The Arithmetic of Even-Odd Trees

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

SYNASC'2015

– research supported by NSF grant 1423324 –

Overview

- we describe a tree-based arithmetic system that applies recursively a run-length compression mechanism
- we perform arithmetic computations symbolically as tree transformations
- tractability of computations is only limited by the tree-representation size rather than the bitsize of their operands
- we describe tree-based arithmetic algorithms that are
  1. within constant factors from their traditional counterparts for their average case behavior
  2. super-exponentially faster on some “interesting” giant numbers
- $\Rightarrow$ we make tractable important computations that are impossible with traditional number representations

Related work

- a tree-based number system occurs in the proof of Goodstein's theorem (1947) , where replacement of finite numbers on a tree's branches by the ordinal ω allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates
- notations vs. computations
  - notations for very large numbers have been invented in the past ex: Knuth's up-arrow
  - in contrast to our tree-based natural numbers, such notations are not closed under successor, addition and multiplication
- this paper is describes a simpler and more elegant tree-based arithmetic system than our previous work
- most likely the final version of a series of papers on unconventional arithmetic

Can we give up the *“egalitarian”* view of numbers?

- moving from unary arithmetic to a binary system results in an exponential speed-up
- this speed-up applies *fairly* to all numbers independently of their completely random or highly regular structure
- from information theory: we cannot improve on the *average* complexity of binary arithmetic operations by changing the representation
- can we do accelerate computations further if we give up this *egalitarian* view?
- can we treat better some classes of numbers, to favor interesting computations with them, without more than constant time prejudice to the others?

Arithmetic operations on top of an *"elitist"* number system

- we introduce an "elitist" number system that answers these questions positively!
- numbers with a "regular structure" (recursively made of large alternating blocks of 0s and 1s) receive preferential treatment (up to super-exponential acceleration)
- the average performance of our arithmetic operations remains within constant factor of their binary equivalents
- our numbers will be represented as ordered rooted trees obtained by recursively applying a form of run-length compression
- our algorithms are presented as purely functional specifications, in a literate programming style
- we use a small subset of Haskell as an executable mathematical notation

Binary arithmetic as function composition

- natural numbers larger than 1 can be seen as represented by iterated applications of the functions $o(x) = 2x$ and $i(x) = 2x + 1$ starting from 1
- each $n \in \mathbb{N} - \{0, 1\}$ can be seen as a unique composition of these functions applied to 1
- $2 = o(1), 3 = i(1), 4 = o(o(1)), 5 = i(o(1))$ etc.
- applying o adds a 0 as the lowest digit of a binary number
- applying i adds an 1 as lowest digit
- also: $i(x) = o(x) + 1$

Arithmetic with the iterated functions o^n and i^n

- simple arithmetic identities can be used to express "one block of o^n or i^n operations at a time" algorithms for various arithmetic operations

$$o^n(k) = 2^n k \tag{1}$$

$$i^n(k) = 2^n(k+1) - 1 \tag{2}$$

In particular

$$o^n(1) = 2^n \tag{3}$$

$$i^n(1) = 2^{n+1} - 1 \tag{4}$$

- one can directly relate o^k and i^k:

$$i^n(k) = o^n(k+1) - 1. \tag{5}$$

Even-Odd trees as a data type

Definition

The Even-Odd Tree *data type* `Pos` *is defined by the Haskell declaration:*

```
data Pos = One | Even Pos [Pos] | Odd Pos [Pos]
           deriving (Eq, Show, Read)
```

corresponding to the recursive data type equation $\mathbb{T} = 1 + \mathbb{T} \times \mathbb{T}^* + \mathbb{T} \times \mathbb{T}^*$.

- the term `One` (empty leaf) corresponds to 1
- the term `Even x xs` counts the number `x` of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications `xs`
- the term `Odd x xs` counts the number `x` of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications `xs`
- we proceed recursively until reaching the empty leaf corresponding to 1

The bijection $p' : Pos \to \mathbb{N}^+ = \mathbb{N} - \{0\}$

$$p'(t) = \begin{cases} 1 & \text{if } t = \texttt{One}, \\ 2^{p'(\texttt{x})} & \text{if } t = \texttt{Even x []}, \\ p'(u)2^{p'(\texttt{x})} & \text{if } t = \texttt{Even x (y:xs)} \text{ and } u = \texttt{Odd y xs}, \\ 2^{p'(\texttt{x})+1} - 1 & \text{if } t = \texttt{Odd x []}, \\ (p'(u)+1)2^{p'(\texttt{x})} - 1 & \text{if } t = \texttt{Odd x (y:xs)} \text{ and } u = \texttt{Even y xs}. \end{cases} \tag{6}$$

Examples

- this bijection ensures that Even-Odd Trees provide a canonical representation of natural numbers
- the equality relation on type `Pos` can be derived by structural induction

- Example 1:
  - `Even (Even One []) [One,One]` $\rightarrow ((2^1+1)2^1-1)2^{2^1}\ 2^{2^{16}}-1 \rightarrow 20$
- Example 2:
  - `Odd (Odd (Odd (Odd One [])[])[])[]`
  - $\rightarrow 2^{2^{2^{2^{1+1}-1+1}-1+1}-1+1}-1$
  - $\rightarrow 2^{2^{16}}-1$

From a binary number to a list of counters

To implement the inverse $p : \mathbb{N}^+ \rightarrow Pos$ of the function p' we first split the binary representation of a number as a list of alternating 0 and 1 counters.

Each counter k corresponds to a block of applications of either o^k or i^k.

```
to_counters :: Integer  -> (Integer, [Integer])
to_counters k = (b,f b k) where
  parity x = x `mod` 2
  b = parity k

  f _ 1 = []
  f b k | k>1 = x:f (1-b) y where (x,y) = split_on b k

  split_on b z | z>1 && parity z == b = (1+x,y) where
    (x,y) = split_on b  ((z-b) `div` 2)
  split_on b z = (0,z)
```

The inverse bijection $p : \mathbb{N}^+ \to Pos$

The function `p` maps the empty list of counters to 1, a non-empty list of counters derived from an even (respectively odd) number to a term of the form `Even x xs` (respectively `Odd x xs`).

```
p :: Integer -> Pos
p k | k>0 = g b ys where
  (b,ks) = to_counters k
  ys = map p ks
  g 1 [] = One
  g 0 (x:xs) = Even x xs
  g 1 (x:xs) = Odd x xs
```

The first 4 positive numbers represented as Even-Odd Trees:

1. One
2. Even One []
3. Odd One []
4. Even (Even One []) []

The DAG representation of the largest known prime number

- a more compact representation is obtained by folding together shared nodes in one or more Even-Odd Trees.
- integers labeling the edges are used to indicate their order.

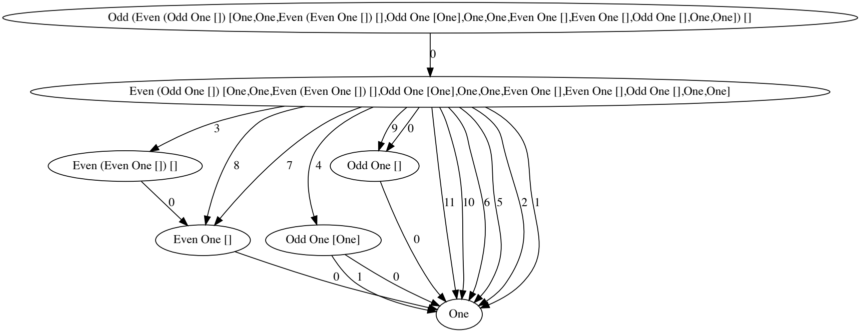

Figure: the Mersenne prime $2^{57885161} - 1$

The successor s

```
s :: Pos -> Pos
s One = Even One []
s (Even One []) =  Odd One []
s (Even One (x:xs)) =  Odd (s x) xs
s (Even z xs) = Odd One (s' z : xs)
s (Odd z [])  = Even (s z) []
s (Odd z [One]) = Even z [One]
s (Odd z (One:y:ys)) = Even z (s y:ys)
s (Odd z (x:xs)) = Even z (One:s' x:xs)
```

The predecessor s'

```
s' :: Pos -> Pos
s' (Even One []) = One
s' (Even z []) = Odd (s' z) []
s' (Even z [One]) =  Odd z [One]
s' (Even z (One:x:xs)) =  Odd z (s x:xs)
s' (Even z (x:xs)) =  Odd z (One:s' x:xs)
s' (Odd One []) = Even One []
s' (Odd One (x:xs)) = Even (s x) xs
s' (Odd z xs) = Even One (s' z:xs)
```

`s` and `s'` are inverses

Proposition

Denote $Pos^+ = Pos - \{One\}$ The functions $s : Pos \to Pos^+$ and $s' : Pos^+ \to Pos$ are inverses.

Proof.

It follows by structural induction after observing that patterns for `Even` in `s` correspond one by one to patterns for `Odd` in `s'` and vice versa. □

More generally, it can be proved by structural induction that Peano's axioms hold and, as a result, $< Pos, One, s >$ is a Peano algebra.

The log^* worst case complexity of `s` and `s'`

Proposition

The worst case time complexity of the `s` *and* `s'` *operations on n is given by the iterated logarithm* $O(\log_2^*(n))$.

Proof.

Note that calls to `s,s'` in `s` or `s'` happen on terms at most logarithmic in the bitsize of their operands. The recurrence relation counting the worst case number of calls to `s` or `s'` is: $T(n) = T(\log_2(n)) + O(1)$, which solves to $T(n) = O(\log_2^*(n))$. $\square$

The constant average complexity of `s` and `s'`

Proposition

The functions `s` and `s'` work in constant time, on the average.

Proof.

Observe that the average size of a contiguous block of 0s or 1s in a number of bitsize n is asymptotically 2 as $\sum_{k=0}^{n} \frac{1}{2^k} = 2 - \frac{1}{2^n} < 2$. $\square$

While the same average case complexity applies to successor and predecessor operations on ordinary binary numbers, their worst case complexity is $O(\log_2(n))$ rather than the asymptotically much smaller $O(\log_2^*(n))$.

Other $O(log^*)$ worst case and $O(1)$ average operations

Doubling a number `db` and reversing the `db` operation (`hf`) reduce to successor/predecessor operations on logarithmically smaller arguments.

```
db :: Pos -> Pos
db One = Even One []
db (Even x xs) = Even (s x) xs
db (Odd x xs) = Even One (x:xs)

hf :: Pos -> Pos
hf (Even One []) = One
hf (Even One (x:xs)) = Odd x xs
hf (Even x xs) = Even (s' x) xs
```

At most one call to `s, s'` are made in each function.

Constant time operations

based on equation (3) the operation `exp2` (computing an exponent of 2) simply inserts `x` as the first argument of an `Even` term.

```
exp2 :: Pos -> Pos
exp2 x = Even x []
```

Its left-inverse `log2` extracts the argument `x` from an `Even` term.

```
log2 :: Pos -> Pos
log2 (Even x []) = x
```

Proposition

The worst case and average time complexity of `exp2, log2` *is* $O(1)$.

Addition and subtraction

- a (fairly long) chain of mutually recursive functions defines addition and subtraction.
- we want to take advantage of large contiguous blocks of o^n and i^m applications
- we will rely on equations like (1) and (2) governing applications and "un-applications" of such blocks

1. leftshiftBy, rightshiftBy
2. detaching and fusing blocks of similar digits: split, fuse
3. addition and subtraction: add, sub
4. comparison operation: cmp
5. bitsize

Bitsize and tree-size

```
bitsize :: Pos -> Pos
bitsize One = One
bitsize (Even x xs) =  s (foldr add x xs)
bitsize (Odd x xs) =  s (foldr add x xs)

treesize :: Pos -> Pos
treesize One = One
treesize (Even x xs) = foldr add x (map treesize xs)
treesize (Odd x xs) = foldr add x (map treesize xs)
```

Proposition

For all terms $t \in$ `Pos`, `treesize t` $\leq$ `bitsize t`.

Other operations working one o^k or i^k block at a time

1. log_2
2. $\log_2^*$
3. general multiplication: mul

A gcd working one o^k or i^k block at a time

```
gcdiv _ One = One
gcdiv One _ = One
gcdiv a b = f px py where
   (px,x,x') = split a
   (py,y,y') = split b

   f 0 0 = g (cmp x y)
   f 0 1 = gcdiv x' b
   f 1 0 = gcdiv a y'
   f 1 1 = h (cmp a b)

   g LT = fuse (0,x,gcdiv x' (fuse (0,sub y x,y')))
   g EQ = fuse (0,x,gcdiv x' y')
   g GT = fuse (0,y,gcdiv y' (fuse (0,sub x y,x')))

   h LT = gcdiv a (sub b a)
   h EQ = a
   h GT = gcdiv b (sub a b)
```

Easy extension to signed integers

- the data type Z:

```
data Z = Zero | Plus Pos | Minus Pos
```

- the bijection from trees of type Z to bitstring-represented integers is implemented by the function z':

```
z' :: Z -> Integer
z' Zero = 0
z' (Plus x) = p' x
z' (Minus x) = - (p' x)
```

- its inverse is implemented by the function z:

```
z :: Integer -> Z
z 0 = Zero
z k | k>0 = Plus (p k)
z k | k<0 = Minus (p (-k))
```

Efficient `cons` and `decons` operations

```
cons :: (Pos,Pos)->Pos
cons (One,One) = Even One []
cons (Even x xs,One)  =  Odd (hf (Even x xs)  ) []
cons (Odd x xs,One)  = Even (hf (s (Odd x xs)   )) []
cons (x,Even y ys) = Odd x (y:ys)
cons (x,Odd y ys) = Even x (y:ys)

decons :: Pos->(Pos,Pos)
decons (Even x []) = (s' (db x),One)
decons (Even x (y:ys)) = (x,Odd y ys)
decons (Odd x []) = (db x,One)
decons (Odd x (y:ys)) = (x,Even y ys)
```

- `cons` and `decons` are constant time on the average
- and $O(log^*(bitsize))$ in the worst case

An application: compact encodings of sequences and sets

- to/from lists: by iterating `decons` and `cons`
- lists to/from sets: with prefix sums and pairwise differences
- compact representation of sparse sets
- also, compact representation of complements of sparse sets:

```
*EvenOdd> p' (treesize (from_set (map p ([1,3,5]++[6..220]))))
218
*EvenOdd> p' (bitsize (from_set (map p ([1,3,5]++[6..220]))))
221
```

Proposition

These encodings/decodings of lists and sets as Even-Odd Trees are size-proportionate i.e., their representation sizes are within constant factors.

Conclusions and future work

- the arithmetic of Even-Odd Trees provides an alternative to bitstring-based binary numbers that favors numbers with comparatively large contiguous blocks of similar binary digits
- while random numbers with high Kolmogorov complexity do not exhibit this property, applications involving sparse/dense or otherwise regular data frequently do
- besides arithmetic operations favoring such numbers, Even-Odd Trees provide bijective size-proportionate encodings of lists and sets
- future work:
  - parallelization of our algorithms as well as design of some non-recursive alternatives
  - encodings and operations on sparse matrices, graphs, and data structures like quadtrees and octrees
- the paper is a literate program, our Haskell code is at `http://www.cse.unt.edu/~tarau/research/2015/EvenOdd.hs`