A Reversible Computing Approach to Forth Floating Point.

Angel Robert Lynas and Bill Stoddart
Formal Methods and Programming Research Group
University of Teesside, UK

Abstract

We describe an implementation of floating point numbers in Reversible Forth using immutable references, and outline the advantages and disadvantages of this approach for the user and system.

Via a probabilistic algorithm example which requires a large number of random trials to create a sufficient sample size, we demonstrate the disadvantage of a naive program with regard to garbage creation, and its semi-automatic resolution using inbuilt features of the reversible virtual machine.

This allows otherwise prohibitively memory-intensive operations to be split into manageable pieces, interim results being saved while garbage is collected after each stage.

Keywords: Forth, Floating Point, Reversible Computing, Garbage Collection

1 Introduction

Forth works particularly well as a means of manipulating values that fit into a cell. Such values may be held as an item on the parameter stack, manipulated by the core stack operations such as `DUP` and `SWAP`, stored in standard (cell sized) constants and variables, and moved to and from the return stack to provide an alternative form of temporary storage. When floating point extensions are added, the situation is not quite so convenient: we now have values that may not fit in a cell. Most implementations use a separate stack to hold these values, they require their own stack operations, and (although we generally think of Forth as untyped) they require their own special variables and constants.

A cleaner approach, which need have only a marginal effect on speed of execution, is to use an "immutable reference semantics". A reference to a value of any kind will fit into a cell, and if the item referenced is immutable, we can hide the difference between references and the values they reference, leaving the application programmer free to to think of such references as being the values referenced. This approach can be applied to any

immutable data type, for example complex numbers. We also use it in our sets package, as described in a previous EuroForth article[8]. More generally immutable references are important in a number of languages, including Java and Python.

The disadvantage of our approach is that it generates garbage. This leads to the second theme of our paper, which is to describe how we handle garbage by exploiting our reversible version of Forth, the Reversible Virtual Machine (RVM)[1]

The rest of the paper is organised as follows. In Section 2 we introduce simple examples which use both integer and floating point types, and we introduce the RVM dispensation for local variables. In section 3 introduce our main working example, a probabilistic algorithm for the calculation of π. We give a naive version of the algorithm, which ignores collection of garbage, and then show how this is "wrapped" in a program structure which preserves the result of the computation and then collects any garbage by reverse execution. In Section 4 we look at memory issues: we describe the memory organisation of the RVM and we calculate the memory requirements for our probabilistic algorithm, and see that for a given level of accuracy it may be necessary to exceed the memory capacity of our virtual machine. In Section 5 we present an algorithm that takes as input a required level of accuracy, calculates the sample size, and, if the computation space requirements exceed the available memory, will collect garbage during the computation. In Section 6 we draw our conclusions.

2 Simple floating point examples

We can define a constant to give the value of π with

```
3.14159 CONSTANT PI
```

and then print the value of `PI` with:

```
PI F. <enter> 3.14159 ok
```

Here the Forth interpreter has been designed to leave a reference on the stack when it encounters a floating point literal in the input stream, and the definition of `F.` has been written to take a reference to a floating point number. However, since we never see, at this level, a floating point number as such, and all appropriate operations are written to use references to floating

[1]The RVM for Linux on the i386 Intel platform and its associated documentation is available from www.scm.tees.ac.uk/formalmethods

point values, we can safely think of the reference as *representing the floating point value*.

As a second example consider the following program which calculates the maturity value of a capital sum invested at a given annual rate of compound interest.

```
: MV ( r1 n r2 -- r3, calculate the maturity value of an in-
vestment of r1 for n years at annual compound interest of r2% )
  100. F+  100. F/ ( convert % rate to yearly ratio )
  SWAP >R SWAP ( P: ratio capital, R: years )
  R> 0 DO ( ratio capital )
    OVER F*
  LOOP NIP ;
```

Here we see Forth stack manipulation operations being used to manipulate both floating and integer values. To find the maturity value of 1000 currency units invested for two years at 5% we could use:

```
1000. 2 5. MV F. <enter> 1102.5 ok
```

We present this value again using a local variable approach. In our dispensation of Forth we use `VALUE` as a defining word *within definitions* to create local variables. Locals defined within `(: .. :)` brackets[2] are initialised from values which are in the stack at the point the stack frame is declared (generally at the start of the definition). Those values are matched against the declared locals from *left to right* (unlike the Forth Standard approach). Locals declared in the body of the procedure are initialised from stack values created by code which follows the stack frame. We also need to state how many stack cell are returned by the definition.

```
: MV ( r1 n r2 -- r3, calculate the maturity value of an in-
vestment of r1 for n years at annual compound interest of r2% )
  (: VALUE CAPITAL  VALUE YEARS  VALUE RATE :)
    RATE 100. F+  100. F/ VALUE RATIO
    YEARS 0 DO
      CAPITAL RATIO F* to CAPITAL
    LOOP
  CAPITAL 1LEAVE ;
```

Here we see that the same defining word can be used for both integer and real variables. We still need to distinguish integers and floats when it comes

[2]Our original work on local variables from 1985[7] uses { } brackets to enclose variable lists. However, since the RVM makes extensive use of sets, and these brackets are the standard math set brackets, we now reserve them for this purpose.

to performing operations on them however: if asked to calculate `1. 2. +` the system will add the references to the floating point values given, a totally meaningless computation.

The disadvantages of using a reference semantics in this context are, firstly a slight speed penalty associated with accessing data values via references, and secondly that we need to manage the memory locations in which the floating point values are stored: we need a garbage collector.

Our method of collecting garbage, previously suggested by Henry Baker[1] is based on the use of a reversible machine. Such machines, which preserve information at each step, are of theoretical interest because they minimise the absolute energy requirements of a computation[3, 5].

3 Example application: a probabilistic algorithm for the calculation of π

.

As a working example for the rest of the paper we present a probabilistic algorithm for estimating of the value of π. Consider Figure 1. If we choose a random value for x, uniformly distributed in the interval -1, to 1, and a random value for y, also uniformly distributed in the interval -1, to 1 we obtain a random point within the given square. The probability that this point will fall within the circle is given by the ratio of the areas of the circle and the square, namely $\pi/4$.

A simple way to estimate $\pi/4$ is therefore to choose n points, record the number S_n that fall within the circle, and calculate the estimate

$$\pi/4 \approx S_n/n$$

We use our random number generator `RAND`, which gives values in the range `0..MAXINT`, to provide a word that gives uniformly distributed floating point values within a given range `r1` to `r2`. We also make use of the word `S>F`, which converts a single precision integer to a floating point value.

```
: FUNIFORM ( r1 r2 -- r3,
  r3 is chosen from the uniform distribution between r1 and r2)
  (: VALUE r1  VALUE r2 :)
    RAND S>F VALUE frand
    RAND_MAX S>F VALUE frmax
    r1  r2 r1 F-  frand frmax F/  F*  F+
  1LEAVE ;
```

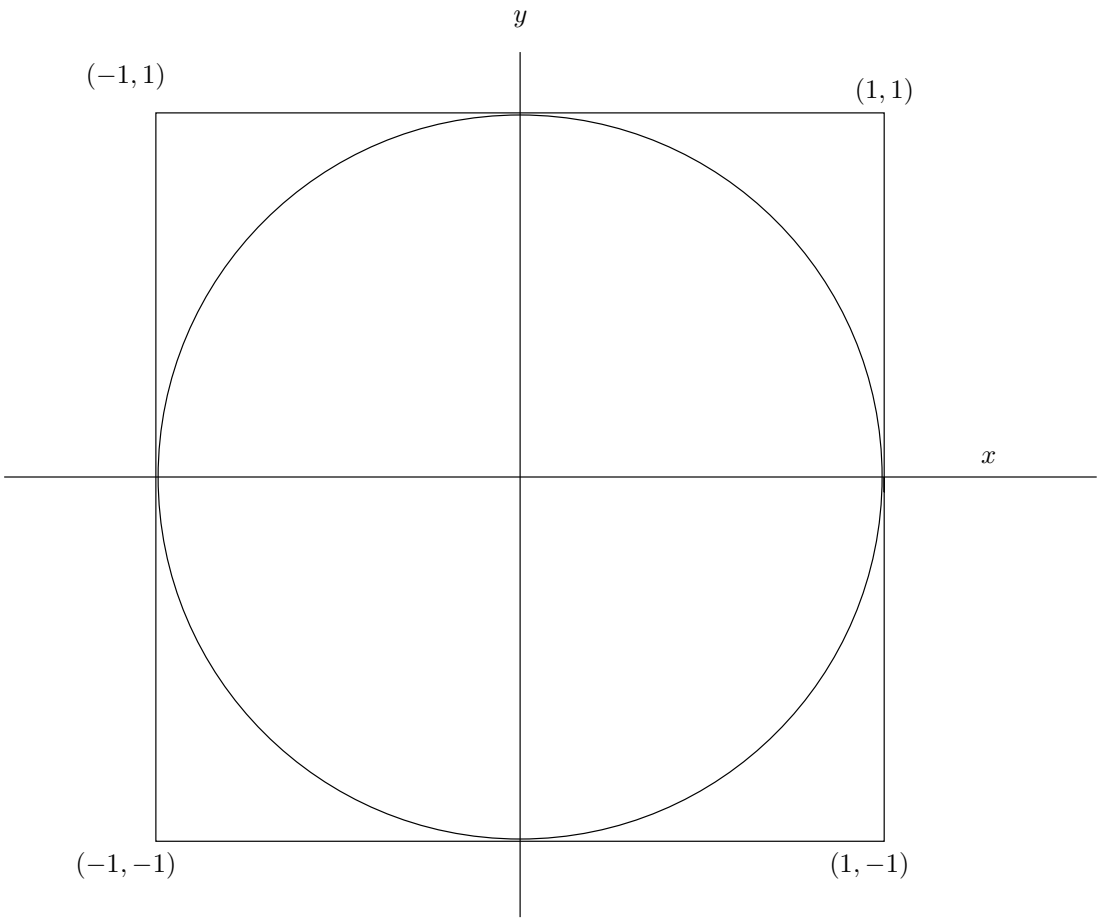

Figure 1: A circle of area π inscribed within a square of area 4.

Thus `-1. 1. FUNIFORM` will yield a random value between -1. and 1.

We can now code a naive word to estimate $\pi/4$. As input we give the sample size n on which the estimate is to be based. On each iteration of the loop we generate a random point (x, y) and calculate whether it is within the circle according to whether $x^2 + y^2 < 1$. If it is we add one to the count of "successes" S_n. Finally we calculate an estimate of $\pi/4$ using a floating point division of S_n/n.

```
: PI/4_0 ( n -- r, n<20000 and r is an estimate of pi/4 based on
a sample size n. Values of n>20000 may cause problems )
  (: VALUE n :)
    0 VALUE Sn  NULL VALUE x  NULL VALUE y
    n 0 DO
      -1. 1. FUNIFORM to x
```

```
    -1. 1. FUNIFORM to y
    x x F*  y y F*  F+ 1. F<
    IF Sn 1+ to Sn THEN
  LOOP
  Sn S>F  n S>F  F/
1LEAVE ;
```

We have called this program naive because it leaves garbage and because its sample size is limited by the memory configuration of the RVM. We first deal with the garbage by wrapping the definition within a programming structure which is particular to reversible computing. The effect of this structure was first described in terms of reversible Turing machines by Lecerf in 1963[6] and was independently given by Bennett[2], who also related it to the physics of reversible computation. Within the RVM it is formulated as follows. Let *prog* represent Forth code whose stack effect is to generate a floating point value. Then the structure:

`<RUN` *prog* `FLOAT>`

will run *prog*, preserve the floating point value created, reverse execution and thus undo all the other effects of *prog* (including removing the garbage that *prog* created) and finally leave the value created by *prog* on the stack. With this structure we can code a refinement of `PI/4_0` called `PI/4_1` which returns the same value as `PI/4_0` but collects garbage.

```
: PI/4_1 ( n -- r, as PI/4_0, but garbage is collected )
  <RUN DUP PI/4_0 FLOAT> NIP ;
```

this program is still limited in the sample size it can handle, and we now turn our attention to calculating the approximate sample size required for our algorithm to (probably) give us some given degree of accuracy.

4 Memory Issues

In our immutable references approach each floating point value created during a computation remains in memory until it is removed by reverse computation. The most general way to hold such memory references is on a heap, but the fixed size and atomic (unnested) nature of floating point values allows us to manage them more efficiently as a stack. Unlike the floating point stack of most implementations, items are only dropped from our floating point stack during reverse execution. The floating point stack is, in effect, a large floating point array, and this form of organisation places an absolute limit

on the number of floating point values that can be accommodated before a program needs to reverse.

The program developed so far is thus strictly limited in the sample size it can handle, and we now turn our attention to calculating the approximate sample size required to give us some given degree of accuracy with a given degree of confidence. The conclusions will be that much larger samples be needed than can be accommodated within the allocated floating point stack memory of the RVM.

We are interested in knowing the number of trials required to obtain an estimate of π which is accurate to within plus or minus some error e with (say) 95% confidence.

Note that if p is the probability of a particular trial falling within the circle, and S_n, the number of trials that fall within the circle when n trials are performed, then S_n follows a binomial distribution with mean $n * p$ and variance n*p*(1-p). For large n this binomial distribution approximates to the normal distribution with the same mean and variance.

Now in general, if X is a random variable with mean μ and variance σ^2 then $a * X$ is a random variable with mean $a * \mu$ and variance $(a * \sigma)^2$, thus S_n/n will have mean p and variance $p * (1-p)/n$ and accordingly also standard deviation $\sqrt{p * (1-p)/n}$. A normally distributed random variable has probability of just over 0.95 of being within ± 1.96 standard deviations of its mean. Thus to have a probability of 0.95 of S_n/n being within $\pm e$ of p we choose n to obtain:

$$e > 1.96 * \sqrt{\frac{p * (1-p)}{n}}$$

Recalling that $p = \pi/4$ we have

$$e > 1.96 * \sqrt{\frac{\frac{\pi}{4} * (1-\frac{\pi}{4})}{n}}$$

assuming a positive square root and squaring both sides

$$e^2 > 1.96^2 * \frac{\frac{\pi}{4} * (1-\frac{\pi}{4})}{n}$$

multiplying each size by $\frac{n}{e^2}$

$$n > \left(\frac{1.96}{e}\right)^2 * \frac{\pi}{4} * \left(1 - \frac{\pi}{4}\right)$$

This tells us the minimum sample size for use with our random algorithm when a certain degree of accuracy is required, but this result depends on π

the value we are trying to calculate! However, if we insert a low estimate of π, say 3.1, we will obtain a high estimate of n since if we treat π as a variable the term $\pi * (4 - \pi)$ has a single maximum where $\pi = 2$.

The following word calculates the value of n as a double number.

```
: ?TRIALS ( e -- d, calculate no of trials d required to have
95% conf of obtaining result within pi-e to pi+e. We use the
crude (and low) estimate of 3.1 for pi in this calculation,
giving a higher value for d than is strictly required )
  (: VALUE e :)
    1.96 e F/ DUP F* ( (1.96/e)^2 )
    3.1 ( crude estimate of pi, but referred to as pi in stack
    comments )
    4. F/ ( (1.96/e)^2 pi/4 )
    1. OVER F- ( (1.96/e)^2 pi/4 1-pi/4 )
    F* F* F>D
  2LEAVE ;
```

Some example calculations showing the number of trial required for various degrees of accuracy are:

```
0.01 ?TRIALS D. 6474 ok
0.001 ?TRIALS D. 647493 ok
0.0001 ?TRIALS D. 64749356 ok
0.00001 ?TRIALS D. 6474935678 ok
```

And we note that the last value is beyond the range of 32 bit integers. We see that our algorithm is not a practical way to calculate a very accurate value. On the other hand it is a simple method of obtaining a rough estimate for π. Since the value of π has been known to over a million places for more than 30 years, the value of this algorithm is purely didactic, suggesting quick solutions to a range of problems where similar techniques can be employed.

5 Estimating π to a given level of accuracy

Our next refinement divides the calculation up into a number of separate experiments, each of which can be accommodated within the memory limitations of the RVM, assumed to be able to handle a sample size of 10000 before needing to reverse.

We use some global variables as an alternative to declaring a stack frame. Their longevity will give us increased visibility in case any debugging is required.

```
NULL VALUE PI_TOTALS  10000 CONSTANT INNER-LOOP
NULL VALUE OUTER-LOOP
```

In the following word the estimates of $\pi/4$ produced by each invocation of `PI/4_1` each use the maximum allowable sample size. These estimates are added together and kept in `PI-TOTALS`. The final total is divided by the outer loop count to find the average estimate of $\pi/4$.

```
: PI/4_2 ( n -- pi,
  produce an n estimate sample of pi/4. If n is too
  large for a single pass the calculation is split
  into a number of stages. Garbage from the outer
  loop is not collected yet )
  INNER-LOOP / 1+ to OUTER-LOOP
  0. to PI_TOTALS
  OUTER-LOOP 0 DO
    I SRAND ( seed pseudo random number generator )
    INNER-LOOP PI/4_1 PI_TOTALS F+ to PI_TOTALS
  LOOP PI_TOTALS OUTER-LOOP S>F F/ ;
```

In the above definition, we see that we are using a pseudo-random number generator. We must seed this with a different value for each iteration. The use of pseudo-random numbers makes our algorithm heavily dependent on the performance of the particular pseudo random number generator employed. Using gnu C library routine `rand`, for example, it is not possible to reliably obtain a result accurate to more than 4 or 5 significant figures.

We now present our final program, It takes as input a required level of accuracy. From this is calculates the required sample size, and will issue an error if more than `UMAXINT` trials are required. Otherwise is uses the `<RUN .. FLOAT>` wrapper to encapsulate the previous program and thus collects the garbage that program leaves. Finally it multiplies the estimate of $\pi/4$ by 4 to give an estimate of π.

```
: ESTIMATE_PI ( e -- r, produce a probabilistic estimate of pi
  which has probability 0.95 of being accurate to within an
  error of +/- e. All garbage is collected. If more than UMAXINT
  trials are required an error will be reported)
  <RUN
    DUP ?TRIALS ABORT" Too many trials needed by ESTIMATE-PI"
    PI/4_2
  FLOAT> NIP 4. F* ;
```

6 Conclusions

An immutable reference semantics allows us to deal neatly with data types, such as floating point, that cannot be held in a cell of memory. The three qualifications to this approach are that the data referenced must be immutable (so it is not a suitable approach for use with arrays), there will be a small speed penalty incurred, and the referenced values remain in memory as garbage until removed by reverse computation.

The availability of reverse execution allows us to collect garbage at no additional cost, and an appropriate program structure, particular to reversible computing, is available to do this in our Reversible Virtual Machine. Where a program has more extensive memory requirements than can be accommodated by a "one pass" algorithm, the application programmer must split its execution into stages, each of which returns information required for the final result, and each of which collects its own garbage. Thus garbage collection is best thought of as "semi-automatic" in our approach. A more general approach to re-use of memory resources during reversible computation is studied in [4]. Unlike the method given there, our approach, which is specific to a certain class of problems, requires virtually no additional execution overhead.

Since the value of π has been known to over a million places for more than thirty years, the value of the algorithm presented in this paper is purely didactic, suggesting simple solutions to a range of problems where similar techniques can be employed. As an example of such a problem consider a circle of radius r_1 with a second circle of radius r_2 constructed with its centre on the circumference of r_1. The problem is to find what proportion of each circle lies within the other.

References

[1] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in Lecture Notes in Computer Science, 1992.

[2] C Bennett. The Logical Reversibility of Computation. *IBM Journal of Research and Development*, 6, 1973.

[3] C Bennett. The Thermodynamics of Computation. *International Journal of Theoretical Physics*, 21 pp 905-940, 1982.

[4] C Bennett. Time-space Trade-offs for Reversible Computation. *SIAM Journal of Computing*, 18, 1989.

[5] R P Feynman. *Lectures on Computation.* Westview Press, 1996.

[6] Yves Lecerf. Machines de Turing Reversibles. *Comptes rendus de l'Académie Française des Sciences*, 257(1963), 1963.

[7] W. J. Stoddart. Readable and Efficient Parameter Access via Argument Records. *The Journal of Forth Application and Research*, 3(1), 1985.

[8] W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In M. A. Ertl, editor, *18th EuroForth Conference Proceedings*, September 2002. On-line proceedings.