The Arithmetic of Even-Odd Trees

***Abstract*—Even-Odd Trees are a canonical tree-based number representation derived from a bijection between trees defined by the data type equation $T = 1 + T * T^* + T * T^*$ and positive natural numbers seen as iterated applications of $o(x) = 2x$ and $i(x) = 2x + 1$ Starting from 1.**

This paper introduces purely functional arithmetic algorithms for operations on Even-Odd Trees.

While within constant factors from their traditional counterparts for their average case behavior, our algorithms make tractable important computations that are impossible with traditional number representations.

***Keywords*—tree-based number representations, arithmetic with giant numbers, unconventional numbering systems, purely functional algorithms**

I. Introduction

Moving from Peano's successor-based unary arithmetic to a binary system not only results in an exponential speed-up but this speed-up applies *fairly* to all numbers independently of their completely random or highly regular structure. Moreover, information theory tells us that we cannot improve on the *average* complexity of arithmetic operations by changing the representation.

Can we do better if we give up this *egalitarian* view? Can we treat better some classes of numbers, to benefit interesting computations with them, without more than constant time prejudice to the others?

This paper introduces arithmetic operations on top of such an "elitist" number system where numbers with a "regular structure" (made of larger alternating blocks of 0s and 1s) receive preferential treatment (up to super-exponential acceleration), while 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. As another somewhat unusual feature, we will first restrict our arithmetic operations to strictly positive numbers and then sketch an extension to signed integers. Finally, our algorithms are presented as purely functional specifications, in a literate programming style, using a small subset of Haskell as an executable mathematical notation.

The paper is organized as follows. Section II introduces our canonical representation of strictly positive natural numbers, Even-Odd Trees. Section III describes algorithms for successor, predecessor and some other low complexity operations on our trees. Section IV introduces algorithms optimizing addition and subtraction for numbers with few large blocks of `0`s and `1`s and related operations. Section V shows how these algorithms are extended to signed integers. Section VI describes size-proportionate encodings of lists, sets in terms of Even-Odd Trees and section VII extends these techniques to term algebras. Section VIII discusses related work and section IX concludes the paper. Several additional arithmetic algorithms on Even-Odd Trees are described in the **Appendix**, as the anonymous submission precludes providing the links to their complete source code.

II. Even-Odd Trees as a canonical representation of positive natural numbers

A. 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. With this representation, one obtains $2 = o(1), 3 = i(1), 4 = o(o(1)), 5 = i(o(1))$ etc. Thus applying o adds a 0 as the lowest digit of the binary representation of a number and applying i adds an 1. And clearly, $i(x) = o(x) + 1$.

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

Some 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. Among them, we start with the following two, showing the connection of our iterated function applications with left shift (multiplication by a power of 2) operations.

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

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

In particular

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

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

Also, one can directly relate o^k and i^k as

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

C. Even-Odd trees as a data type

First we define a data type for our tree represented natural numbers.

Definition 1: 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 $\mathtt{T} = 1 + \mathtt{T} \times \mathtt{T}^* + \mathtt{T} \times \mathtt{T}^*$. We will call *terms* the trees described by the data type `Pos`.

The intuition behind the type `Pos` is the following:

- 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`

Thus a term of the form `Even x xs` would correspond to multiplications by 2^x and a term of the form `Odd x xs` to a similar operation as indicated by the equations (1) and (2). The bijection between the set of binary represented positive natural numbers and Pos is provided by the function p'.

Definition 2: The function $p' : Pos \to \mathbb{N}^+ = N - \{0\}$ shown in equation (**6**) defines the unique natural number associated to a term of type `Pos`.

The corresponding Haskell code uses structural recursion over the patterns provided by the type `Pos`.

```
p' :: Pos → Integer
p' One = 1
p' (Even x []) = 2^(p' x)
p' (Even x (y:xs)) = 2^(p' x)*(p' (Odd y xs))
p' (Odd x []) = 2^(p' x+1)-1
p' (Odd x (y:xs)) = 2^(p' x)*(p' (Even y xs)+1)-1
```

Proposition 1: $p' : Pos \to N^+$ is a bijection, i.e., each term canonically represents the corresponding natural number.

Proof: It follows from the equations (1) and (2) by replacing the power of 2 functions with the corresponding iterated applications of o and i. ■

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

The following examples illustrate the workings of the bijection p':

$20 \to$ `Even (Even One []) [One,One]`
$\to ((2^1+1)2^1-1)2^{2^1}$

$2^{2^{16}}-1 \to$ `Odd (Odd (Odd (Odd One [])[])[])[]`

$\to 2^{2^{2^{2^{1+1}-1+1}-1+1}-1+1}-1$

To implement the inverse $p : \mathbb{N}^+ \to Pos$ 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 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 following example illustrates the first 4 positive numbers represented as Even-Odd Trees.

```
*Arith> mapM_ print (map p [1..4])
One
Even One []
Odd One []
Even (Even One []) []
```

III. O(log^*) WORST CASE AND $O(1)$ AVERAGE OPERATIONS

A. Successor (`s`) and predecessor (`s'`)

We will now specify successor and predecessor on data type `Pos` through two mutually recursive functions

```
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 (pz : xs) where pz = s' z
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:px:xs) where px = s' x
```

```
s' :: Pos → Pos
s' (Even One []) = One
s' (Even z []) = Odd pz [] where pz = 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:px:xs) where px = s' x
s' (Odd One []) = Even One []
s' (Odd One (x:xs)) = Even (s x) xs
s' (Odd z xs) = Even One (pz:xs) where pz = s' z
```

Please note that this definitions are justified by the equations (3), (1), (4) and (2). Note also that termination is guaranteed as both `s` and `s'` use only induction on the structure of the terms.

Proposition 2: 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.

Proposition 3: 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

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

to `s` or `s'` is: $T(n) = T(log_2(n)) + O(1)$, which solves to $T(n) = O(log_2^*(n))$. ■

Proposition 4: The functions `s` and `s'` work in constant time, on the average.

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

Note that 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 $O(log_2^*(n))$.

We will now describe algorithms for basic arithmetic operations that take advantage of our number representation.

B. 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 px xs where px = s' x
```

Proposition 5: The average time complexity of `db, hf` is constant and their worst case time complexity is $O(log_2^*(n))$.

Proof: It follows by observing that at most one call to `s, s'` are made in each function. ■

C. 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 6: The worst case and average time complexity of `exp2, log2` is $O(1)$.

IV. Optimizing addition and subtraction for numbers with few large blocks of 0s and 1s

Next, we will derive addition and subtraction operations *favoring terms with large contiguous blocks of o^n (corresponding to n zeros) and i^n (corresponding to n ones) applications*, on which they will lower complexity so that it depends on the *number of blocks rather than the total number of o and i applications forming the blocks*.

Given the recursive, self-similar structure of our trees, as the algorithms mimic the data structures they operate on, we will have to work with a chain of *mutually recursive* functions. As 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.

A. Left and right shift operations

The functions `leftshiftBy`, `leftshiftBy'` and respectively `leftshiftBy"` correspond to $2^n k$, $(\lambda x.2x + 1)^n(k)$ and $(\lambda x.2x + 2)^n(k)$.

```
leftshiftBy :: Pos→Pos→Pos
leftshiftBy k One = Even k [] -- exp2
leftshiftBy k (Even x xs) = Even (add k x) xs
leftshiftBy k (Odd x xs) = Even k (x:xs)
```

```
leftshiftBy' :: Pos → Pos → Pos
leftshiftBy' x k = s' (leftshiftBy x (s k))
```

```
leftshiftBy'' :: Pos → Pos → Pos
leftshiftBy'' x k = s' (s' (leftshiftBy x (s (s k))))
```

The last two are derived from the identities (2) and (7)

$$(\lambda x.2x + 2)^n(k) = 2^n(k + 2) - 2 \quad (7)$$

They are part of a *chain of mutually recursive functions* as they are already referring to the `add` function, to be implemented later. Note also that instead of naively iterating, they implement a more efficient algorithm, working "one block at a time". For instance, when detecting that its argument counts a number of `1`s, `leftshiftBy'` just increments that count. As a result, the algorithm favors numbers with relatively few large blocks of `0` and `1` digits.

The function `rightshiftBy` goes over its argument `y` one block at a time, by comparing the size of the block and its argument `x` that is decremented after each block by the size of the block. The local function `f` handles the details, irrespectively of the nature of the block, and stops when the argument is exhausted. More precisely, based on the result `EQ, LT, GT` of the comparison, `f` either stops or it calls `rightshiftBy` on the the value of `x` reduced by the size of the block.

```
rightshiftBy :: Pos → Pos → Pos
rightshiftBy k z = f (cmp k x) where
  (b,x,y) = split z
  f LT = fuse (b,sub x k,y)
  f EQ = y
  f GT = rightshiftBy (sub k x) y
```

Note also the use of the functions `split` and `fuse` that facilitate recursing on subtrees "generically" - independently of their specific structure.

```
split :: Pos → (Int, Pos, Pos)
split (Even x []) = (0,x,One)
split (Odd x []) = (1,x,One)
split (Even x (y:ys)) = (0,x,Odd y ys)
split (Odd x (y:ys)) = (1,x,Even y ys)
```

```
fuse :: (Int, Pos, Pos) → Pos
fuse (0,x,One) = Even x []
fuse (1,x,One) = Odd x []
fuse (0,x,Odd y ys) = Even x (y:ys)
fuse (1,x,Even y ys) = Odd x (y:ys)
```

B. Addition

We are now ready to define addition.

```
add :: Pos → Pos → Pos
```

The base cases are:

```
add One y = s y
add x One = s x
```

The general case is handled by the local helper functions `l`, `r` and `f`. In the case when both terms represent even numbers, the two blocks add up to an even block of the same size. Depending on the parity of the two operands, the functions `l` and `r` select the appropriate shift operation to apply on the two sub-components of the recursion in function `f`. Based on the ordering of the operands corresponding to the sizes of the first block of 0s or 1s, `f` advances over as many 0s or 1s as possible, in one step. So he function `f` ensures that, at each step, a block of the same size is extracted, depending on which of the two operands `a` or `b` is larger.

```
add x y = f c where

  c = cmp a b

  (p1,a,as) = split x
  (p2,b,bs) = split y

  l 0 0 = leftshiftBy
  l 0 1 = leftshiftBy'
  l 1 0 = leftshiftBy'
  l 1 1 = leftshiftBy''

  r _ 0 0 = leftshiftBy
  r GT 0 1 = leftshiftBy
  r LT 1 0 = leftshiftBy
  r _ _ _ = leftshiftBy'

  f EQ = l p1 p2 a (add as bs)
  f GT = l p1 p2 b (add (r GT p1 p2 (sub a b) as) bs)
  f LT = l p1 p2 a (add as (r LT p1 p2 (sub b a) bs))
```

Note the presence of the comparison operation `cmp`, to be defined later, also part of our chain of mutually recursive operations.

C. Subtraction

Subtraction, in the absence of 0 will will only be defined when the first operand is strictly larger than the second. Note that extension to signed integers and 0 is straightforward. Otherwise, the code for the subtraction function `sub` is similar to that for `add`:

```
sub :: Pos → Pos → Pos
sub x One = s' x
sub x y = f p1 p2 (cmp a b) where

  (p1,a,as) = split x
  (p2,b,bs) = split y

  l 0 0 = leftshiftBy
  l 1 1 = leftshiftBy

  r 0 0 = leftshiftBy
  r 1 1 = leftshiftBy'

  f 1 0 _  = s' (sub (s x) y)
  f 0 1 _  = s' (sub x (s' y))

  f _ _ EQ = l p1 p2 a (sub as bs)
  f _ _ GT = l p1 p2 b (sub (r p1 p2 (sub a b) as) bs)
  f _ _ LT = l p1 p2 a (sub as (r p1 p2 (sub b a) bs))
```

Like in the case of `add`, the function `f` applies, depending on parity, the appropriate shift operation `r` to recurse on the remaining blocks of digits and `l` to merge it with the result of the operations on the first block.

Note that these algorithms collapse to the ordinary binary addition and subtraction most of the time, given that the average size of a block of contiguous `0s` or `1s` is `2` bits (as shown in Prop. III-A), so their average performance is within a constant factor of their ordinary counterparts. On the other hand, the algorithms favor deeper trees made of large blocks, representing giant "towers of exponents"-like numbers by working (recursively) one block at a time rather than 1 bit at a time, resulting in possibly super-exponential gains.

D. Comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our type `Pos`. It relies on `bitsize` computing the number of binary digits corresponding to a term in `Pos`. Note that `bitsize` is part of our mutually recursive functions, to be defined later.

We first observe that only terms of the same bitsize need detailed comparison, otherwise the relation between their bitsizes is enough, *recursively*. More precisely, the following holds:

Proposition 7: Let `bitsize` count the number of binary digits of a strictly positive natural number. Then $\texttt{bitsize}(x) < \texttt{bitsize}(y) \Rightarrow x < y$.

Proof: Observe that their lexicographic enumeration ensures that the bitsize of the underlying binary numbers is a non-decreasing function. ■

The comparison operation proceeds one block at a time, and it also takes some inferential shortcuts, when possible.

```
cmp :: Pos → Pos → Ordering
cmp One One = EQ
cmp One _ = LT
cmp _ One = GT
cmp x y | x′ /= y′  = cmp x′ y′ where
  x′ = bitsize x
  y′ = bitsize y
cmp x y = revCmp x′ y′ where
  x′ = revOrd x
  y′ = revOrd y
```

The function `revCmp` compares two terms known to have the same `bitsize`. It works on reversed (highest order digit first) variants, computed by `revOrd` and it takes advantage of the block structure using the following fact.

Proposition 8: Assuming two terms of the same bitsizes, the one with its highest order `Odd` subterm is larger than the one with highest order subterm `Even`.

Proof: Observe that the corresponding binary numbers seen as "highest order digit first" are lexicographically ordered with $0 < 1$. ■

As a consequence, `cmp` only recurses when *identical* blocks lead the sequence of blocks, otherwise it infers the `LT` or `GT` relation. The nontrivial cases are when parity is the same, in which case the one with the larger block of $1s$ prevails. In the two cases, recursion proceeds further when the blocks are equal.

```
revCmp :: Pos → Pos → Ordering
revCmp One One = EQ
revCmp (Even x xs) (Even y ys) = f (cmp x y) where
    f EQ | xs==[] && ys == [] = EQ
    f EQ = revCmp (Odd x′ xs′) (Odd y′ ys′) where
       (x′:xs′) = xs
       (y′:ys′) = ys
    f LT = GT
    f GT = LT
revCmp (Odd x xs) (Odd y ys) = f (cmp x y) where
    f EQ | xs==[] && ys == [] = EQ
    f EQ = revCmp (Even x′ xs′) (Even y′ ys′) where
       x′:xs′ = xs
       y′:ys′ = ys
    f LT = LT
    f GT = GT
```

Comparison is clear between blocks of 0s and 1s - note again that this happens after bringing the largest block to the front.

```
revCmp (Even _ _) (Odd _ _) = LT
revCmp (Odd _ _) (Even _ _) = GT
```

Thus the effort for reversing the order of digits is proportional to the number of blocks, significantly fewer that the bitsize for numbers with a uniform structure.

```
revOrd :: Pos → Pos
revOrd One = One
revOrd (Even x xs) = revPar (Even y ys) where (y:ys) =
reverse (x:xs)
revOrd (Odd x xs) = revPar (Odd y ys) where (y:ys) =
reverse (x:xs)
```

Inferring parity of the reversed number also happens in time proportional to the number of blocks. It only involves counting the blocks – as their parities alternate.

```
revPar :: Pos → Pos
revPar (Even x xs) | even (length xs)= Even x xs
revPar (Odd x xs) | even (length xs)= Odd x xs
revPar (Even x xs) = Odd x xs
revPar (Odd x xs) = Even x xs
```

E. Bitsize and tree-size

The function `bitsize` computes the number of binary digits. It works by summing up (using Haskell's `foldr`) the counts of `0` and `1` digit blocks composing a tree-represented natural number.

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

Note that `bitsize` concludes the set of mutually recursive operations helping with the definition of addition and subtraction.

The function `treesize` computes the size of the tree corresponding to a positive natural number in terms of its number of nodes.

```
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 9: For all terms $t \in$ `Cat`, `treesize t` $\leq$ `bitsize t`.

Proof: By induction on the structure of t, observing that the two functions have similar definitions and corresponding calls to `treesize` return terms inductively assumed smaller than those of `bitsize`. ■

As a consequence, complexity of various arithmetic algorithms operating on our trees is *within a constant factor* of their binary number counterparts.

V. Extension to integers

Extension to integers happens naturally, by providing also a canonical representation in the form of the data type `Z` that simply adds `Zero` and marks elements of `Pos` with `Plus` or `Minus` signs.

```
data Z = Zero | Plus Pos | Minus Pos
  deriving (Eq, Show, Read)
```

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))
```

The following example illustrates its work:

```
*Arith> mapM_ print (map z [-3..3])
Minus (Odd One [])
Minus (Even One [])
Minus One
Zero
Plus One
Plus (Even One [])
Plus (Odd One [])
```

The `opposite` operation is simply flipping signs:

```
opposite Zero = Zero
opposite (Plus x) = Minus x
opposite (Minus x) = Plus x
```

The successor function `sZ` and its inverse `sZ'` use successor `s` on `Pos` or its inverse `s'` depending on the sign.

```
sZ :: Z → Z
sZ Zero = Plus One
sZ (Plus x) = Plus y where  y = s x
sZ (Minus One) = Zero
sZ (Minus x) = Minus y where y = s' x

sZ' :: Z → Z
sZ' Zero = Minus One
sZ' (Minus x) = Minus y where y = s x
sZ' (Plus One) = Zero
sZ' (Plus x) = Plus y where y = s' x
```

Note that both functions are *total* on `Z` and work as expected:

```
*Arith> map (z'.sZ'.sZ.z) [-5..5]
[-5,-4,-3,-2,-1,0,1,2,3,4,5]
```

VI. Representing sequences and sets

We will start by describing bijective mappings between *collection* types. *Our goal is to show that natural number encodings for sparse or dense (seen as complements of sparse) instances of these collections will have space-efficient representations as natural numbers represented as trees of type* `Pos`*, in contrast to bitstring-based representations.*

A. Bijections between strictly positive naturals represented as Even-Odd Trees and pairs

We will first convert between natural numbers and lists, by using the bijection between numbers and pairs corresponding to the function `cons`, which extends the second argument of the pair with a block of odd/even digits encoded as its first argument and adjusts parity as needed.

```
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)
```

The function `decons` inverts `cons` to a Haskell ordered pair by "taking out the first block of digits" and adjusting parity as needed.

```
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)
```

Proposition 10: The operations `cons` and `decons` are constant time on the average and $O(log^*(bitsize))$ in the worst case, where log^* is the iterated logarithm function, counting how many times log can be applied before reaching 0.

Proof: Observe that a constant number of `s` and `s'` operations is used in each branch of `cons` and `decons`, therefore the worst case and average complexity of `cons` and `decons` are also the same as that of `s` and `s'`. ■

B. The bijection between natural numbers and lists

The bijection between natural numbers and lists of natural numbers `to_list` and its inverse `from_list` apply repeatedly `decons` and `cons`.

```
to_list :: Pos → [Pos]
to_list One = []
to_list z = x : to_list y where (x,y) = decons z
```

```
from_list :: [Pos] → Pos
from_list [] = One
from_list (x:xs) = cons (x,from_list xs)
```

C. The Bijection between sequences and sets

Increasing sequences provide a canonical representation for sets of natural numbers. While finite sets and finite lists of elements of $\mathbb{N}$ share a common representation `[N]`, sets are subject to the implicit constraint that their ordering is immaterial. This suggest that a set like $[4,1,9,3]$ could be represented canonically as a sequence by first ordering it as $[1,3,4,9]$ and then computing the differences between consecutive elements i.e. $[x_0, x_1 \ldots x_i, x_{i+1} \ldots] \rightarrow [x_0, x_1 - x_0, \ldots x_{i+1} - x_i \ldots]$. This gives $[1,2,1,5]$, with the first element 1 followed by the increments $[2,1,5]$

Therefore, *incremental sums*, computed with Haskell's `scanl`, are used to transform arbitrary lists to canonically represented sets of natural numbers, inverted by *pairwise differences* computed using `zipWith`.

```
list2set, set2list :: [Pos] → [Pos]

list2set [] = []
list2set (n:ns) = scanl add n ns

set2list [] = []
set2list (m:ms) = m : zipWith sub ms (m:ms)
```

D. The bijections between natural numbers and sets of natural numbers represented as Even-Odd Trees

By composing with natural number-to-list bijections, we obtain bijections to sets of natural numbers.

```
to_set :: Pos → [Pos]
to_set = list2set . to_list

from_set :: [Pos] → Pos
from_set = from_list . set2list
```

As the following example shows, trees of type `Pos` offer a significantly more compact representation of sparse sets than conventional binary numbers.

```
*Arith> p' (bitsize (from_set
   (map p [42,1234,6789])))
4013
*Arith> p' (treesize
   (from_set (map p [42,1234,6789])))
60
```

Note that a similar compression occurs for sets of natural numbers with only a few elements missing (that we call *dense sets*), as they have the same representation size as their sparse complement.

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

The following holds:

Proposition 11: These encodings/decodings of lists and sets as Even-Odd Trees are size-proportionate i.e., their representation sizes are within constant factors.

VII. Bijective size-proportionate encoding of term algebras

Term algebras can be seen as the underlying data type shared by a large number of widely used concepts ranging from terms used in logic programming system and proof assistants to syntax trees and XML files.

As shown in in [1], devising an encoding (also called a "Gödel numbering") for term algebras that is both size-proportionate and bijective is a difficult task, when the traditional bitstring-based number representation is used. It involves a fairly sophisticated ranking/unranking algorithm for Catalan families [2], in combination with a generalization of Cantor's pairing function to tuples (see [3], [4]), the only known polynomial bijection between $\mathbb{N}$ and sequences of natural numbers.

The solution to the same problem using Even-Odd Trees as a representation of natural numbers is strikingly simple. *The basic intuition is that we avoid exponential blow-up as we are mapping trees to trees rather than to strings of bits.*

We start by defining the data type `Term` hosting a generic term algebra with countable many variables `Var`, constants `Const` and function symbols `Fun`.

```
data Term a = Var a | Const a | Fun a [Term a]
  deriving (Eq,Show,Read)
```

The bijection to positive natural numbers represented as Even-Odd Trees uses them also to associate names to variables, constants and function symbols. This ensures that the representation are *size-proportionate* even if codes for variables or constant are very large numbers.

```
toTerm :: Pos → Term Pos
toTerm One = Var One
toTerm (Odd x []) = Var (s x)
toTerm (Even x []) = Const x
toTerm (Odd x (y:xs)) = Fun (s' (db x))
  (map toTerm (y:xs))
toTerm (Even x (y:xs)) = Fun (db x) (map toTerm (y:xs))
```

Note that parity is used to uniquely encode various data types. The decoding reverses the process. As they are based on "tree-to-tree mappings", both functions work in time proportional to their inputs.

```
fromTerm :: Term Pos → Pos
fromTerm (Var One) = One
fromTerm (Var y) = Odd (s' y) []
fromTerm (Const x) = Even x []
fromTerm (Fun One xs) = Odd One (map fromTerm xs)
fromTerm (Fun x@(Odd _ _) xs) =
Odd (hf (s x)) (map fromTerm xs)
fromTerm (Fun x@(Even _ _) xs) =
Even (hf x) (map fromTerm xs)
```

The following examples illustrate that this bijection is indeed *size-proportionate* when numbers are represented as Even-Odd Trees, but it would overflow the computer's memory as the corresponding bitstring-represented 2^{1027}-bit number.

```
*Arith> fromTerm (Fun One [Fun One
  [Fun One [Fun One [Const One]]]])
Odd One [Odd One [Odd One
  [Odd One [Even One []]]]]
*Arith> p' (ilog2 (ilog2 it))
1027
*Arith> toTerm (Odd One [Odd One
  [Odd One [Odd One [Even One []]]]])
Fun One [Fun One [Fun One
  [Fun One [Const One]]]]
```

Note also that we have used trees of type `Term Pos` rather than the more obvious type `Term Integer` to ensure that the encoding is size proportionate both ways. Otherwise, if using the type `Term Integer`, numbering variables, constants and function symbols could explode when converted from an tree of type `Pos` that, for instance, contains as subterm a tower exponents.

VIII. Related work

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [5] (a notation for towers of exponents). In contrast to our tree-based natural numbers, such notations are not closed under addition and multiplication, and consequently they cannot be used as a replacement for ordinary binary or decimal numbers.

The first instance of a tree-based *hereditary number system* that we are aware of, occurs in the proof of Goodstein's theorem [6], 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.

Another hereditary number system is Knuth's TCALC program [7] that decomposes $n = 2^a + b$ with $0 \leq b < 2^a$ and then recurses on a and b with the same decomposition. Given the constraint on a and b, while hereditary, the TCALC system is not based on a bijection between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ and therefore not every tree built with it corresponds to a number. Moreover, the literate C-program that defines it only implements successor, addition, comparison and multiplication and does not provide a similar constant time exponent of 2 and low complexity leftshift / rightshift operations, like our tree representation does.

In [8] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. They are based on the decomposition of a natural number $n = g + x_p d$ where $x_p = 2^{2^p}$, $0 < g < x_p$ and $0 < d < x_p$. Like the representations used in this paper, they can also be used as a compact representation of sets. However in contrast to the tree representation proposed in this paper, they only compress "sparse" numbers and sets, consisting of relatively few `1` bits in their binary representation.

Note also that both [7] and [8] use representations that are "additive" while the one described in this paper is "multiplicative"(i.e.; components are glued together by multiplications rather than additions). This enables it to naturally scale up to supporting efficient;y (tower's of) exponents of 2 operations.

Interestingly, arithmetic packages similar to our strictly positive natural numbers view of arithmetic operations, are part of libraries of proof assistants like Coq [9], where they are used as the first layer on top of which natural numbers including 0 and then signed integers are built. Also, as a historical note, Peano's original axioms for natural numbers start with 1 rather than 0.

Much closer to the current paper are [10] and [11] that both describe arithmetic computations with hereditary numbering systems represented as trees. In [10] a bijective base-2 representation starting from 0 and iterated application of $o(x) = 2x + 1$ and $i(x) = 2x + 2$ are used. While worst-case and average case complexity of arithmetic operations is similar to the ones in this paper, working with strictly positive numbers allows us to recover the well known logic of ordinary binary numbers, facilitates extension to signed integers and simplifies and speeds up most of the algorithms. The data structure used in [11], on the other hand, recovers the use of ordinary binary numbers (starting from 0) and brings some simplification to most algorithms. However, the parity information in [11] is inferred from the number of subtrees of a tree and us such the actual performance of the implementation in a purely functional style is significantly slower than [10].

In a way, by rebuilding a purely functional representation similar to [10] on top of strictly positive natural numbers as done in this paper, merges together the reliance on ordinary binary representation introduced in [11] with the ability to work in a purely functional framework introduced in [10].

IX. Conclusion 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. Among them, we note encodings and operations on sparse matrices, graphs, and data structures like quadtrees and octrees.

Besides arithmetic operations favoring such numbers, Even-Odd Trees provide bijective size-proportionate encodings of lists, sets and term algebras. Such encodings can be easily extended to other data types and have applications to generation of random instances of a desired size, with uses for automated software testing.

Future work will focus on parallelization of our algorithms as well as design of some non-recursive alternatives. With possible FPGA realization this is likely to help turning our tree-based arithmetic into a practical tool for specialized high-performance computations with very large numbers.

Acknowledgement

This research is supported by NSF grant ########.

References

[1] P. Tarau, "Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) ," *Theory and Practice of Logic Programming*, vol. 13, no. 4-5, pp. 847–861, 2013.

[2] R. P. Stanley, *Enumerative Combinatorics*. Belmont, CA, USA: Wadsworth Publ. Co., 1986.

[3] P. Cegielski and D. Richard, "On arithmetical first-order theories allowing encoding and decoding of lists," *Theoretical Computer Science*, vol. 222, no. 1-2, pp. 55–75, 1999.

[4] M. Lisi, "Some remarks on the Cantor pairing function," *Le Matematiche*, vol. 62, no. 1, 2007. [Online]. Available: http://www.dmi.unict.it/ojs/index.php/lematematiche/article/view/14

[5] D. E. Knuth, "Mathematics and Computer Science: Coping with Finiteness," *Science*, vol. 194, no. 4271, pp. 1235 –1242, 1976.

[6] R. Goodstein, "On the restricted ordinal theorem," *Journal of Symbolic Logic*, no. 9, pp. 33–41, 1944.

[7] D. E. Knuth, "TCALC program," Dec. 1994, http://www-cs-faculty.stanford.edu/~uno/programs/tcalc.w.gz.

[8] J. Vuillemin, "Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, Jun. 2009, pp. 7 –14.

[9] The Coq development team, *The Coq proof assistant reference manual*, 2014, version 8.4pl4. [Online]. Available: http://coq.inria.fr

[10] P. Tarau, " Bijective Collection Encodings and Boolean Operations with Hereditarily Binary Natural Numbers," in *PPDP '14: Proceedings of the 16th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: ACM, 2014.

[11] ——, "The Arithmetic of Recursively Run-length Compressed Natural Numbers," in *Proceedings of 8th International Colloquium on Theoretical Aspects of Computing, ICTAC 2014*, G. Ciobanu and D. Méry, Eds. Bucharest, Romania: Springer, LNCS 8687, Sep. 2014, pp. 406–423.

[12] R. P. Brent and P. Zimmernann, *Modern Computer Arithmetic*. Cambridge, UK: Cambridge University Press, 2011.

APPENDIX

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

1) Integer logarithm in base 2: The base-2 integer logarithm is computed as

```
ilog2 :: Pos → Pos
ilog2 = bitsize . s'
```

The iterated logarithm log_2^* can be also defined as

```
ilog2star :: Pos → Pos
ilog2star One = One
ilog2star x = s (ilog2star (ilog2 x))
```

2) General multiplication: This example illustrates that our arithmetic operations are not limited by the size of their operands, but only by the size of the tree representation of number.

Note that, by contrast to Karatsuba or Toom-Cook [12] multiplication algorithms (that improve on the $O(n^2)$ complexity of traditional multiplication by lowering the exponent of n), our algorithm focuses on optimizing the case of operands containing large blocks of similar binary digits for, which it can show exponential speed-ups. On the other hand, on the average, its performance is the same as traditional multiplication's $O(n^2)$.

When the first term represents an even number we apply the `leftshiftBy` operation and we reduce the other case to this one.

```
mul :: Pos → Pos → Pos
mul x y = f (cmp x y) where
  f GT = mul1 y x
  f _ = mul1 x y

  mul1 One x = x
  mul1 (Even x []) y = leftshiftBy x y
  mul1 (Even x (x':xs)) y =
     leftshiftBy x (mul1 (Odd x' xs) y)
  mul1 x y  = add y (mul1 (s' x) y)
```

Note that when the operands are composed of large blocks of alternating `0` and `1` digits, the algorithm is quite efficient as it works (roughly) in time depending on the the number of blocks in its first argument rather than the the number of digits. The following example illustrates a blend of arithmetic operations benefiting from complexity reductions on giant tree-represented numbers:

```
*Arith> let term1 =
  sub (exp2 (exp2 (p 12345))) (exp2 (p 6789))
*Arith> let term2 =
  add (exp2 (exp2 (p 123))) (exp2 (p 456789))
*Arith> bitsize (bitsize (mul term1 term2))
Even One [One,One,Odd One [],Even One [One],One]
*Arith> p' it
12346
```

This hints toward a possibly new computational paradigm where arithmetic operations are not limited by the size of their operands, but only by their representation complexity.

We first specialize our multiplication for a faster squaring operation, using the identity:

$$(i^n(a))^2 = i^{2n}(a^2 + 2a) - 2i^n(a) \tag{8}$$

```
square :: Pos → Pos
square One = One
square z = f b y where
  (b,x,y) = split z
  f 0 y = fuse (0,db x,square y)
  f 1 One = sub (exp2 (db (s x))) (exp2' (s x))
  f 1 y = r where
     k = add (square y) (db y)
     m = fuse (1,db x,k)
     r = sub m (db z)
```

Note that, based on equation (4), the constant time $2^{x+1} - 1$ operation `exp2'` used by `square` can be defined as follows:

```
exp2' :: Pos → Pos
exp2' x = Odd x []
```

We can implement a simple but efficient "power by squaring" operation x^y as follows:

```
superSquare One a = square a
superSquare x a = square (superSquare (s' x) a)

pow a One = a
pow a (Even x []) = superSquare x a
pow a (Even x (x':xs)) =
  pow (superSquare x a) (Odd x' xs)
pow a x  = mul a (pow a (s' x))
```

It works well with fairly large numbers, by also benefiting from efficiency of multiplication on terms with large blocks of o^n and i^m applications:

```
*Arith> p' (ilog2star (pow
    (add (exp2 (p 50)) (exp2 (p 30)))
    (p 60)))
6
```

4) Greatest common divisor: Interestingly, the well-known binary GCD algorithm is naturally generalized to work "one block at 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)
```

This results in significantly fewer steps for numbers built of large blocks of 0s and 1s.

The following examples illustrate the correct behavior of these operations.

```
*Arith> p' (ilog2 (p 100))
7
*Arith> p' (ilog2star (p 1000000))
6
*Arith> p' (gcdiv (p 360) (p 1000))
40
*Arith> p' (gcdiv (p 33) (p 26))
1
```

B. Extending other operations to signed integers

Addition is extended to `Z` by case analysis:

```
addZ :: Z → Z → Z
addZ Zero x = x
addZ x Zero = x
addZ (Plus x) (Plus y) = Plus (add x y)
addZ (Minus x) (Minus y) = Minus (add x y)
addZ (Plus x) (Minus y) = f (cmp x y) where
  f EQ = Zero
  f LT = Minus (sub y x)
  f GT = Plus (sub x y)
addZ (Minus y) (Plus x) = f (cmp x y) where
  f EQ = Zero
  f LT = Minus (sub y x)
  f GT = Plus (sub x y)
```

Subtraction is derived as addition with `opposite`;

```
subZ :: Z → Z → Z
subZ x y = addZ x (opposite y)
```

As usual, comparison on `Z` is derived by case analysis.

```
cmpZ :: Z → Z → Ordering
cmpZ Zero Zero = EQ
cmpZ Zero (Plus _) = LT
cmpZ Zero (Minus _) = GT
cmpZ (Minus _) Zero = LT
cmpZ (Plus _) Zero = GT
cmpZ (Minus x) (Minus y) = cmp y x
cmpZ (Plus x) (Plus y) = cmp x y
cmpZ (Minus _) (Plus _) = LT
cmpZ (Plus _) (Minus _) = GT
```

A generalized shift operation handles left and right shifting depending on the sign of its first argument.

```
shiftBy :: Z → Z → Z
shiftBy Zero x = x
shiftBy (Plus x) (Plus y) = Plus (leftshiftBy x y)
shiftBy (Minus x) (Plus y) = Plus (rightshiftBy x y)
shiftBy (Plus x) (Minus y) = Minus (leftshiftBy x y)
shiftBy (Minus x) (Minus y) = Minus (rightshiftBy x y)
```

```
mulZ Zero _ = Zero
mulZ _ Zero = Zero
mulZ (Plus x) (Plus y) = Plus (mul x y)
mulZ (Plus x) (Minus y) = Minus (mul x y)
mulZ (Minus x) (Plus y) = Minus (mul x y)
mulZ (Minus x) (Minus y) = Plus (mul x y)
```

The following examples illustrate the correct behavior of these operations.

```
*Arith> z' (shiftBy (z (-3)) (z 1001))
125
*Arith> z' (shiftBy (z 3) (z 1001))
8008
```

To implement division on signed integers, we first define division with a strictly positive natural number.

```
divWithPos :: Z → Pos → (Z, Z)
divWithPos n One = (n,Zero)
divWithPos n k = divStep aligned Zero n where
    d = Plus k

    up Zero = Zero
    up (Plus x) = Plus (db x)

    down Zero = Zero
    down (Plus x@(Even _ _)) = Plus (hf x)
    down (Plus x@(Odd _ _)) = Plus (hf (s' x))

    aligned = down (until over up d)

    over x = cmpZ x n == GT

    divStep t q r | cmpZ t d == LT    = (q, r)
    divStep t q r | cmpZ t r /= GT  =
       divStep (down t) (sZ (up q )) (subZ r t)
    divStep t q r = divStep (down t) (up q)     r
```

Next, division with reminder is implemented by case analysis.

```
div_and_rem :: Z → Z → (Z, Z)
div_and_rem Zero x | x/=Zero = (Zero,Zero)
div_and_rem (Plus x) (Plus y) = divWithPos (Plus x) y
div_and_rem (Minus x) (Minus y) = (a,opposite b) where
  (a,b)=divWithPos (Plus x) y
div_and_rem (Plus x) (Minus y) =
f (divWithPos (Plus x) y) where
  f (q,Zero)= (opposite q,Zero)
  f (q,Plus r) = (opposite q, Plus r)
div_and_rem (Minus x) (Plus y) =
f (divWithPos (Plus x) y) where
  f (q,Zero)= (opposite q,Zero)
  f (q,Plus r) = (opposite q,Minus r)

divide n m = fst (div_and_rem n m)

remainder n m = snd (div_and_rem n m)
```

The following examples illustrate the correct behavior of these operations.

```
*Arith> z' (divide (z 207) (z 20))
10
*Arith> z' (remainder (z 207) (z 20))
7
```

C. DAG representations

A more compact representation is obtained by folding together shared nodes in one or more Even-Odd Trees. Assuming that an immutable data structure is used for this representation,

significant memory savings are possible as frequently occurring small subtrees populate the lower levels of these trees.

Fig. 1 shows the DAG representation of the largest prime number known so far. Displaying the corresponding binary integer representation would be impractical.

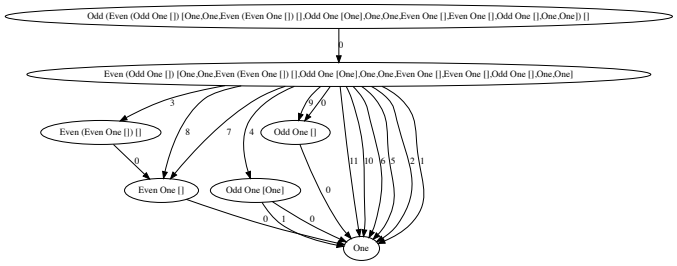

Fig. 1: Largest known prime number discovered in January 2013: the Mersenne prime $2^{57885161} - 1$, represented as a DAG

However, we can show the representation tagged with the intermediate positive or signed integer nodes as shown in Figs. 2, 3, 4 and 5.

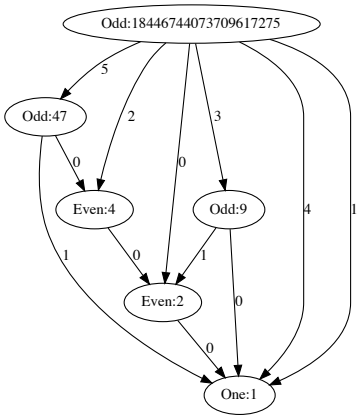

Fig. 2: $2^{64} + 2^{16} + 123$ represented as a DAG

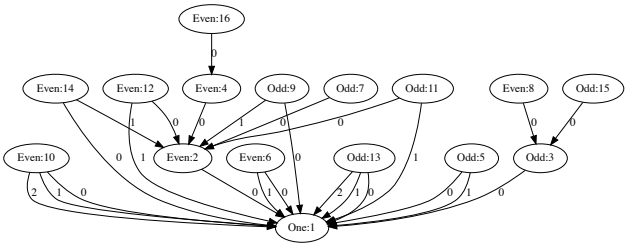

Fig. 3: The first 16 strictly positive naturals represented as a DAG

Note that in all these figures, order is indicated as labels on the edges. Given that Even-Odd trees are rooted ordered trees, this order is relevant.

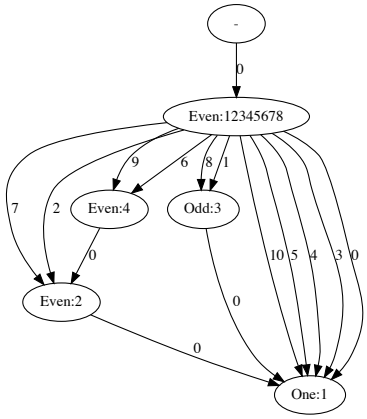

Fig. 4: The signed integer -12345678 represented as a DAG

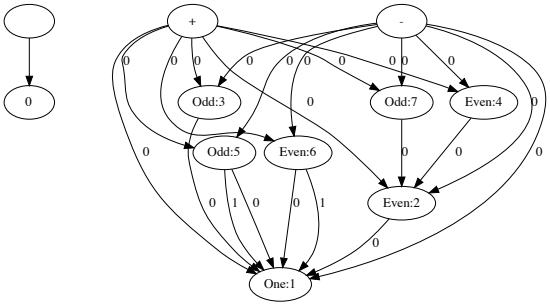

Fig. 5: Signed integers from -7 to 7 represented as a DAG