec also wrote the simple register allocator. Patrick Hulin and Steven Valdez wrote the code to translate the low ir directly into assembly code. Each member of the team spent a considerable amount of time debugging the final product.

3 Assumptions about Spec

The following were assumptions we made about the Decaf spec while we wrote the code generator:

- The Array index expression is evaluated before the assignment expression.
- Binary operators are evaluated left to right.
- Method arguments are evaluated right to left.

The first two assumptions are fairly arbitrary, but the third was chosen in this way because it is easier generate assembly instructions in right-to-left order because stack-based arguments are stored in pop-friendly order.

4 Mid IR

We decided on having two levels of intermediate representations between the AST and the assembly code. The first is the Mid IR, which is the subject of this section, and the second is the Low IR, which is the subject of the next. Each of these IRs are graph structures where each vertex is a sequence of instructions culminating in some kind of jump, and each edge represents the connection from the jumper to the jumpee. The distinction between the two is that the Mid IR deals with symbolic variables where the Low IR deals with registers only, and also the Mid IR has no idea about the `d(r,r,s)`-style addressing schemes, where the Low IR does. We took the instruction-level design of our IR from Muchnick.

The conversion from the AST to the Mid IR was accomplished by a continuation-passing-style algorithm where each continuation was the ID of a basic block to jump to in the circumstances of 1) success, 2) breaking from a loop, and 3) continuing the next iteration of a loop. Inside expressions, the continuation was limited to only success, but an additional continuation, the temporary variable in which to store the result of a subexpression, was also passed around.

The continuation style made it straightforward to implement all of the control structures. The `if` statement gives a taste of the general method:

```
statementToMidIR env s c b (HIfSt _ pos expr cons malt)
    = do cons' <- statementToMidIR env s c b cons
         alt' <- case malt of
                   Just alt -> statementToMidIR env s c b alt
                   Nothing -> return s
         tvar <- genTmpVar
         block <- newBlock [] (IRTest (OperVar tvar)) pos
         addEdge block True cons'
         addEdge block False alt'
         e <- expressionToMidIR env block tvar expr
         return e
```

where each statement conversion has the type

```
statementToMidIR :: IREnv
                 -> Int -- ^ BasicBlock on success
                 -> Int -- ^ BasicBlock on continue
                 -> Int -- ^ BasicBlock on break
                 -> HStatement a -> State MidIRState Int
```

Remarkably, `break` and `continue` have rather simple definitions:

```
statementToMidIR env s c b (HBreakSt _ pos)
    = return b
statementToMidIR env s c b (HContinueSt _ pos)
    = return c
```

To remove the need to maintain lexical environments, we did alpha renaming of the variables in functions so that variables have the same name if and only if they represent the same variable (i.e., are in the same lexical scope).

No effort was made to generate reasonable Mid IR code output since we are assuming we'll be able to remove the junk in later optimization passes.

We did make a minor effort toward optimizing the IR graph. We made a small graph rewrite algoritm framework to be able to convert an IR graph into normal form where each basic block is as big as possible. This algoirthm gave unreachable block removal for free.

5 Low IR

The Low IR is similar to the Mid IR except it has a better idea of the x86 architecture including its addressing schemes. We let the Low IR use symbolic registers to simulate a machine with infinitely many registrs so that we don't need to deal with the hard truth that there are only 16 registers on our CPU.

The translation from the Mid IR involved keeping track of a table of variable name to symbolic register. Each basic block is converted one at a time, statement by statement inside of them.

We tried doing some basic optimizations at this stage involving converting the Low IR to a tree-based low ir (where statements in the basic blocks are tree structures, and converting back using tree rewrite rules. This produces nice-looking low ir code which has optimized out most of the spurious register created by the previous steps. However, we have disabled this feature for now since it involved too much untested code, and it produced code which failed on `12-huge.pdf` (however, last we checked, each of the other tests worked fine). Tho optimizer can be enabled by passing in the `--opt` argument.

The intermediate representations each emit `graphviz` code which can be translated into a graphical representation by `dot -Tpdf -o`*`outfile infile`*. We would include such images in this document to show them off, but but they are generally much too large to make any sense of from a distance.

6 Register Allocation

All of the symbolic registers from the Low IR. The allocator works by maintaining state through the use of the State Monad to keep track of where previously encountered symbolic registers are located on the stack. When it encounters an unallocated symbolic register, it gives that register the next location on the stack and adds that mapping to its state. It subsequently updates the next location on the stack as well.

The register allocator also generates additional code to be inserted into the low ir to allow for the loading and storing of memory locations that correspond to the symbolic registers. At the moment, this code is very basic and very inefficient. Individual values are often shuffled through many more registers and memory locations than is necessary. Despite this inefficiency, the register allocator is useful because of its simplicity. The framework also exists now to produce a more complex and more efficient register allocator in the future.

Problems that arose while implementing the register allocator primarily consisted of incorrectly mapping symbolic registers to temporary registers during loads and stores. Additional code needed

to be inserted to account for indirect loads and stores of symbolic registers. All of this code is documented and included in `RegisterAllocator.hs`

7 Assembly Generation

The assembly generation was mainly accomplished via a bunch of methods which translated the Low IR to lists of strings of instructions. These methods were then combined to build a bigger part of the overall representation. While we probably caould have designed a better mechanism to make the type system check more of our code's safety, most of the errors we made at this stage were far beyond the type system's abilities. With our design decisions, the code itself was very straightforward to write, but the debugging was a total pain, probably taking 70% of the time. It's not clear that we could have avoided this. Most of the problems were due to an insufficient understanding of or incorrect implementation of the calling convention.