Assignment 8: Primitive Inlining and Callsite Caches

Patrick S. Li

November 12, 2015

1 Introduction and Motivation

The largest source of inefficiency in our current system is the time required to perform a method/slot lookup in the implementation of the CALL_SLOT, GET_SLOT, and SET_SLOT bytecodes. In general, the lookup is required because we do not know the type of the receiver object until runtime. However, for typical programs, most callsites will encounter very few different types of receiver objects. In fact, for a large number of callsites, only one type of receiver object is ever encountered. We will take advantage of this temporal locality and cache the result of the method lookup to avoid the expensive general lookup procedure.

Additionally, because operations on the primitive types are so prevalent, possible primitive operations are handled specially. We will speculatively inline the primitive operation with a test at the beginning to determine whether or not the receiver object is a primitive type.

2 Primitive Inlining

For the following method calls:

1. x.add(y)

2. x.sub(y)
3. x.mul(y)
4. x.div(y)
5. x.mod(y)
6. x.eq(y)
7. x.ne(y)
8. x.le(y)
9. x.lt(y)
10. x.ge(y)
11. x.gt(y)
12. x.get(i)
13. x.set(i, v)
14. x.length()

assume that x is either an integer, or array, appropriately. Instead of generating the code to trap to C for performing the method lookup, generate the instructions needed to directly perform the operation. Insert an additional receiver type test at the beginning of the code that will perform the general method lookup if the assumptions are invalid.

To generate the code for directly performing the primitive operations you will need the following additional assembly instructions.

- cqo
- idivq
- imulq
- sete
- setne
- setle
- setl

- setge
- setg

3 Callsite Caches

To cache the results of method lookups, we will reserve two words in the instruction stream in the generated code for CALL_SLOT bytecode. The first word will store the address of the method, and the second word will store the type of the receiver object. If the stored type is -1, then the cache is empty. Otherwise, the method address corresponding to a receiver object of the stored type is in the cache, and we can jump to it directly. If the cache is empty, or if the actual receiver object type is different from the stored type, then we must perform the general lookup procedure.

The general lookup procedure must now also update the cache appropriately. Be especially wary of Feeny's inheritance model when deciding when to update or invalidate the cache.

The following code outlines the structure of the CALL_SLOT bytecode. Note the use of the `.quad` pseudo-instruction to reserve space for the cache.

```
goto general_call_slot if stored type != receiver type
   call stored method address
      with return address = end_call
general_call_slot:
   perform general call
      with return address = end_call
.quad -1 //Stored method address
.quad -1 //Stored receiver type
end_call:
```

4 Slot Caches

The implementation of the SET_SLOT and GET_SLOT bycodes are similar to the CALL_SLOT bytecodes except that instead of caching the method address, we cache the slot index. Again, be especially wary of Feeny's inheritance model when deciding when to update or invalidate the cache.

5 Harness Infrastructure

You will be expected to execute Feeny programs from a given AST, and will be using the harness from the bytecode compiler assignment.

The function

```
Program* compile (ScopeStmt* stmt)
```

in the file `src/compiler.c` will be the entry point of your bytecode compiler, and will be called with the AST datastructure.

As in the last assignment

```
void interpret_bc (Program* p)
```

in the file `src/vm.c` will be the entry point of your bytecode interpreter and will be called with the result of your bytecode compiler.

Your assembly instructions are expected to reside in the file `src/vm.s`.

5.1 Test Harness

The provided bash script `run_tests` will parse the test programs in the `test` directory into ASTs and run your bytecode compiler and interpreter on the result. To run it, type:

```
./run_tests compiler.c vm.c vm.s
```

at the terminal. For each test program, e.g. `cplx.feeny`, the output of your bytecode interpreter will be stored in `output/cplx.out`. The default implementations of `compile` and `interpret_bc` do nothing except print out the AST and bytecode IR respectively.

Please ensure that your implementation can be driven correctly by `run_tests`. Your assignment will be graded using this script.

5.2 Compiling and Running Manually

To compile and run your implementation manually, you may also follow these steps. Compile your interpreter by typing:

```
gcc -std=gnu99 src/cfeeny.c src/utils.c src/ast.c
   src/bytecode.c src/compiler.c src/vm.c src/vm.s
   -o cfeeny -Wno-int-to-void-pointer-cast
```

at the terminal. This will create the `cfeeny` executable.

Next, you have to run the supplied parser which will read in the source text for a Feeny program and dump it to a binary AST file.

```
./parser -i tests/cplx.feeny -oast output/cplx.ast
```

Finally, call the `cfeeny` executable with the binary AST file as its argument to start the interpreter.

```
./cfeeny output/cplx.ast
```

6 Report

Implement your bytecode compiler, interpreter, and assembly snippets and place it in the `src` directory, naming it `compiler_xx.c`, `vm_xx.c`, and `vm_xx.s`, where `xx` is replaced with your initials. Then ensure that it can be properly driven by the testing harness by typing:

```
./run_tests compiler_xx.c vm_xx.c vm_xx.s
```

Include your implementation of Towers of Hanoi, Stacks, and More Towers of Hanoi from exercise 1, and ensure that your compiler and interpreter works correctly for those as well.

1. (100 points) **Compiler Statistics**: Create a table in your report that measure and calculate the following statistics for the test programs: `bsearch.feeny`, `inheritance.feeny`, `cplx.feeny`, `lists.feeny`, `vector.feeny`, `fibonacci.feeny`, `sudoku.feeny`, `sudoku2.feeny`, `hanoi.feeny`, `stacks.feeny`, `morehanoi.feeny`.
   (a) Total amount of time (in milliseconds) for `interpret_bc` to run and return.
   (b) Reduction in total time spent in `interpret_bc` relative to before implementing the optimizations (new/old).

(c) Total number of times a general method or slot lookup is performed during execution.

7 Deliverables

Students may work in pairs or alone. Please submit your answers to section 6 as *report_XX.pdf*, and your programs as *compiler_XX.c*, *vm_XX.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign8_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny8]` in the subject header.