Arithmetic Algorithms for Hereditarily Binary Natural Numbers

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
tarau@cse.unt.edu

Bill Buckles
Department of Computer Science and Engineering
University of North Texas
Bill.Buckles@unt.edu

ABSTRACT

In a typed functional language we specify a new tree-based number representation, *hereditarily binary numbers*, defined by applying recursively run-length encoding of bijective base-2 digits.

Hereditarily binary numbers support arithmetic operations that are limited by the structural complexity of their operands, rather than their bitsizes.

As a result, they enable new algorithms that, in the best case, collapse the complexity of arithmetic computations by super-exponential factors and in the worst case are within a constant factor of their traditional counterparts.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*Data types and structures*

General Terms

Algorithms, Languages, Theory

Keywords

arithmetic computations with giant numbers, hereditary numbering systems, declarative specification of algorithms, compressed number representations.

1. INTRODUCTION

Number representations have evolved over time from the unary "cave man" system where one scratch on the wall represented a unit, to the base-n (and in particular base-2) number system, with the remarkable benefit of a logarithmic representation size. Over the last 1000 years, this base-n representation has proved to be unusually resilient, partly because all practical computations could be performed with reasonable efficiency within the notation.

However when thinking about the next evolutionary steps, arithmetic computations that are intractable with the current bitstring / word-array representation are likely to be needed in fields such as number theory or cryptography. While *notations* like Knuth's "up-arrow" [3] are useful in describing some very large numbers, they do not provide the ability to actually *compute* with them – as, for instance, addition or multiplication with a natural number results in a number that no longer can be expressed with the notation.

The novel contribution of this paper is a tree-based numbering system that *allows computations* with numbers comparable in size with Knuth's "arrow-up" notation. Moreover, these computations have a worst case complexity that is comparable with the traditional binary numbers, while their best case complexity outperforms binary numbers by an arbitrary tower of exponents factor. Simple operations like successor, multiplication by 2, exponent of 2 are constant time on the average and a number of other operations benefit from significant best case complexity reductions.

For the curious reader, it is basically a *hereditary number system* based on recursively applied *run-length* compression of a special (bijective) binary digit notation.

To make our results easy to validate, we have adopted a *literate programming* style, i.e. the code contained in the paper forms a self-contained Haskell module (tested with ghc 7.6.3) also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/hbin.hs` .

The paper is organized as follows. Section 2 gives some background on bijective base-2 numbers and iterated function applications. Section 3 introduces our tree representation for *hereditarily binary numbers*. Section 4 describes successor and predecessor operations on our tree-represented numbers. Section 5 shows an emulation of bijective base-2 with hereditarily binary numbers and section 6 describes novel algorithms for arithmetic operations taking advantage of our number representation. Section 7 defines a concept of *structural complexity* and studies best and worst cases. Section 8 discusses related work. Section 9 concludes the paper and discusses future work.

2. NATURAL NUMBERS SEEN AS ITERATED FUNCTION APPLICATIONS

Natural numbers can be seen as represented by iterated applications of the functions $o(x) = 2x+1$ and $i(x) = 2x+2$ corresponding to the so called *bijective base-2* representation [5] together with the convention that 0 is represented as the empty sequence. As each $n \in \mathbb{N}$ can be seen as a unique composition of these functions, we can make this precise as follows:

DEFINITION 1. *We call bijective base-2 representation of*

SAC'14, March 24-28, 2014, Gyeongju, Korea
 978-1-4503-2469-4/14/03 ...$15.00.

$n \in \mathbb{N}$ the unique sequence of applications of functions o and i to the empty sequence (denoted ϵ) that evaluates to n.

With this representation, and denoting the empty sequence ϵ, one obtains $0 = \epsilon, 1 = o(\epsilon), 2 = i(\epsilon), 3 = o(o(\epsilon)), 4 = i(o(\epsilon)), 5 = o(i(\epsilon))$ etc. and the following holds:

$$i(x) = o(x) + 1 \tag{1}$$

2.1 Properties of the iterated functions o^n and i^n

PROPOSITION 1. *Let f^n denote application of function f n times. Let $o(x) = 2x + 1$ and $i(x) = 2x + 2$, $s(x) = x + 1$ and $s'(x) = x - 1$. Then $k > 0 \Rightarrow s(o^n(s'(k)) = k2^n$ and $k > 1 \Rightarrow s(s(i^n(s'(s'(k)))) = k2^n$. In particular, $s(o^n(0)) = 2^n$ and $s(s(i^n(0))) = 2^{n+1}$.*

PROOF. By induction. Observe that for $n = 0, k > 0$, $s(o^0(s'(k))) = k2^0$ because $s(s'(k)) = k$. Suppose that $P(n) : k > 0 \Rightarrow s(o^n(s'(k))) = k2^n$ holds. Then, P(n+1) follows, given that $k > 0 \Rightarrow s(o^{n+1}(s'(k))) = s(o^n(o(s'(k)))) = s(o^n(s'(2k))) = 2k2^n = k2^{n+1}$. Similarly, the second part of the proposition also follows by induction on n. □

The underlying arithmetic identities are:

$$k > 0 \Rightarrow 1 + o^n(k-1) = 2^n k \tag{2}$$

$$k > 1 \Rightarrow 2 + i^n(k-2) = 2^n k \tag{3}$$

from where one can deduce

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

$$i^n(k) = 2^n(k+2) - 2 \tag{5}$$

and in particular

$$o^n(0) = 2^n - 1 \tag{6}$$

$$i^n(0) = 2^{n+1} - 2 \tag{7}$$

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

$$o^n(k) + 2^n = i^n(k) + 1 \tag{8}$$

$$i^n(k) = o^n(k) + o^n(0) \tag{9}$$

$$o^n(k+1) = i^n(k) + 1 \tag{10}$$

2.2 The iterated functions o^n, i^n and their conjugacy results

Results from the theory of *iterated functions* apply to our operations. The following proposition is proven in [9]:

PROPOSITION 2. *If $f(x) = ax + b$ with $a \neq 1$, let x_0 be such that $f(x_0) = x_0$ i.e. $x_0 = \frac{b}{1-a}$. Then $f^n(x) = x_0 + (x - x_0)a^n$.*

For $a = 2, b = 1$ and respectively $a = 2, b = 2$ this provides an alternative proof for proposition 1.

A few properties similar to *topological conjugation* apply to our functions

DEFINITION 2. *We call two functions f, g conjugates through h if h is a bijection such that $g = h^{-1} \circ f \circ h$, where $\circ$ denotes function composition.*

PROPOSITION 3. *If f, g are conjugates through h then f^n and g^n are too, i.e. $g^n = h^{-1} \circ f^n \circ h$.*

PROOF. By induction, using the fact that $h^{-1} \circ h = \lambda x.x = h \circ h^{-1}$. □

PROPOSITION 4. *o^n and i^n are conjugates with respect to s and s', i.e. the following 2 identities hold:*

$$o^n(k) = s(i^n(s'(k))) \tag{11}$$

$$i^n(k) = s'(o^n(s(k))) \tag{12}$$

PROOF. An immediate consequence of $i = s \circ o$ (by 1) and Prop. 3. □

Note also that proposition 1 can be seen as stating that o^n is the conjugate of the leftshift operation $l(n, x) = 2^n x$ through $s(x) = x + 1$ (eq. 2) and so is i^n through $s \circ s$ (eq. 3).

The following equations relate successor and predecessor to the iterated applications of o and i:

$$s(o^n(k)) = i(o^{s'(n)}(k)) \tag{13}$$

$$s(i^n(k)) = o^n(s(k)) \tag{14}$$

$$s'(o^n(k)) = i^n(s'(k)) \tag{15}$$

$$s'(i^n(k)) = o(i^{s'(n)}(k)) \tag{16}$$

3. HEREDITARILY BINARY NUMBERS

Let us observe that conventional number systems, as well as the bijective base-2 numeration system described so far, represent blocks of 0 and 1 digits somewhat naively - one digit for each element of the block. Alternatively, one might think that counting them and representing the resulting counters as *binary numbers* would be also possible. This brings us to the idea of hereditary number systems. At our best knowledge the first instance of such a system is used in [2], by iterating the polynomial base-n notation to the exponents used in the notation. We will next explore a hereditary number representation that implements the simple idea of representing the number of contiguous 0 or 1 digits in a number, as bijective base-2 numbers, recursively.

First, we define a data type for our tree represented natural numbers, that we call *hereditarily binary numbers* to emphasize that *binary* rather than *unary* encoding is recursively used in their representation.

DEFINITION 3. *The data type $\mathbb{T}$ of the set of hereditarily binary numbers is defined by the Haskell declaration:*

```
data T = E | V T [T]  | W T [T] deriving (Eq,Read,Show)
```

that automatically derives the equality relation "==", as well as reading and string representation operations. For shortness, we will call the members of type $\mathbb{T}$ *terms*. The intuition behind the disjoint union type $\mathbb{T}$ is the following:

- The term `E` (empty leaf) corresponds to zero
- the term `V x xs` counts the number `x+1` of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications
- the term `W x xs` counts the number `x+1` of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications
- the same principle is applied recursively for the counters, until the empty sequence is reached

One can see this process as run-length compressed bijective base-2 numbers, represented as trees with either empty leaves or at least one branch, after applying the encoding recursively.

DEFINITION 4. *The function $n : \mathbb{T} \to \mathbb{N}$ shown in equation* **17** *defines the unique natural number associated to a term of type $\mathbb{T}$.*

The computation of $n(W(VE[])[E,E,E]) = 42$ expands to $(((2^{0+1}-1+2)2^{0+1}-2+1)2^{0+1}-1+2)2^{2^{0+1}-1+1}-2$.

The Haskell equivalent of equation (17) is:

```
n E = 0
n (V x []) = 2^(n x +1)-1
n (V x (y:xs)) = (n u+1)*2^(n x + 1)-1 where u = W y xs
n (W x []) = 2^(n x+2)-2
n (W x (y:xs)) = (n u+2)*2^(n x + 1)-2 where u = V y xs
```

One can also view equation (17) as computing $n(E) = 0$, $n(V\ x_0\ [x_1, x_2, \ldots]) = (o^{1+n(x_0)} \circ i^{1+n(x_1)} \circ o^{1+n(x_2)} \circ \ldots)(E)$, $n(W\ x_0\ [x_1, x_2, \ldots]) = (i^{1+n(x_0)} \circ o^{1+n(x_1)} \circ i^{1+n(x_2)} \circ \ldots)(E)$ where "$\circ$" denotes function composition.

The following example illustrates the values associated with the first few natural numbers.

```
0 = n E
1 = n (V E [])
2 = n (W E [])
3 = n (V (V E []) [])
4 = n (W E [E])
5 = n (V E [E])
```

Note that a term of the form `V x xs` represents an odd number and a term of the form `W x xs` represents a strictly positive even number. The following holds:

PROPOSITION 5. *$n : \mathbb{T} \to \mathbb{N}$ is a bijection, i.e., each term canonically represents the corresponding natural number.*

PROOF. It follows from the equations (4), (5) by replacing the power of 2 functions with the corresponding iterated applications of o and i. □

4. SUCCESSOR (s) AND PREDECESSOR (s')

We will now specify successor and predecessor on data type $\mathbb{T}$ through two mutually recursive functions

```
s E = V E []
s (V E []) =  W E []
s (V E (x:xs)) =  W (s x) xs
s (V z xs) = W E (s' z : xs)
s (W z [])  = V (s z) []
s (W z [E]) = V z [E]
s (W z (E:y:ys)) = V z (s y:ys)
s (W z (x:xs)) = V z (E:s' x:xs)
```

```
s' (V E []) = E
s' (V z []) = W (s' z) []
s' (V z [E]) =  W z [E]
s' (V z (E:x:xs)) =  W z (s x:xs)
s' (V z (x:xs)) =  W z (E:s' x:xs)
s' (W E []) = V E []
s' (W E (x:xs)) = V (s x) xs
s' (W z xs) = V E (s' z:xs)
```

Please note that this definition is justified by the equations (13)–(16). Note also that termination is guaranteed as both `s` and `s'` use only induction on the structure of the terms.

The following holds:

PROPOSITION 6. *Denote $\mathbb{T}^+ = \mathbb{T} - \{\mathtt{E}\}$. The functions $s : \mathbb{T} \to \mathbb{T}^+$ and $s' : \mathbb{T}^+ \to \mathbb{T}$ are inverses.*

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

More generally, it can be proved by structural induction that Peano's axioms hold and, as a result, $< \mathbb{T}, E, s >$ is a Peano algebra.

PROPOSITION 7. *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 the average 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 $T(n) = O(log_2^*(n))$. □

PROPOSITION 8. *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 has the upper bound 2 as $\sum_{k=0}^{n} \frac{1}{2^k} = 2 - \frac{1}{2^n} < 2$. As on 2-bit numbers we have an average of 0.25 more calls, we can conclude that the total average number of calls is constant, with upper bound $2 + 0.25 = 2.25$. □

A quick empirical evaluation confirms this. When computing the successor on the first $2^{30} = 1073741824$ natural numbers, there are in total 2381889348 calls to `s` and `s'`, averaging to 2.2183 per computation. The same average for 100 successor computations on 5000 bit random numbers also oscillates around 2.22.

5. EMULATING BIJECTIVE BASE-2 OPERATIONS

To be of any practical interest, we will need to ensure that our data type $\mathbb{T}$ emulates also binary arithmetic. We will first show that it does, and next we will show that on a number of operations like exponent of 2 or multiplication by an exponent of 2, it significantly lowers complexity.

Intuitively the first step should be easy, as we need to express single applications or "un-applications" of `o` and `i` in terms of their iterates encapsulated in the `V` and `W` terms.

First we emulate single applications of `o` and `i` seen as virtual "constructors" on `data B = Zero | O B | I B`.

$$n(t) = \begin{cases} 0 & \text{if } t = \texttt{E}, \\ 2^{n(x)+1} - 1 & \text{if } t = \texttt{V x []}, \\ (n(u)+1)2^{n(x)+1} - 1 & \text{if } t = \texttt{V x (y:xs)} \text{ and } u = \texttt{W y xs}, \\ 2^{n(x)+2} - 2 & \text{if } t = \texttt{W x []}, \\ (n(u)+2)2^{n(x)+1} - 1 & \text{if } t = \texttt{W x (y:xs)} \text{ and } u = \texttt{V y xs}. \end{cases} \quad (17)$$

```
o E = V E []
o (V x xs) = V (s x) xs
o (W x xs) = V E (x:xs)

i E = W E []
i (V x xs) = W E (x:xs)
i (W x xs) = W (s x) xs
```

Next we emulate the corresponding "destructors" that can be seen as "un-applying" a single instance of `o` or `i`.

```
o' (V E []) = E
o' (V E (x:xs)) = W x xs
o' (V x xs) = V (s' x) xs

i' (W E []) = E
i' (W E (x:xs)) = V x xs
i' (W x xs) = W (s' x) xs
```

Finally the "recognizers" `e_` corresponding to `E`, `o_` (corresponding to *odd* numbers) and `i_` (corresponding to positive *even* numbers) simply detect `V` and `W`, indicating that `o` (and respectively `i`) was the last function applied.

```
e_ E = True
e_ _ = False

o_ (V _ _ ) = True
o_ _ = False

i_ (W _ _ ) = True
i_ _ = False
```

Note that each of the functions `o, o'` and `i, i'` calls `s` or `s'` exactly once, therefore:

PROPOSITION 9. *`o, o'` and `i, i'` are, on the average, constant time.*

PROPOSITION 10. *The worst case time complexity of the `o, o'` and `i, i'` operations on n is given by the* iterated logarithm $O(log^*(n))$.

DEFINITION 5. *The function $t : \mathbb{N} \to \mathbb{T}$ defines the unique tree of type $\mathbb{T}$ associated to a natural number as follows:*

$$t(x) = \begin{cases} \texttt{E} & \text{if } x = \texttt{0}, \\ \texttt{o}(t(\frac{x-1}{2})) & \text{if } x > 0 \text{ and } x \text{ is odd}, \\ \texttt{i}(t(\frac{x}{2} - 1)) & \text{if } x > 0 \text{ and } x \text{ is even}, \end{cases} \quad (18)$$

We can now define the corresponding Haskell function `t`:$\mathbb{T} \to \mathbb{N}$ that converts from trees to natural numbers.

Note that `pred x=x-1` and `div` is integer division.

```
t 0 = E
t x | x>0 && odd x = o(t (div (pred x) 2))
t x | x>0 && even x = i(t (pred (div x 2)))
```

The following holds:

PROPOSITION 11. *Let `id` denote $\lambda x.x$ and $\circ$ function composition. Then, on their respective domains*

$$t \circ n = id, \quad n \circ t = id \quad (19)$$

PROOF. By induction, using the arithmetic formulas defining the two functions. □

6. ARITHMETIC OPERATIONS

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

6.1 A few average constant time operations

Doubling a number `db` and reversing the `db` operation (`hf`) are quite simple, once one remembers that the arithmetic equivalent of function `o` is $o(x) = 2x + 1$.

```
db = s' . o
hf = o' . s
```

Note that efficient implementations follow directly from our number theoretic observations in section 2.

For instance, as a consequence of proposition 1, the operation `exp2`, computing an exponent of 2 , has the following simple definition in terms of `s` and `s'`.

```
exp2 E = V E []
exp2 x = s (V (s' x) [])
```

PROPOSITION 12. *The average time complexity of `db, hf` and `exp2` is constant and their worst case time complexity is $O(log_2^*(n))$.*

PROOF. It follows by observing that only 2 calls to `s, s', o, o'` are made. □

Next, we will derive additions and subtraction operations *favoring terms with large contiguous blocks of o^n and i 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, the algorithms are in uncharted territory and consequently more intricate than their traditional counterparts.

6.2 Addition and subtraction one block at a time

Next, we will derive more efficient addition and subtraction operations similar to `s` and `s'` that *work on one run-length encoded block at a time*, rather than by individual `o` and `i` steps.

We first define the functions `otimes` corresponding to $o^n(k)$ and `itimes` corresponding to $i^n(k)$.

```
otimes E y = y
otimes n E = V (s' n) []
otimes n (V y ys) = V (add n y) ys
otimes n (W y ys) = V (s' n) (y:ys)
```

```
itimes E y = y
itimes n E = W (s' n) []
itimes n (W y ys) = W (add n y) ys
itimes n (V y ys) = W (s' n) (y:ys)
```

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". When detecting that its argument counts a number of applications of `o`, `otimes` just increments that count. On the other hand, when the last function applied was `i`, `otimes` simply inserts a new count for `o` operations. A similar process corresponds to `itimes`. As a result, performance depends on the number of blocks rather than the bitsize of argument `n`.

We will state a number of arithmetic identities on $\mathbb{N}$ involving iterated applications of o and i.

PROPOSITION 13. *The following hold:*

$$o^k(x) + o^k(y) = i^k(x+y) \tag{20}$$

$$o^k(x) + i^k(y) = i^k(x) + o^k(y) = i^k(x+y+1) - 1 \tag{21}$$

$$i^k(x) + i^k(y) = i^k(x+y+2) - 2 \tag{22}$$

PROOF. By (4) and (5), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides. □

The corresponding Haskell code is:

```
oplus k x y =  itimes k (add x y)

oiplus k x y = s' (itimes k (s (add x y)))

iplus k x y = s' (s' (itimes k (s (s (add x y)))))
```

Note the use of `add` that we will define later as part of a chain of mutually recursive function calls, that together will provide an implementation of the intuitively simple idea: *they work on one run-length encoded block at a time. "Efficiency", in what follows, will be, therefore, conditional to numbers having comparatively few such blocks.*

The corresponding identities for subtraction are:

PROPOSITION 14.

$$x > y \;\Rightarrow\; o^k(x) - o^k(y) = o^k(x-y-1) + 1 \tag{23}$$

$$x > y+1 \;\Rightarrow\; o^k(x) - i^k(y) = o^k(x-y-2) + 2 \tag{24}$$

$$x \geq y \;\Rightarrow\; i^k(x) - o^k(y) = o^k(x-y) \tag{25}$$

$$x > y \;\Rightarrow\; i^k(x) - i^k(y) = o^k(x-y-1) + 1 \tag{26}$$

PROOF. By (4) and (5), we substitute the 2^k-based equivalents of o^k and i^k, and observe that the same reduced forms appear on both sides. Note that special cases are handled separately to ensure that subtraction is defined. □

The Haskell code, also covering the special cases, is:

```
ominus _ x y | x == y = E
ominus k x y = s (otimes k (s' (sub x y)))

iminus _ x y | x == y = E
iminus k x y =  s (otimes k (s' (sub x y)))

oiminus k x y | x==s y = s E
oiminus k x y | x == s (s y) = s (exp2 k)
oiminus k x y =  s (s (otimes k (s' (s' (sub x y)))))

iominus k x y = otimes k (sub x y)
```

Note the reference to `sub`, to be defined later, which is also part of the mutually recursive chain of operations.

The next two functions extract the iterated applications of o^n and respectively i^n from `V` and `W` terms:

```
osplit (V x []) = (x,E )
osplit (V x (y:xs)) = (x,W y xs)

isplit (W x []) = (x,E )
isplit (W x (y:xs)) = (x,V y xs)
```

We are now ready for defining addition. The base cases are the same as for `simpleAdd`:

```
add E y = y
add x E = x
```

In the case when both terms represent odd numbers, we apply the identity (20), after extracting the iterated applications of o as `a` and `b` with the function `osplit`.

```
add x y |o_ x && o_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = osplit y
  f EQ = oplus (s a) as bs
  f GT = oplus (s b) (otimes (sub a b) as) bs
  f LT = oplus (s a) as (otimes (sub b a) bs)
```

In the case when the first term is odd and the second even, we apply the identity (21), after extracting the iterated application of o and i as `a` and `b`.

```
add x y |o_ x && i_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = isplit y
  f EQ = oiplus (s a) as bs
  f GT = oiplus (s b) (otimes (sub a b) as) bs
  f LT = oiplus (s a) as (itimes (sub b a) bs)
```

In the case when the first term is even and the second odd, we apply the identity (21), after extracting the iterated applications of i and o as, respectively, `a` and `b`.

```
add x y |i_ x && o_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = osplit y
  f EQ = oiplus (s a) as bs
  f GT = oiplus (s b) (itimes (sub a b) as) bs
  f LT = oiplus (s a) as (otimes (sub b a) bs)
```

In the case when both terms represent even numbers, we apply the identity (22), after extracting the iterated application of i as `a` and `b`.

```
add x y |i_ x && i_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = isplit y
  f EQ = iplus (s a) as bs
  f GT = iplus (s b) (itimes (sub a b) as) bs
  f LT = iplus (s a) as (itimes (sub b a) bs)
```

Note the comparison operation `cmp`, to be defined later, also part of our chain of mutually recursive operations. Note also the local function `f` that in each case ensures that a block of the same size is extracted, depending on which of the two operands `a` or `b` is larger. The code for the subtraction function `sub` is similar:

```
sub x E = x
sub x y | o_ x && o_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = osplit y
  f EQ = ominus (s a) as bs
  f GT = ominus (s b) (otimes (sub a b) as) bs
  f LT = ominus (s a) as (otimes (sub b a) bs)
```

In the case when both terms represent odd numbers, we apply the identity (23), after extracting the iterated applications of o as `a` and `b`. For the other cases, we use, respectively, the identities 24, 25 and 26:

```
sub x y |o_ x && i_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = isplit y
  f EQ = oiminus (s a) as bs
  f GT = oiminus (s b) (otimes (sub a b) as) bs
  f LT = oiminus (s a) as (itimes (sub b a) bs)
```

```
sub x y |i_ x && o_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = osplit y
  f EQ = iominus (s a) as bs
  f GT = iominus (s b) (itimes (sub a b) as) bs
  f _ = iominus (s a) as (otimes (sub b a) bs)
```

```
sub x y |i_ x && i_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = isplit y
  f EQ = iminus (s a) as bs
  f GT = iminus (s b) (itimes (sub a b) as) bs
  f LT = iminus (s a) as (itimes (sub b a) bs)
```

6.3 Efficient arithmetic comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our type $\mathbb{T}$. It relies on `bitsize` computing the number of applications of o and i constructing a term in $\mathbb{T}$. It 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 15. *Let* `bitsize` *count the number of applications of o and i operations on a bijective base-2 number. Then* $\texttt{bitsize}(x) < \texttt{bitsize}(y) \Rightarrow x < y$.

Proof. Observe that their lexicographic enumeration ensures that the bitsize of bijective base-2 numbers is a non-decreasing function. □

```
cmp E E = EQ
cmp E _ = LT
cmp _ E = GT
cmp x y | x' /= y'  = cmp x' y' where
  x' = bitsize x
  y' = bitsize y
cmp x y =
  compBigFirst (reversedDual x) (reversedDual y)
```

The function `compBigFirst` compares two terms known to have the same `bitsize`. It works on reversed (big digit first) variants, computed by `reversedDual` and it takes advantage of the block structure using the following proposition:

Proposition 16. *Assuming two terms of the same bitsizes, the one starting with i is larger than one starting with o.*

Proof. Observe that "big digit first" numbers are lexicographically ordered with $o < i$. □

As a consequence, `cmp` only recurses when *identical* blocks head the sequence of blocks, otherwise it infers the `LT` or `GT` relation.

```
compBigFirst E E = EQ
compBigFirst x y | o_ x && o_ y = f (cmp a b) where
    (a,c) = osplit x
    (b,d) = osplit y
    f EQ = compBigFirst c d
    f LT = GT
    f GT = LT
compBigFirst x y | i_ x && i_ y = f (cmp a b) where
    (a,c) = isplit x
    (b,d) = isplit y
    f EQ = compBigFirst c d
    f other = other
compBigFirst x y | o_ x && i_ y = LT
compBigFirst x y | i_ x && o_ y = GT
```

The function `reversedDual` reverses the order of application of the o and i operations to a "biggest digit first" order. For this, it only needs to reverse the order of the alternative blocks of o^k and i^l. It uses the function `len` to compute the number of these blocks and infer that if odd, the last block is the same as the first and otherwise it is its alternate.

```
reversedDual E = E
reversedDual (V x xs) = f (len (y:ys)) where
  (y:ys) = reverse (x:xs)
  f l | o_ l = V y ys
  f l | i_ l = W y ys
reversedDual (W x xs) = f (len (y:ys)) where
  (y:ys) = reverse (x:xs)
  f l | o_ l = W y ys
  f l | i_ l = V y ys
```

```
len [] = E
len (_:xs) = s (len xs)
```

6.4 Computing `dual` and `bitsize`

The function `dual` flips `o` and `i` operations for a natural number seen as written in bijective base 2. Note that with our tree representation it is constant time, as it simply flips once the constructors `V` and `W`.

```
dual E = E
dual (V x xs) = W x xs
dual (W x xs) = V x xs
```

The function `bitsize` computes the number of applications of the `o` and `i` operations. It works by summing up (using Haskell's `foldr`) the counts of `o` and `i` operations composing a tree-represented natural number.

```
bitsize E = E
bitsize (V x xs) = s (foldr add1 x xs)
bitsize (W x xs) = s (foldr add1 x xs)

add1 x y = s (add x y)
```

Note that `bitsize` also provides an efficient implementation of the integer log_2 operation `ilog2`.

```
ilog2 x = bitsize (s' x)
```

6.5 Fast multiplication by an exponent of 2

The function `leftshiftBy` operation uses the fact that repeated application of the `o` operation (`otimes`) , provides an efficient implementation of multiplication with an exponent of 2.

```
leftshiftBy _ E = E
leftshiftBy n k = s (otimes n (s' k))
```

The following holds:

PROPOSITION 17. `leftshiftBy` *is (roughly) logarithmic in the bitsize of its arguments.*

PROOF. it follows by observing that at most one addition on data logarithmic in the bitsize of the operands is performed. □

6.6 Fast division by an exponent of 2

Division by an exponent of 2 (equivalent to the *rightshift operation* is more intricate. It takes advantage of identities (4) and (5) in a way that is similar to `add` and `sub`. First, the function `toShift` transforms the outermost block of o^m or i^m applications to a multiplication of the form $k2^m$. It also remembers if it had o^m or i^m, as the first component of the triplet it returns. Note that, as a result, the operation is actually reversible.

```
toShift x | o_ x = (o E,s a,s b) where
  (a,b) = osplit x
toShift x | i_ x = (i E,s a,s (s b)) where
  (a,b) = isplit x
```

Next the function `rightshiftBy` goes over its argument `k` one block at a time, by comparing the size of the block and its argument `m` that is decremented after each block by the size of the block. The local function `f` handles the details, according to the nature of the block (o^m or i^m), and stops when the argument is exhausted. More precisely, based on the result `EQ, LT, GT` of the comparison, as well on the type of block (as recognized by `o_ p` and `i_ p`), it applies back `otimes` or `itimes` when the block is larger than the value of `m`. Otherwise, it calls itself with the value of `m` reduced by the size to the block as its first argument.

```
rightshiftBy _ E = E
rightshiftBy m k = f (cmp m a) where
  (p,a,b) = toShift k

  f EQ | o_ p = sub b p
  f EQ | i_ p = s (sub b p)
  f LT | o_ p = otimes (sub a m) (sub b p)
  f LT | i_ p = s (itimes (sub a m) (sub b p))
  f GT =  rightshiftBy (sub m a) b
```

The following example illustrates the fact that `rightShiftBy` inverts the result of `leftshiftBy` on a very large number.

```
*HBin> s' (exp2 (t 100000))
V (V (W E [E]) [E,E,E,E,V E [],V (V E []) [],E]) []
*HBin> leftshiftBy (t 1000) it
W E [W (W E []) [V E [],V (V E []) []],W (W E [E])
    [V E [],E,E,V E [],V (V E []) [],E]]
*HBin> rightshiftBy (t 1000) it
V (V (W E [E]) [E,E,E,E,V E [],V (V E []) [],E]) []
```

7. STRUCTURAL COMPLEXITY

As a measure of structural complexity we define the function `tsize` that counts the nodes of a tree of type $\mathbb{T}$ (except the root).

```
tsize E = E
tsize (V x xs) = foldr add1 E (map tsize (x:xs))
tsize (W x xs) = foldr add1 E (map tsize (x:xs))
```

It corresponds to the function $c : \mathbb{T} \to \mathbb{N}$ defined as follows:

$$c(t) = \begin{cases} 0 & \text{if } \texttt{t} = \texttt{0}, \\ \sum_{y \in (\texttt{x:xs})} (1 + ts(y)) & \text{if } \texttt{t} = \texttt{V x xs}, \\ \sum_{y \in (\texttt{x:xs})} (1 + ts(y)) & \text{if } \texttt{t} = \texttt{W x xs}. \end{cases} \quad (27)$$

The following holds:

PROPOSITION 18. *For all terms* $t \in \mathbb{T}$, `tsize t` $\leq$ `bitsize t`.

PROOF. By induction on the structure of t, by observing that the two functions have similar definitions and corresponding calls to `tsize` return terms assumed smaller than those of `bitsize`. □

The following example illustrates their use:

```
*HBin> map (n.tsize.t) [0,100,1000,10000]
[0,6,8,10]
*HBin> map (n.bitsize.t) [0,100,1000,10000]
[0,6,9,13]

*HBin> map (n.tsize.t) [2^16,2^32,2^64,2^256]
[4,5,5,5]
*HBin> map (n.bitsize.t) [2^16,2^32,2^64,2^256]
[16,32,64,256]
```

Figure 1 shows the reductions in structural complexity compared with bitsize for an initial interval of $\mathbb{N}$.

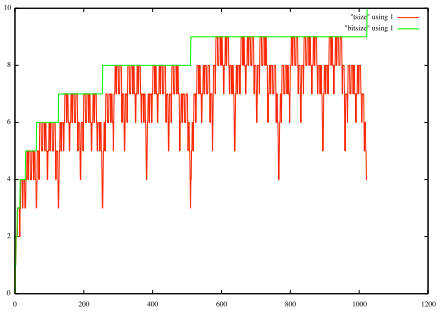

Figure 1: Structural complexity (lower line) bounded by bitsize (upper line) from 0 to $2^{10} - 1$

After defining the function `iterated` that applies f k times

```
iterated f E x = x
iterated f k x = f (iterated f (s' k) x)
```

we can exhibit, for a given bitsize, a best case

```
bestCase k = iterated wTree k E where wTree x = W x []
```

and two worst cases

```
worstCase k = iterated (i.o) k E

worstCase' k = iterated (o.i) k E
```

The following examples illustrate these functions:

```
*HBin> bestCase (t 3)
W (W (W E []) []) []
*HBin> n it
65534
*HBin> n (bitsize (bestCase (t 3)))
```

```
15
*HBin> n (tsize (bestCase (t 3)))
3

*HBin> worstCase (t 3)
W E [E,E,E,E,E]
*HBin> n it
84
*HBin> n (bitsize (worstCase (t 3)))
6
*HBin> n (tsize (worstCase (t 3)))
6
```

It follows from identity (7) that the predicate `bestCase` computes the iterated exponent of 2 (tetration) and then applies the predecessor to it twice, i.e., it computes $2^{2^{\cdots^2}} - 2$. A simple closed formula can also be found for `worstCase`:

PROPOSITION 19. *The function* `worstCase k` *computes the value in* $\mathbb{T}$ *corresponding to the value* $\frac{4(4^k-1)}{3} \in \mathbb{N}$.

PROOF. By induction or by applying the iterated function formula to $f(x) = i(o(x))) = 2(2x+1)+2 = 4(x+1)$. □

8. RELATED WORK

Several notations for very large numbers accommodating "towers of exponents" have been invented in the past. Examples include Knuth's *arrow-up* notation [3] covering operations like the *tetration* (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 hereditary number system, to our best knowledge, occurs in the proof of Goodstein's theorem [2].

Another hereditary number system is Knuth's TCALC program [4] 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 the representation is not canonical. 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] a similar exponential-based notation called "integer decision diagrams" is introduced, providing a compressed representation for sparse integers, sets and various other data types.

Like our trees of type $\mathbb{T}$, Conway's surreal numbers [1] can also be seen as inductively constructed trees. While our focus is on efficient large natural number arithmetic and sparse set representations, surreal numbers model games, transfinite ordinals and generalizations of real numbers.

Another hereditary number system is described in [7]. Like [4], it uses a binary tree-based representation derived from the bijection $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+, f(x,y) = 2^x(2y+1)$. While the representation is canonical, it provides constant time exponent of 2 and leftshift operation and, like the proposal in this paper, it implements all the basic arithmetic operations, it does not handle well arbitrary linear combinations of towers of exponent numbers, as for instance, numbers of the form $2^x - 1$ are expanded to large unbalanced binary trees. Like [4], [7] defines arithmetic computations based on a simple recursive data type. On the other hand, we are using a more complex 2-colored (with tags `V` and `W`) multiway tree type, while, like in [7] ensuring that the representation is bijective and canonical. Several other algorithms and examples of representations for very large record holder primes using hereditarily binary numbers are described in an extended version of this paper in the *arxiv* repository [6].

9. CONCLUSION AND FUTURE WORK

We have provided in the form of a literate Haskell program a declarative specification of a tree-based number system. Our emphasis here was on the correctness and the theoretical complexity bounds of our operations rather than the packaging in a form that would compete with a C-based arbitrary size integer package like GMP. We have also ensured that our algorithms are as simple as possible and we have closely correlated our Haskell code with the formulas describing the corresponding arithmetical properties. Our algorithms rely on properties of blocks of iterated applications of functions rather than the "digits as coefficients of polynomials" view of traditional numbering systems. While the rules are often more complex, restricting our code to a purely declarative subset of functional programming made managing a fairly intricate network of mutually recursive dependencies much easier.

As future work we plan building a practical package (that uses our representation only for numbers larger than the size of the machine word) and specialize our algorithms for this hybrid representation. In particular, parallelization of our algorithms, that seems natural given our tree representation, would follow once the sequential performance of the package is in a practical range. Other developments with practicality in mind would involve extensions to signed integers and rational numbers.

10. REFERENCES

[1] J. H. Conway. *On Numbers and Games*. AK Peters, Ltd., 2nd edition, 2000.

[2] R. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, (9):33–41, 1944.

[3] D. E. Knuth. Mathematics and Computer Science: Coping with Finiteness. *Science*, 194(4271):1235 –1242, 1976.

[4] D. E. Knuth. TCALC program, Dec. 1994.

[5] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[6] P. Tarau. Arithmetic Algorithms for Hereditarily Binary Natural Numbers, June 2013. http://arxiv.org/abs/1306.1128.

[7] P. Tarau and D. Haraburda. On Computing with Types. In *Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track*, pages 1889–1896, Riva del Garda (Trento), Italy, Mar. 2012.

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

[9] Wikipedia. Iterated function — wikipedia, the free encyclopedia, 2013. [Online; accessed 18-April-2013].