Assignment 6: Tagged Primitives Optimization

Patrick S. Li

October 15, 2015

1 Motivation

From your previous assignments, you will have noticed that simple integers comprise the majority of memory that is allocated during execution of a program. This is especially shameful when we consider how much memory is actually wasted on a value that should theoretically only require 32 bits. To represent an integer, we use a 64-bit pointer to point to a 128-bit heap allocated structure, where the first 64-bits is used for the tag word, and the actual integral value is stored in the latter 32-bits of the second word. Thus in total, we are occupying six times more memory than we should. Even worse than the memory footprint is the tremendous strain this places upon the garbage collector.

In this assignment, you will implement a common optimization known as "tagged primitives" that will reduce the memory footprint of an integer to a single 64-bit word, and also eliminate the strain placed upon the garbage collector due to primitive values.

The algorithm is motivated by asking why *can't* we directly represent our integer values using their bit representation? Why do we *have* to represent them through an extra pointer indirection to a heap allocated object? The answer lies in the fact that Feeny is a *dynamically* typed language, and thus for any given value, it needs to be able to *dynamically* determine what type of object it is. Is it an integer, or an array, or an object of some class?

Currently this operation is simple. *Every* value is represented using a pointer to a heap allocated structure, and the first word in that structure is an

integer describing the type of the object. Thus the *only* reason why we heap allocate all of our values is to support this core operation: given a value, dynamically determine its type. If we directly represented integers using their bit representation, then there would be no way to distinguish between an integer and a pointer to an object.

We will see that there is a much cheaper representation of values that still preserves this core operation.

2 Algorithm

The algorithm follows from the simple observation that all of our heap allocated structures have sizes that are integer multiples of 64 bits. Thus if our heap starts on a 64 bit aligned boundary (which `malloc` already guarantees), then any pointer to a heap structure will necessarily end with the bits 000 in its least significant bits.

In this optimization, we will take advantage of these three bits of spare memory, and use it to partially store the type information of a value. We are only partially storing the type information because there may be many classes defined in a Feeny program, and three bits is not enough to uniquely encode each.

Here is the strategy we will use. We will represent values of the primitive type Null as the value 2. We will represent Integer values as $8v$, where v is the numerical value of the integer. And finally we will represent objects as $p + 1$, where p is a 64-bit pointer that points to a heap-allocated structure. This representation is called the "tagged" representation.

3 Algorithm Details

The following things need to be updated in your virtual machine:

1. Creation of Literals: Literal objects are now no longer allocated on the heap. Thus update the code to directly represent them using the tagged representations given above.

2. Dynamic Dispatch: Instructions which require a dynamic dispatch upon the type of an object, such as CALL_SLOT, are now more complicated. Determing the type of an object is now a two step process. First determine its partial type by checking the least significant three bits of the value. If it is an object, then retrieve its tag word from the head of its heap-allocated structure and determine its type from the tag word.

3. Branches: The predicate test for branches have to be changed to reflect the new representation of the Null object.

4. Arithmetic Operations: The arithmetic operations need to be changed to reflect the new representations of the integers. See section 4 for details.

5. Load/Store Operations: Load and store operations to slots within an object need to be changed to reflect the 1-byte bias inherent in the value representation.

6. Garbage Collector: The `get_post_gc_ptr` function has to be changed to reflect the new representation of objects. If the object is a primitive type then its post garbage-collection value remains the same, and can be returned directly. No broken heart needs to be set in this case either.

4 Arithmetic Details

The naive method of implementing the arithmetic operators is to convert the integers into their standard bit representations, perform the operation, and then convert the result back into its tagged representation.

However, we have chosen a particularly clever value representation that permits us to perform arithmetic operations directly on the tagged representations, thus eliding the required conversion operations.

Here is an example derivation of the addition operation on tagged representations. Notationally we will use $f(x)$ to represent the tagged representation of the integer value x. As mentioned in section 2, our representation of integers is:

$$f(x) = 8x$$

Our goal, for the addition operation, is to determine the value of $f(x + y)$ given the tagged arguments $f(x)$ and $f(y)$. It is straightforward to see that

$$f(x + y) = 8(x + y) = 8x + 8y = f(x) + f(y)$$

. Thus we can perform addition simply by directly adding the tagged representations! Derive the rest of the arithmetic operators in a similar fashion.

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.

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
```

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=c99 src/cfeeny.c src/utils.c src/ast.c
   src/bytecode.c src/compiler.c src/vm.c
   -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 and interpreter (with GC and tagged primitives) and place it in the `src` directory, naming it `compiler_xx.c` and `vm_xx.c`, 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
```

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. (80 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) Total amount of time (in milliseconds) spent in the `garbage_collector` function.

(c) Total number of bytes allocated by `halloc`.

(d) Reduction in total number of bytes allocated relative to before implementing the tagged primitives optimization (new/old).

(e) Reduction in total time spent in the garbage collector relative to before implementing the tagged primitives optimization (new/old).

(f) Reduction in total time spent in `interpret_bc` relative to before implementing the tagged primitives optimization (new/old).

2. (10 points) **Multiply Operation**: Feeny's specification currently mandates 32 bit integers, but you may have noticed that on our 64 bit system, we can easily support up to 61 bit integers. Suppose we make this extension to the language. In the implementation of the multiply operator you might have noticed that you have two implementation choices:

$$f(x \cdot y) = f(x) \cdot f^{-1}(f(y))$$

or

$$f(x \cdot y) = f^{-1}(f(x) \cdot f(y))$$

. Which one is preferable and why?

3. (10 points) **Comparison Operations**: Suppose I give you a function $g(a \text{ op } b)$ where a and b are integers, and op can be any of the comparison operators. g returns 1 when the predicate is true, and returns 0 otherwise. How can you utilize this function to efficiently compute the result of a comparison operation on tagged representations? (While not easily accessible from C, g effectively models a conditional bit-set assembly instruction available on many processors.)

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 *assign5_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny6]` in the subject header.