# The Most Unreliable Technique in the World to compute $\pi$

Jerzy Karczmarczuk
*Dept. of Computer Science, University of Caen, France*

(mailto:karczma@info.unicaen.fr)

**Abstract**

This paper is an atypical exercice in lazy functional coding, written for fun and instruction. It can be read and understood by anybody who understands the programming language Haskell. We show how to implement the Bailey-Borwein-Plouffe formula for $\pi$ in a co-recursive, incremental way which produces the digits 3, 1, 4, 1, 5, 9... until the memory exhaustion. This is *not* a way to proceed if somebody needs many digits! Our coding strategy is perverse and dangerous, and it *provably* breaks down. It is based on the arithmetics over the domain of infinite sequences of digits representing proper fractions expanded in an integer base. We show how to manipulate: add, multiply by an integer, etc. such sequences from the left to the right *ad infinitum*, which obviously cannot work in all cases because of ambiguities. Some deep philosophical consequences are discussed in the conclusions.

## 1 Introduction

We all know that the world oceans are saturated by algorithms to compute $\pi$. These algorithms have accumulated over centuries, and it is difficult to become famous by just computing another billion digits of this transcendental number, despite the fact that the Humanity badly needs those digits. The reader will find almost all the relevant and irrelevant information about $\pi$ on the Web: [1]. Why almost? Because the topic is so vital that many people still work hard on it, day and night. The glorious times of William Shanks, who computed many digits of $\pi$ by hand in 1853 (and spent ten years on finding erroneous digits which followed a mistake infiltrated somehow into the sequence...) are *not* over. Last year Fabrice Bellard [2] found the billionth (european, $10^{12}$) binary digit of $\pi$ employing a modified Bailey-Borwein-Plouffe formula [3], accelerated by him by the whole 43%, and the world did not freeze in astonishment.

Seriously, the pleasure of searching is much more important than the result itself, and the future belongs perhaps to those who will invent the most unusual, baroque and expensive ways to compute $\pi$ or other magic numbers. Perhaps new experimental methods? The book [4] quotes the experiment (published on Usenet lists) of Dave Boll, who analyzed the structure of filaments on the border of the Mandelbrot fractal set (the "bug") at $c = -0.75 + \epsilon i$, and found that the number of iterations needed to leave the "neck" is equal to $\pi/\sqrt{\epsilon}$, which generated such numbers as 3141592 when $\epsilon$ was an even power of $10^{-1}$. The folklore of $\pi$ continues. But the BBP formula itself is really remarkable both from the theoretical point of view, and for eventual applications. It is based on some special identities fulfilled by certain polylogarithmic functions [3]. The authors are known specialists in constructive mathematics. Their formula is a development in powers of 1/16:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left( \frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right). \tag{1}$$

Each coefficient is rational, and its contribution for the distant digits may be estimated. It is possible thus to compute *any* hexadecimal (or binary) digit of $\pi$ without infinite-precision arithmetic, nor large amount of memory, because previous digits are not needed. Of course, converting the result into decimal is another story. Now the entire sequence of digits is involved.

Our ambition does not go into the direction of computing new digits of $\pi$. We want just to show on an amusing example that the lazy semantics provides some new *coding techniques*, which might be useful, at least pedagogically, and for the brave people in the domain of scientific computing. In a sense this paper is a conceptual continuation of [5]. The authors of [3] used the standard floating-point arithmetic to construct the numerical result. Our coding is purely integer, and it may be coded even using standard short `Ints`, but then one has to be very careful.

# 2 Lazy Arithmetic of Infinite Fractions

Everybody agrees that representing 2/7 by 0.285714285714... is the worst possible solution for practical computations. It must be truncated, and all numerical manipulations of floating-point numbers introduce errors. If the mantissa is long, the errors are smaller, but the computation takes longer. All truncation is awkward anyway, in all computations as in real life. We propose thus to represent such fraction as above by a *lazy* infinite list, for example:

```
twosevenths=0 : cycle [2,8,5,7,1,4]
```

This is just a particular case. Even simpler is the "lifting" of zero: `sZero = repeat 0`. We shall keep the entire part of the number as the first digit of the list without expanding it further, and we can generate other numbers by appropriate lazy algorithms. Choosing an integer `base` (10 for testing, and 16 for our final BBP computation), we can easily convert any rational fraction into our domain by the standard Euclidean chain:

```
fcn n d = let (a,b)=quotRem n d in a : fcn (base*b) d
```

which has nothing unusual, it is a perfectly legitimate co-recursive procedure, guaranteed to progress, and its incremental consumption is absolutely sane and safe, see [6]. The call `fcn 5 3` generates the list $[1, 6, 6, 6, \ldots]$. But can we do something with such fractions? Adding element by element the expansions of 2/7 and 5/7=0.71428571... gives 0.9999999... which is the most silly way to represent 1, but it coexists with 1.000... as a different piece of data.

However, if we avoid such cases, i.e. if we admit that our arithmetic is inherently sick and incomplete (like standard floating-point arithmetic for a True Mathematician), we may construct the addition of these infinite fractions. The carry propagation is resolved by lookahead, and the main trick consists in combining the lazy and strict recursion. First we add everything element-wise, and then we propagate the carry from the unknown future by almost trivial, deterministic guessing. This propagation breaks down, and forces another level of strict recursion when we encounter the digit 9 (`base-1` in general case).

```
u <+> v = let (w0:wq)=zipWith (+) u v in cpr w0 wq where
 cpr u0 (u1:uq)| u1<base1  = u0 : cpr u1 uq
               | u1>base1  = (u0+1) : cpr (u1-base) uq
               | otherwise = let v@(v0:vq)=cpr u1 uq in
                 if v0<base then u0:v else (u0+1):(v0-base):vq
```

The subtraction is straightforward, in order to negate a number we permit the 0-th element to be negative, and the fractional part is 9-complemented

```
neg (u0:uq) = (negate u0 - 1) : map (base1 -) uq
```

Of course it is better not to try to subtract a number from itself, as this particular way of computing zero is quite expensive, it generates *bottom*: $-1 + 0.9999\ldots$. We don't discuss further the manipulation of negative numbers. For example their division by something may be performed by negating it, dividing, and negating again.

We might need the multiplication and the division of an expanded number by a digit `m`. The division `(u >/ m)` is simpler, and it is co-recursively sane.

```
u@(u0:_) >/ m = dvd 0 u where
  dvd c (u0:uq) = let (n,r) = quotRem (base*c+u0) m
                  in n : dvd r uq
```

The multiplication `(m *> u)` is again a *horrendum*. Multiplying $1/3$ by 3 will not work, and if we want to be able to compute $5 \cdot 1/3$, it is better to avoid the construction of the multiplication as the iterated addition. The multiplication procedure is the longest piece of our code:

```
0 *> u = sZero
1 *> u = u
m *> u@(u0:u1:uq) =
 let (c,r)=quotRem (m*(u0*base+u1)) base
     cm c r (v0:vq) =
       let (a,b)=quotRem (m*v0) base;    p=a+r
       in if p<base1 then c : cm p b vq else
          if p>base1 then (c+1) : cm (p-base) b vq
          else let w@(w0:wq) = cm p b vq
               in if w0<base then c : w else (c+1) : 0 : wq
     in cm c r uq
```

Again, we construct a tentative digit, summing the contribution of two neighbouring digits of the fraction `u`, and an eventual correction from the unknown future may consist only in adding 1. This is safe if the concerned digit is different from `base-1`, because in this case the *previous* digit is frozen. Now we can convert the expanded fractions from one base to another, and this is our main contribution to the BBP algorithm, admittedly neither too original nor laborious. Here it is, the old base is the global `base`, and the new one is passed as the second parameter.

```
bconv (u0:uq) nBase = u0 : bconv (nBase *> (0:uq)) nBase
```

This conversion from the base 16 into 10 will fail if the number of consecutive "9" is bigger than the size of the computer recursive stack. Of course $\pi$ must include such sequence, and our algorithm *must* break down, but it might also die earlier and happier, because of the accumulation of lazy thunks within the main heap. (This did not happen in our experiments, we tried to avoid spurious memory leaks in forming the lazy fractions.)

## 3 The Main Computation

We may implement now the formula (1). We can either sum the rational fractions and convert the sum into our lazy expansion:

```
frm i = let j=8*i
            (a :% b)=4%(j+1)-2%(j+4)-1%(j+5)-1%(j+6)
        in fcn a b
```

or sum the four expansions separately. Both methods have their advantages. The first one produces less lazy lists in the storage, but needs the long `Integer` numbers, the `Int` datatype on some platforms (for example the Hugs 1.4 implementation unders Windows NT) fails, because the rational sum above creates quickly very voluminous rationals. The second one is slower, but safer from this point of view. However, here the user should treat separately the 0-th term, otherwise the program will immediately squeak and die, producing bottom by subtraction $4 - 1/2$. This is relatively easy to correct, but we shall not insist upon it.

And that is almost all, but how do we compute the infinite sum in (1)? In fact this is a *shifted* sum: $U^{(0)} + U^{(1)}/B + U^{(2)}/B^2 + \cdots$ where $B$ is the base (16). We construct first a list of terms: $[U^{(0)}, U^{(1)}, U^{(2)}, U^{(3)}, \ldots]$, where each term $U^{(k)}$ represents the coefficient in the BBP formula. The addition $f + g/B + X/B^2$ is "almost" easy, it first digit needs only the first digit of $f$, and eventually the carry propagated from $g$, if *its* first digit is too big. We exploit the same look-ahead trick already seen in normal addition, and we code

```
pi16 = ssum (map frm [0 ..]) where
  ssum (u@(u0:u1:uq) : v@(v0:_) : r) =
   if u1+v0<base1 then u0 : ssum ((u1:uq)<+>v : r)
   else let s = ssum (v:r) in u <+> (0:s)

ourpi=bconv pi16 10
```

And now, unbelievable, but true, it suffices to type `ourpi`, and to start writing this article. Before it is finished we get on the screen

```
[3141592653589793238462643383279502884197169399375105820974944
4459230781640628620899862803482534211706798214808651328230664
4709384460955058223172535940812848111745028410270193852110555
5964462294895493038196442881097566593344612847564823378678316
6527120190914564856692346034861045432664821339360726024914127
7372458700660631558817488152092096282925409171536436789259036
6001133053054882046652138414695194151160943305727036575959195
5309218611738193261179310511854807446237996274956735188575272
2489122793818301194912983367336244065664308602139494639522473
3719070217986094370277053921717629317675238467481846766940513
3200056812714526356082778577134275778960917363717872146844090
0122495343014654958537105079227968925892354201995611212902196
6086403441815981362977477130996051870721134999999837297804995
5105973173281609631859502445945534690830264252230825334468503
3526193118817101000313783875288658753320838142061717766914730
0359825349042875546873115956286388235378759375195778185778053
3217122680661300192787661119590921642019893809525720106548586
6327886593615338182796823030195203530185296899577362259941389
9124972177528347913151557485724245415069595082953311686172785
5588907509838175463746493931925506040092770167113900984882401
1285836160356370766010471018194295559619894676783744... ]
```

where we have reformatted a little the output, which was actually longer, in fact *we* gave up after having generated 1800 digits. We have checked the result, which was almost redundant, as the program is so trivial that we had no chance to commit an error, but a similar computation in the past discovered an error in one rational number package...

# 4 Conclusions

Our main conclusion is that the program works, although it needn't to, and after a finite number of steps it will fail. In fact, this is the only such program which works without knowing how many digits it should generate, if at all. In such a way we prove that we have done a serious work. A mathematically-oriented reader whose private definition of the word "serious" is different from ours, may reject our technique, and our only defense is that this technique is not ours... All the financial aid demanded by the governments of the Third World is based on the principle of borrowing money without *any* guarantee to pay it back one day. In practice this works, because the bank buffers of the rich countries are voluminous enough. The very idea of a bank credit is a lazy co-recursive (runaway) algorithm, and the true Game consists in running away before the bottom reaches you. The social security depenses in several european countries operate on non-existing funds which will (hopefully) un-virtualize themselves thanks to the work of future generations. Yes, the economy is a non-strict science. So, why should we hesitate to borrow a carry from the "future" digits? At least this is our free choice, while paying taxes is not.

The idea might be much more fundamental than that. From the relativistic cosmology we know that the negative gravitational energy almost entirely compensates the energy stored in the matter. Thus, apparently the Big Bang is based on a lazy quantum algorithm, borrowing energy from the dynamic of the future unrolling Universe in order to construct it, and compensating the budget by the gravitational tension. The Almighty obviously preferred the functional programming over the imperative one, but that we knew already for years, the first Light in the Universe is a `let` construct.

This paper has been written for sheer intellectual pleasure, and it will remain unpublished in the Virtual Journal of Unpublished Papers. Publishing everything makes it impossible to discover the Truth in the literature, and thus it serves the Devil. The author thanks his family for their constant emotional support and aspirin.

## References

[1] Everything you ever wanted to know about $\pi$, and much more:
`http://www.cecm.sfu.ca/pi` or
`www.go2net.com/internet/useless/useless/pi/pi-pages.html`

[2] The Web site of Fabrice Bellard:
`http://www-stud.enst.fr/~bellard/pi-challenge/index.html`

[3] David Bailey, Peter Borwein, Simon Plouffe, *On the Rapid Computations of Various Polylogarithmic Constants*, Math. Comp. **66** (1997), pp. 903–913.

[4] H.-0. Peitgen, H. Jürgens, D. Saupe, *Fractals for the Classroom. Part 2: Complex Systems and Mandelbrot Set*, Springer, N. Y., (1992).

[5] Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*, Theoretical Computer Science **187**, (1997), pp. 203–219.

[6] David A. Turner, *Elementary Strong Functional Programming*, Proceedings of the Symposium *Functional Programming Languages in Education*, FPLE'95, Nijmegen; Springer, LNCS 1022, (1995), pp. 1–13.