Benchmarks

"There are three kinds of lies: lies, damned lies, and statistics." Benjamin Disraeli, according to Mark Twain (likely a misattribution).

"In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks.

William Gardiner Pritchard (1942–1994), a mathematician and experimentalist *extraordinaire*, explained to me in 1976 the two things needed to design an experiment:

(1) You must figure out very precisely what it is you want to measure, and
(2) You must figure out very precisely how to measure it.

And benchmarks *are* experiments. So in this short note I'd like to examine a few benchmarks for the Scheme programming language in the context of Bill's principles.

The benchmarks are taken from the web site:
`https://github.com/ecraven/r7rs-benchmarks`
commit f0ebd3ca4899ad932b7ebbf108dd80f801906f81, Sep 13 22:47:07 2017 +0200.
We'll call these the "Ecraven" benchmarks.

In turn, these programs were adapted from the Larceny project:
`http://www.larcenists.org/benchmarksAboutR7.html`.
The Larceny benchmarks started as R6RS Scheme benchmarks, then were modified to run as R7RS Scheme programs.

Some of the Larceny benchmarks originated in the Gambit benchmark suite:
`https://github.com/gambit/gambit/tree/master/bench`.

We'll run the programs with Gambit Scheme:
`https://github.com/gambit/gambit`
commit d6238d5db039f25abd05e1a26f47aa857be56c4b, Nov 12 20:45:28 2017 -0500.
Gambit was configured with arguments
"'CC=gcc -march=native -D___CAN_IMPORT_CLIB_DYNAMICALLY' '–enable-single-host' '–enable-multiple-versions' '–enable-shared"' and the gcc version was "7.2.0 (Ubuntu 7.2.0-8ubuntu3)".

Here's the TL;DR version of this note: I recommend changing the R7RS benchmarks as follows:

(1) For `maze`, use native implementation versions of `bitwise-not` and `bitwise-and`, or move definitions of these procedures to the prelude for each scheme.

If in the prelude, employ definition of `bitwise-and` that uses built-in R7RS functions `quotient` and `modulo`, not the new defintions of `div` and `mod`.

(2) In `matrix`, `paraffins`, `compiler`, and `fft`, textually replace uses of `div` and `mod` with the R7RS built-in functions `quotient` and `mod`.

(3) In `quicksort`, replace `(vector-map values v)` with `(vector-copy v)`, which is a built-in R7RS procedure.

Every benchmark suite is run with a number of assumptions, which I'll try to make explicit here:

(1) In R5RS Scheme, one can change the definitions of built-in procedures (`+`, `list`, etc.). These benchmarks are run assuming that this doesn't happen.

(2) We also assume that the benchmarks don't redefine Gambit-specific library functions.

(3) Gambit Scheme supports separate compilation and linking of modules, but we assume that any global procedure whose definition is not modified in its module of origin is never modified in another module.

(4) Minimal error checking is done, i.e., we assume that the programs are correct Scheme.

(5) Arithmetic operations are "generic", so for subraction, for example, we use the generic `-` (which is valid for integer arithmetic of any size, floating-point arithmetic, rational arithmetic, complex arithmetic, or any combination of these) and not the fixnum-specific `fx-` or the flonum-specific `fl-`.

Since we're going to be discussing several versions of Scheme (R5RS plus extensions, R6RS, and R7RS (small)), various implementations, and various programs we should try to answer Bill Pritchard's first question clearly:

(1) The purpose of a benchmark is to compare *implementations* when running a program that exercises *specific, known language features*.

The Benchmarks

When we speak of R7RS Scheme, we mean the R7RS Scheme (small) standard.

Maze. `maze` began as a Gambit benchmark for R5RS Scheme with bitwise integer operations as extensions.

The Larceny team then modified `maze.scm` to run on R6RS Scheme as follows:

(1) Instead of importing the R6RS library `(rnrs r5rs (6))` to supply the `quotient` and `modulo` procedure, these were replaced by the R6RS standard procedures `div` and `mod`. These R6RS procedures do not have the same semantics for all arguments as the R5RS procedures they replaced, but for the

arguments encountered in `maze.scm` (all arguments are exact nonnegative integers), the results are the same.

(2) The `(rnrs arithmetic bitwise)` R6RS library was imported to provide `bitwise-not` and `bitwise-and`.

That gave a valid R6RS program using standard R6RS procedures.

It appears that the Larceny team then converted the R6RS version of `maze.scm` to run under R7RS Scheme.

(1) As bitwise operations are not defined in R7RS (small), a short and relatively fast definition of `bitwise-not` was given as

```
(define (bitwise-not x)
  (- (- x) 1))
```

A straightforward one-bit-at-a-time implementation of `bitwise-and` was given as

```
(define (bitwise-and x y)
  (cond ((= x 0) 0)
        ((= y 0) 0)
        ((= x -1) y)
        ((= y -1) x)
        (else
         (let ((z (bitwise-and (div x 2) (div y 2))))
           (if (and (odd? x) (odd? y))
               (+ z z 1)
               (+ z z))))))
```

The use of `div` in `bitwise-and` cannot be replaced with `quotient`.

(2) Because `div` and `mod` are not R7RS procedures, complete definitions of `div` and `mod` were added to the R7RS version of `maze.scm`. Again, for the places where `quotient` was used in the *original* program, these routines give the same results as `quotient` and `modulo`, which *are* standard, built-in, R7RS procedures. But this is no longer true for the new definition of `bitwise-and`, which would be incorrect if we directly replaced `div` by `quotient`.

Maze: the results. The questions now are as Bill Pritchard asked: What do we want to measure with this benchmark? Are we succeeding?

When we run the Ecraven version of `maze.scm` on our version of Gambit, it runs in 1.594 seconds.

If we replace the handwritten `div` and `mod` procedures with the R7RS built-in `quotient` and `modulo` and replace the definition of `bitwise-and` with

```
(define (bitwise-and x y)
  (cond ((= x 0) 0)
        ((= y 0) 0)
        ((= x -1) y)
        ((= y -1) x)
        (else
         (let ((z (bitwise-and (quotient (- x (modulo x 2)) 2)
                               (quotient (- y (modulo y 2)) 2))))
           (if (and (odd? x) (odd? y))
               (+ z z 1)
               (+ z z))))))
```

then the benchmark runs in 1.299 seconds.

The arguments of `bitwise-not` have absolute value no more than 8, and the arguments of `bitwise-and` have absolute value no more than 18, i.e., they're small fixnums. The new definitions of `bitwise-not` and `bitwise-and` rely only on the built-in operations `+`, `-`, `=`, `quotient`, and `modulo`, and all these procedures are inlined when their arguments and results are fixnums (together with the checks to make sure that this is valid).

If we replace handwritten definitions of `bitwise-not` and `bitwise-and` with the built-in Gambit functions, the benmarck runs in 1.296 seconds. I was surprised that using built-in procedures did not improve performance.

After some investigating, I discovered that `bitwise-not` and `bitwise-and` were not inlined when the arguments are found to be fixnums. After I replaced them with definitions

```
(define bitwise-not
  (let ((old-bw-not bitwise-not))
    (lambda (x)
      (if (fixnum? x)
          (let ()
            (declare (not safe))
            (fxnot x))
          (old-bw-not x)))))
(define bitwise-and
  (let ((old-bw-and bitwise-and))
    (lambda (x y)
      (if (and (fixnum? x)
               (fixnum? y))
          (let ()
            (declare (not safe))
            (fxand x y))
          (old-bw-and x y)))))
```

the benchmark ran in 1.021 seconds.

We note that `maze.scm` is the only program in either the Gambit or Ecraven benchmark suites that has any bitwise operations.

We conclude that `maze.scm` is a good benchmark for measuring the performance of Scheme implementations on bitwise operations on small integers (fixnums) in the following circumstances:

(1) The Gambit version of the benchmarks is used to compare R5RS systems plus extensions for bitwise operations.

(2) Larceny's R6RS version is used to compare R6RS implementations.

The R7RS version in the Ecraven benchmark suite, however, does not add information to that provided by other programs (which also use arithmetic operations on small integers) unless it is changed as follows:

(1) The original procedures `quotient` and `modulo`, used originally in the Gambit benchmark sources, are used in the R7RS sources instead of makeshift versions of `div` and `mod`,

(2) The definitions of `bitwise-not` and `bitwise-and` are moved out of the `maze.scm` file, and

(3) Each implementation either provides `bitwise-not` and `bitwise-and` natively, or definitions that use the R7RS standard procedures `quotient` and `modulo` are provided in the associated "prelude" file.

Limiting oneself to standard R7RS (small) procedures and using inefficient, handwritten versions of `bitwise-and` and `bitwise-not` means this benchmark does not add any useful information about implementation quality beyond that found using other benchmarks.

Matrix. The `matrix.scm` is again found in the Gambit (R5RS plus extensions), Larceny (R6RS), and Ecraven (R7RS) benchmark suites.

Again, the Larceny version replaces `quotient` and `modulo` in the Gambit version by `div` and `mod` instead of just importing the `r5rs` library.

Then, in the R7RS Ecraven version, hand-rolled versions of `div` and `mod` were provided instead of using the built-in `quotient` and `modulo`.

The original Ecraven R7RS version run under Gambit takes 2.302 seconds.

Replacing the hand-written `div` and `mod` by the built-in `quotient` and `modulo` cut the runtime to 2.238 seconds.

Paraffins. Here again `quotient` in the Gambit version was replaced by `div` in the Larceny R6RS version instead of importing the `r5rs` library.

Then, instead of using the built-in `quotient` in the R7RS version, the following code was added:

```
(define (div x y)
  (quotient x y))
```

The Gambit compiler expands this to:

```
(define div
  (lambda (x y)
    (if (and ('#<procedure #8 ##fixnum?> y)
             (and ('#<procedure #8 ##fixnum?> x)
                  ('#<procedure #9 ##not>
                   ('#<procedure #33 ##eqv?> y 0))))
```

```
        (if ('#<procedure #33 ##eqv?> y -1)
            (or ('#<procedure #34 ##fx-?> x)
                ('#<procedure #35 quotient> x y))
            ('#<procedure #36 ##fxquotient> x y))
        ('#<procedure #35 quotient> x y))))
```

This inlined expansion of `quotient` performs suitable fixnum operations for fixnum arguments when appropriate, and calls the library `quotient` if necessary.

Gambit's inliner decides whether to inline a procedure based on the how much the inlining would increase the relative size of the destination procedure. The new `div` is large enough that it is not inlined at all call locations, as `quotient` would be.

And wherever this `div` is not inlined, Gambit's constant propagator cannot turn, e.g., `(div x 2)` into the more efficient

```
(if ('#<procedure #8 ##fixnum?> x)
    ('#<procedure #36 ##fxquotient> x 2)
    ('#<procedure #35 quotient> x 2))
```

Gambit runs the R7RS/Ecraven benchmark as written in 3.082 seconds. Textually replacing all uses `div` with `quotient` causes little performance difference with the tested Gambit: it runs in 3.075 seconds.

But unless one wants to measure how well the compiler inlines, constant folds, etc., small user-defined procedures, all uses of `div` should be textually replaced with `quotient`.

Compiler. This R7RS/Ecraven version of this benchmark again replaces the built-in procedures `quotient` by a handwritten `div` and the built-in procedures `modulo` and `remainder` with a handwritten `mod`.

The original benchmark runs in 2.701 seconds with Gambit. Textually replacing `div` with `quotient`, and `div` with `modulo`, gives a runtime of 2.364 seconds.

FFT. The analysis is the same as for `compiler.scm`.

The original benchmark runs in 2.791 seconds. Textually replacing `div` with `quotient` gives a runtime of 2.719 seconds.

Quicksort. In this benchmark one finds the code `(vector-map values v)`, which is equivalent to `(vector-copy v)`, which is a built-in R7RS procedure.

Making this substitution changes the execution time from 3.960 seconds to 3.785 seconds on Gambit.

Pi. The `pi.scm` benchmark computes π to an accuracy of 50 to 500 decimal digits in increments of 50 decimal digits. These computations are done twice for each number of decimal digits. So this benchmark tests the efficiency of integer arithmetic using relatively small bignums, far below the size where FFT-based algorithms would be of benefit, for example.

The original Gambit version uses an inefficient Newton-type iteration to compute roots of integer values (both square roots and fourth roots). The Ecraven R7RS benchmark uses the built-in `exact-integer-sqrt` built-in procedure of R7RS to compute quare roots, and iterates this procedure to compute fourth roots. Even though `exact-integer-sqrt` is also an R6RS procedure, the R6RS benchmark still incorporates the root procedures found in the Gambit benchmark.

The original Gambit version of the benchmark runs in .327 seconds. The R7RS version runs in 0.020 seconds.

So the original Gambit version of `pi.scm` is not a good test of generic bignum arithmetic—it spends nearly all its time computing square roots and fourth roots in an ad hoc, inefficient way. Switching to a more efficient library procedure cuts the run time by a factor of ten.

But even the R7RS version gives undue weight to the speed of square root. In general bignum arithmetic, square roots are relatively rare compared to addition/subtraction/multiplication/division, but they are a much more common operation in `pi`.

The `chudnovsky` benchmark, in the R7RS Ecraven benchmark suite, computes the same decimal approximations of π using a straightforward implementation of binary splitting applied to a series devised by the Chudnovsky brothers for this purpose. For each result, the program computes a single bignum square root and a single bignum quotient, while most of the rest of the operations are bignum multiplications, carefully arranged that there are few operations with the largest bignums. So in a number of ways `chudnovsky` is a better bignum benchmark than `pi` is.

In our case, Gambit runs `chudnovsky` in 0.010 seconds for 20 repetitions, compared to 0.020 seconds for 2 repetitions of `pi`; `chudnovsky` is about 20 times as fast as `pi` to compute the same results.

Or, `chudnovsky` does the same thing as the original `pi` program, only 200 times faster, and using a more representative mix of bignum operations.