New Arithmetic Algorithms for Hereditarily Binary Natural Numbers

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

***Abstract*—Hereditarily binary numbers are a tree-based number representation derived from a bijection between natural numbers and iterated applications of two simple functions corresponding to bijective base 2 numbers. This paper describes several new arithmetic algorithms on hereditarily binary numbers that, while within constant factors from their traditional counterparts for their average case behavior, make tractable important computations that are impossible with traditional number representations.**

***Keywords*-hereditary numbering systems, compressed number representations, arithmetic computations with giant numbers, compact representation of large prime numbers**

I. Introduction

This paper is a sequel to [1][1] where we have introduced a tree based number representation, called *hereditarily binary numbers*.

In [1] we present specialized algorithms for basic arithmetic operations (addition, subtraction, comparison, double, half, exponent of 2 and bitsize), that favor *numbers with relatively few blocks of contiguous* `0` *and* `1` *bijective base-2 digits*, for which dramatic complexity reductions result even when operating on very large, "towers-of-exponents" numbers.

This paper covers several new arithmetic operations on hereditarily binary numbers and some specialized arithmetic algorithms that are used in number theory and cryptography, like modular operations and Miller-Rabin and Lucas-Lehmer primality tests.

In addition, we perform a performance evaluation confirming our theoretical complexity estimates of [1] and illustrate compact representations of some record-holder large prime numbers, while emphasizing that hereditarily binary numbers replace bitsize with a measure of *structural complexity* as the key parameter deciding tractability of arithmetic operations.

The paper is organized as follows. Section II overviews basic definitions for *hereditarily binary numbers* and summarizes some of their properties, following [1]. Section III describes new arithmetic algorithms for hereditarily binary numbers. Section IV compares the performance of several algorithms operating on hereditarily binary numbers with their conventional counterparts. Section V illustrates compact tree-representations of some record-holder number-theoretical entities like Mersenne, Fermat, Proth and Woodall primes. Section VI discusses related work. Section VII concludes the paper and discusses future work. The Appendix wraps our arithmetic operations as instances of Haskell's number and order classes and provides the function definitions from [1] referenced in this paper.

We have adopted a *literate programming* style, i.e., the code contained in the paper forms a Haskell module, available at http://www.cse.unt.edu/~tarau/research/2014/hbinx.hs. It imports code described in detail in the paper [1], from file http://www.cse.unt.edu/~tarau/research/2014/hbin.hs. A Scala package implementing the same tree-based computations is available from http://code.google.com/p/giant-numbers/. We hope that this will encourage the reader to experiment interactively and validate the technical correctness of our claims.

[1]An extended draft version of [1] is available at the *arxiv* repository [2].

II. Hereditarily Binary Numbers

We will summarize, following [1], the basic concepts behind *hereditarily binary numbers*. Through the paper, we denote $\mathbb{N}$ the set of natural numbers and $\mathbb{N}^+$ the set of strictly positive natural numbers.

A. Bijective base-2 numbers

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 (defined for an arbitrary base in [3] pp. 90-92 as "m-adic" numbers). 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 $n \in \mathbb{N}$ the unique sequence of applications of functions o and i to 0 that evaluates to n.

With this representation, one obtains $, 1 = o(0), 2 = i(0), 3 = o(o(0)), 4 = i(o(0)), 5 = o(i(0))$ etc. Clearly:

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

B. Efficient arithmetic with iterated functions o^n and i^n

Several arithmetic identities are proven in [1] and used to express efficient "one block of o^n or i^n operations at a time" algorithms for various arithmetic operations. Among them, we recall the following two, showing the connection of our iterated function applications with "left shift/multiplication by a power of 2" operations.

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

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

In particular

$$o^n(0) = 2^n - 1 \tag{4}$$

$$i^n(0) = 2^{n+1} - 2 \tag{5}$$

C. Hereditarily binary numbers as a data type

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 2: The data type `T` of the set of hereditarily binary numbers is defined in [1] by the Haskell declaration:

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

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

The intuition behind the type `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 `xs`
- 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 `xs`

In [1] the bijection between $\mathbb{N}$ and `T` is provided by the function $n : \mathtt{T} \to \mathbb{N}$ and its inverse $t : \mathbb{N} \to \mathtt{T}$.

Definition 3: The function $n : \mathtt{T} \to \mathbb{N}$ shown in equation (**6**) defines the unique natural number associated to a term of type `T`.

This bijection ensures that hereditarily binary numbers provide a canonical representation of natural numbers and the equality relation on type `T` can be derived by structural induction.

The following examples show the workings of the bijection n and illustrate that "structural complexity", defined in [1] as the *size of the tree representation without the root*, is bounded by the bitsize of a number and favors numbers in the neighborhood of towers of exponents of 2.

$2^{2^{16}} - 1 \to$ `V (V (V (V (V (V E[]) []) []) []) []) []`

$\to 2^{2^{2^{2^{2^{0+1}-1+1}-1+1}-1+1}-1+1} - 1$

$20 \to$ `W E [E,E,E]`

$\to (((2^{0+1} - 1 + 2)2^{0+1} - 2 + 1)2^{0+1} - 1 + 2)2^{0+1} - 2$

In [1] basic arithmetic operations are introduced with complexity parameterized by the size of the tree representation of their operands rather than their bitsize.

After defining constant average time *successor* and *predecessor* functions `s` and `s'`, constant average time definitions of `o` and `i` are given in [1], as well as for the corresponding inverse operations `o'` and `i'`, that can be seen as "unapplying" a single instance of `o` or `i`, and "recognizers" `e_` (corresponding to `E`), `o_` (corresponding to *odd* numbers) and `i_` (corresponding to *even* numbers).

III. Arithmetic operations working one o^k or i^k block at a time

In [1] we provide algorithms for the basic arithmetic operations of addition (`add`) subtraction (`sub`), exponent of 2 (`exp2`), multiplication by a power of `2`, division by a power of `2` (`leftshiftBy, rightshiftBy`) arithmetic comparison (`cmp`) and computation of the size of the equivalent binary representation of a number, `bitsize`. We will describe in this section algorithms for several other arithmetic operations, optimized for working *one block of o^k or i^k applications at a time*.

General multiplication

Like in the case of `add` and `sub` operations in [1] we can derive a multiplication algorithm based on several arithmetic identities involving exponents of 2 and iterated applications of the functions `o` and `i`.

Proposition 1: The following holds:

$$o^n(a)o^m(b) = o^{n+m}(ab + a + b) - o^n(a) - o^m(b) \tag{7}$$

Proof: By (2), we can expand and then reduce:
$o^n(a)o^m(b) = (2^n(a+1) - 1)(2^m(b+1) - 1) =$
$2^{n+m}(a+1)(b+1) - (2^n(a+1) + 2^m(b+1)) + 1 =$
$2^{n+m}(a+1)(b+1) - 1 - (2^n(a+1) - 1 + 2^m(b+1) - 1 + 2) + 2 =$
$o^{n+m}(ab + a + b + 1) - (o^n(a) + o^m(b)) - 2 + 2 =$
$o^{n+m}(ab + a + b) - o^n(a) - o^m(b)$ ∎

Proposition 2:

$$i^n(a)i^m(b) = i^{n+m}(ab + 2(a+b+1)) + 2 - i^{n+1}(a) - i^{m+1}(b) \tag{8}$$

Proof: By (3), we can expand and then reduce:
$i^n(a)i^m(b) = (2^n(a+2) - 2)(2^m(b+2) - 2) =$
$2^{n+m}(a+2)(b+2) - (2^{n+1}(a+2) - 2 + 2^{m+1}(b+2) - 2) =$
$2^{n+m}(a+2)(b+2) - i^{n+1}(a) - i^{m+1}(b) =$
$2^{n+m}(a+2)(b+2) - 2 - (i^{n+1}(a) + i^{m+1}(b)) + 2 =$
$2^{n+m}(ab + 2a + 2b + 2 + 2) - 2 - (i^{n+1}(a) + i^{m+1}(b)) + 2 =$
$i^{n+m}(ab + 2a + 2b + 2) - (i^{n+1}(a) + i^{m+1}(b)) + 2 =$
$i^{n+m}(ab + 2(a + b + 1)) + 2 - i^{n+1}(a) - i^{m+1}(b)$ ∎

The corresponding Haskell code starts with the obvious base cases:

```
mul _ E = E
mul E _ = E
```

When both terms represent odd numbers we apply the identity (7):

```
mul x  y | o_ x && o_ y = r2 where
  (n,a) = osplit x
  (m,b) = osplit y
  p1 = add (mul a b) (add a b)
  p2 = otimes (add (s n) (s m)) p1
  r1 = sub p2 x
  r2 = sub r1 y
```

Note that the function `osplit` defined in [1] is used to separate the first block of `o` applications. The next two cases are reduced to the previous one by using the identity $i = s \circ o$.

```
mul x y | o_ x && i_ y = add x (mul x (s' y))
mul x y | i_ x && o_ y = add y (mul (s' x) y)
```

$$n(t) = \begin{cases} 0 & \text{if } t = \texttt{E}, \\ 2^{n(\texttt{x})+1} - 1 & \text{if } t = \texttt{V x []}, \\ (n(u)+1)2^{n(\texttt{x})+1} - 1 & \text{if } t = \texttt{V x (y:xs)} \text{ and } u = \texttt{W y xs}, \\ 2^{n(\texttt{x})+2} - 2 & \text{if } t = \texttt{W x []}, \\ (n(u)+2)2^{n(\texttt{x})+1} - 2 & \text{if } t = \texttt{W x (y:xs)} \text{ and } u = \texttt{V y xs}. \end{cases} \tag{6}$$

Finally, the most difficult case implements identity (8)

```
mul x y | i_ x && i_ y = r2 where
  (n,a) = isplit x
  (m,b) = isplit y
  p0 = mul a b
  s0 = s (add a b)
  p1 = add p0 (db s0)
  e1 = itimes (add (s n) (s m)) p1
  e2 = i x
  e3 = i y
  r1 = sub (s (s e1)) e2
  r2 = sub r1 e3
```

Note that when the operands are composed of large blocks of alternating o^n and i^m applications, the algorithm is quite efficient as it works (roughly) in time depending on the the number of blocks 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:

```
*HBinX> let term1 =
  sub (exp2 (exp2 (t 12345))) (exp2 (t 6789))
*HBinX> let term2 =
  add (exp2 (exp2 (t 123))) (exp2 (t 456789))
*HBinX> ilog2 (ilog2 (mul term1 term2))
V E [E,E,W E [],V E [E],E]
*HBinX> n it
12345
```

This example illustrates that our arithmetic operations are not limited by the size of their operands, but only by their "structural complexity" defined in [1] as the size of the tree representation of number.

Note that, by contrast to algorithms like Karatsuba or Toom-Cook [4] 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 bijective base 2 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 $O(n^2)$.

A. Power

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

$$(o^n(a))^2 = o^{2n}(a^2 + 2a) - 2o^n(a) \tag{9}$$

$$(i^n(a))^2 = i^{2n}(a^2 + 2(2a+1)) + 2 - 2i^{n+1}(a) \tag{10}$$

```
square E = E
square x | o_ x = r where
  (n,a) = osplit x
  p1 = add (square a) (db a)
  p2 = otimes (i n) p1
  r = sub p2 (db x)
square x | i_ x = r where
  (n,a) = isplit x
  p1 = add (square a) (db (o a))
  p2 = itimes (i n) p1
  r = sub (s (s p2)) (db (i x))
```

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

```
pow _ E = V E []
pow x y | o_ y = mul x (pow (square x) (o' y))
pow x y | i_ y = mul x2 (pow x2 (i' y)) where x2 =
square 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:

```
*HBinX> n (bitsize (pow (t 2014) (t 100)))
1097
*HBinX> pow (t 32) (t 10000000)
W E [W (W (V E []) [])
  [W E [E],V (V E []) [],E,E,E,W E [E],E]]
```

B. Division operations

We give here a fairly efficient integer division algorithm. However, it does not provide the same complexity gains as, for instance, multiplication, addition or subtraction. Finding a "one block at a time" fast division algorithm using something like Newton's method is subject of future work.

```
div_and_rem x y | LT == cmp x y = (E,x)
div_and_rem x y | y /= E = (q,r) where
  (qt,rm) = divstep x y
  (z,r) = div_and_rem rm y
  q = add (exp2 qt) z
```

The function `divstep` implements a step of the division operation.

```
  divstep n m = (q, sub n p) where
    q = try_to_double n m E
    p = leftshiftBy q m
```

The function `try_to_double` doubles its second argument while smaller than its first argument and returns the number of steps it took. This value will be used by `divstep` when applying the `leftshiftBy` operation.

```
  try_to_double x y k =
    if (LT==cmp x y) then s' k
    else try_to_double x (db y) (s k)

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

C. Integer square root

An efficient integer square root, using Newton's method, is implemented as follows:

```
isqrt E = E
isqrt n = if cmp (square k) n==GT then s' k else k where
  two = i E
  k=iter n
  iter x = if cmp (absdif r x)  two == LT
    then r
    else iter r where r = step x
  step x = divide (add x (divide n x)) two

absdif x y = if LT == cmp x y then sub y x else sub x y
```

D. Modular power

The modular power operation $x^y (mod\ m)$ can be optimized to avoid the creation of large intermediate results, by combining "power by squaring" and pushing the modulo operation inside the inner function `modStep`.

```
modPow m base expo = modStep expo (V E []) base where
  modStep (V E []) r b  = (mul r b) `remainder` m
  modStep x r b | o_ x =
    modStep (o' x) (remainder (mul r b) m)
                   (remainder (square b)  m)
  modStep x r b = modStep (hf x) r
    (remainder (square b) m)
```

E. Lucas-Lehmer primality test for Mersenne numbers

The Lucas-Lehmer primality test has been used for the discovery of all the record holder largest known prime numbers of the form $2^p - 1$ with p prime in the last few years (see section V). It is based on iterating $p - 2$ times the function $f(x) = x^2 - 2$, starting from $x = 4$. Then $2^p - 1$ is prime if and only if the result modulo $2^p - 1$ is 0, as proven in [5]. The function `ll_iter` implements this iteration.

```
ll_iter k n m | e_ k = n
ll_iter k n m = fastmod y m where
   x = ll_iter (s' k) n m
   y = s' (s' (square x))
```

It relies on the function `fastmod` which provides a specialized fast computation of $k\ modulo\ (2^p - 1)$.

```
fastmod k m | k == s' m = E
fastmod k m | LT == cmp k m = k
fastmod k m = fastmod (add q r) m where
  (q,r) = div_and_rem k m
```

Finally the Lucas-Lehmer test is implemented as follows:

```
lucas_lehmer (W E []) = True
lucas_lehmer p = e_ (ll_iter p_2 four m) where
  p_2 = s' (s' p)
  four = i (o E)
  m  = exp2 p
```

We illustrate its use for detecting a few Mersenne primes:

```
*HBinX> map n (filter lucas_lehmer
            (map t [3,5..31]))
[3,5,7,13,17,19,31]
*HBinX> map (\p->2^p-1) it
[7,31,127,8191,131071,524287,2147483647]
```

Note that the last line contains the Mersenne primes corresponding to p. We refer to the section V for the compact hereditarily binary representation of the largest known Mersenne prime.

F. Miller-Rabin probabilistic primality test

Let $\nu_2(x)$ denote the *dyadic valuation of x*, i.e., the largest exponent of 2 that divides x. The function `dyadicSplit` defined by equation 11

$$dyadicSplit(k) = (k, \frac{k}{2^{\nu_2(k)}}) \tag{11}$$

can be implemented as an average constant time operation as:

```
dyadicSplit z | o_ z = (E,z)
dyadicSplit z | i_ z = (s x, s (g xs)) where
  V x xs = s' z

  g [] =  E
  g (y:ys) = W y ys
```

After defining a sequence of `k` random natural numbers in an interval

```
randomNats seed k smallest largest = map t ns  where
  ns :: [Integer]
  ns = take k (randomRs
    (n smallest,n largest) (mkStdGen seed))
```

we are ready to implement the function `oddPrime` that runs `k` tests and concludes primality with probability $1 - \frac{1}{4^k}$ if all `k` calls to function `strongLiar` succeed.

```
oddPrime k m = all strongLiar as where
  m' = s' m
  as = randomNats k k (W E []) m'
  (l,d) = dyadicSplit m'

  strongLiar a = (x == V E []) ||
      (any (== m') (squaringSteps l x)) where
    x = modPow m a d

    squaringSteps E _ = []
    squaringSteps l x = x:squaringSteps (s' l)
      (remainder (square x) m)
```

Note that we use `dyadicSplit` to find a pair `(l,d)` such that `l` is the largest power of `t 2` dividing the predecessor `m'` of the suspected prime `m`. The function `strongLiar` checks, for a random base `a`, a condition that primes (but possibly also a few composite numbers) verify.

Finally `isProbablyPrime` handles the case of even numbers and calls `oddPrime` with the parameter specifying the number of tests, `k=42`.

```
isProbablyPrime (W E [])  = True
isProbablyPrime (W _ _) = False
isProbablyPrime p = oddPrime 42 p
```

The following example illustrates the correct behavior of the algorithm on a the interval `[2..100]`.

```
*HBinX> map n (filter isProbablyPrime
            (map t [2..100]))
[2,3,5,7,11,13,17,19,23,29,31,37,41,
 43,47,53,59,61,67,71,73,79,83,89,97]
```

IV. Performance evaluation

In [1] functions returning worst and best case tree representations are defined. The worst case representation of hereditarily binary numbers is proportional to their bitsize:

```
> worstCase 5
W E [E,E,E,E,E,E,E,E,E]
> n it
1364
```

On the other hand, the "tower of exponents" best case provides a very compact representation to gigantic numbers:

```
> bestCase 5
W (W (W (W (W E []) []) []) []) []
> n (bitsize (bitsize it))
65536
```

Therefore, it is clearly possible, provided that the algorithms in [1] and in this paper provide super-exponential speed-ups for numbers that are in the neighborhood of towers of exponents of `2`, to have computations with such giant numbers that are intractable even with the very best arbitrary integer packages like the GMP library used by Haskell.

At the same time, GMP is written in highly optimized compiled C (+ some assembler), while our hereditarily binary numbers are supported by a tree data structure in interpreted Haskell (ghci). Consequently, one should expect large, up to 2-3 orders of magnitude constant factors on computations where our worst case dominates. This will be especially the case when GMP's multiplication (Karatsuba's algorithm, and fast Fourier transform based) competes with our simpler algorithm, focussed exclusively on taking advantage of low structural complexity numbers.

On the other hand, our constant time successor and predecessor algorithm will be, surprisingly, very close to its native counterpart, while addition/subtraction-intensive operations will be somewhere in-between.

Our performance measurements (run on a Mac Air with 8GB of memory and an Intel i7 processor) serve two objectives:

1) to show that, on the average, our tree based representations perform on a blend of arithmetic computations within a constant factor compared to conventional bitstring-based computations
2) to show that on interesting special computations they match or outperform the bitstring-based arbitrary size `Integer`, due to the much lower asymptotic complexity of such operations on data type `T`, despite of the 2-3 orders of magnitude gap with GMP.

Finally, let us emphasize that this comparison is by no means a benchmarking of two competing arbitrary integer arithmetic libraries, but rather a straightforward empirical check that our arithmetic algorithms have the complexity predicted by their theoretical study in [1] and that they work within their expected complexity bounds.

For instance, objective *1* is well served by the Ackermann function that exercises the successor and predecessor functions quite heavily.

On the other hand, objective *2* is served by benchmarks that take advantage of the compressed representation of very large, record holder primes. In some cases, the conventional representations are unable to run these benchmarks within existing computer memory and CPU-power limitations, as marked with "?" in the comparison table of figure 1. In other cases, like in the "`v (t 22) (t 11)`" benchmark, computing a very large boolean projection variable made of relatively few alternating blocks of `0`s and `1`s, data type `T` performs significantly faster than binary representations. The last two rows in figure 1 show two more extreme examples. The first computes `1000` steps of the *Syracuse function*, starting from a giant towers of exponents number and the second multiplies together 5 very large, record holder prime numbers.

Benchmark	Integer	tree type `T`
$2^{2^{30}}$	10192	0
v 22 11	4850	297
Ackermann 3 7	491	718
2^{21} predecessors	1979	2330
fibonacci 30	3249	19414
sum of first 2^{16} naturals	68	10016
powers	46	13485
generating primes	6	4807
factorial of 200	2	8040
1000 syracuse steps from $2^{\cdots^{2^2}}$	?	9070
product of 5 giant primes	?	904

Fig. 1: Time (in ms.) on a few small benchmarks

Together they indicate that optimized libraries derived from our tree-based representations are likely to be competitive with existing bitstring-based packages on typical computations and outperform them by an arbitrary margin on some number-theoretically interesting computations, on giant numbers. While the code of the benchmarks is omitted due to space constraints, it is part of the companion Haskell file at http://www.cse.unt.edu/~tarau/Research/2014/hbinx.hs.

V. Compact representation of some record-holder giant numbers

Let's first observe that Fermat, Mersenne, perfect numbers have all compact expressions with our tree representation of type `T`.

```
fermat n = s (exp2 (exp2 n))

mersenne p = s' (exp2 p)

perfect p = s (V q [q]) where q = s' (s' p)
```

```
HBin> mersenne 127
170141183460469231731687303715884105727
> mersenne (t 127)
V (W (V E [E]) []) []
```

The largest known prime number, found by the GIMPS distributed computing project [6] in January 2013 is the 48-th Mersenne prime = $2^{57885161} - 1$ (with possibly smaller Mersenne primes below it). It is defined in Haskell as follows:

```
-- exponent of the 48-th known Mersenne prime
prime48 = 57885161
-- the actual Mersenne prime
mersenne48 = s' (exp2 (t prime48))
```

While it has a bit-size of 57885161, its compressed tree representation is rather small:

```
> mersenne48
V (W E [V E [],E,E,V (V E []) [],
   W E [E],E,E,V E [],V E [],W E [],E,E]) []
```

The equivalent DAG representation of the 48-th Mersenne prime, shown in Figure 2, has only 7 shared nodes and structural complexity 22.

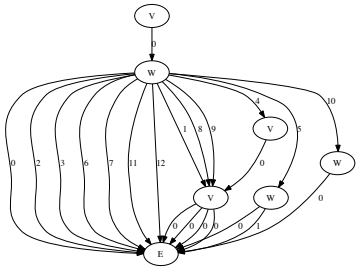

Fig. 2: Largest known prime number discovered in January 2013: the 48-th Mersenne prime, represented as a DAG

Due to their relation to Mersenne primes, similar compact representations can also be derived for perfect numbers. For instance, the largest known perfect number, derived from the largest known Mersenne prime as $2^{57885161-1}(2^{57885161}-1)$, (involving only 8 shared nodes and structural complexity 43) is:

```
perfect48 = perfect (t prime48)
```

Similarly, the largest Fermat number that has been factored so far, $F_{11} = 2^{2^{11}} + 1$ is compactly represented as

```
> fermat (t 11)
V E [E,V E [W E [V E []]]]
```

with structural complexity 8. By contrast, its (bijective base-2) binary representation consists of 2048 digits.

Moreover, even "towering up" `mersenne` numbers leads to compact representations. For instance the Catalan sequence is defined as:

```
catalan E = i E
catalan n = s' (exp2 (catalan (s' n)))
```

and Catalan's conjecture states that they are all primes. The conjecture is still unsolved, and as the following example shows, even primality of `catalan 5` is currently intractable.

```
> catalan (t 5)
V (W (V E [W E [E]]) []) []
> n (tsize (catalan (t 5)))
6
> n (bitsize (catalan (t 5)))
170141183460469231731687303715884105727
```

Figures 3 and 4 show the DAG representations of Catalan-Mersenne numbers 5 and 10.

Some other very large primes that are not Mersenne numbers also have compact representations.

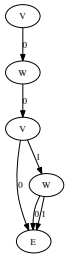

Fig. 3: Catalan-Mersenne number 5

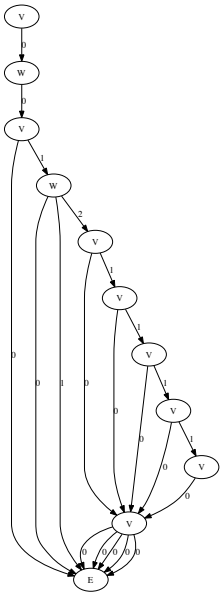

Fig. 4: Catalan-Mersenne number 10

The generalized Fermat prime $27653 * 2^{9167433} + 1$, (currently the 15-the largest prime number) computed as a tree (with 7 shared nodes and structural complexity 30) is:

```
genFermatPrime = s (leftshiftBy n k) where
  n = t (9167433::Integer)
  k = t (27653::Integer)
```

```
> genFermatPrime
V E [E,W (W E []) [W E [],E,
  V E [],E,W E [],W E [E],E,E,W E []],
  E,E,E,W (V E []) [],V E [],E,E]
```

Figure 5 shows the DAG representation of this generalized Fermat prime.

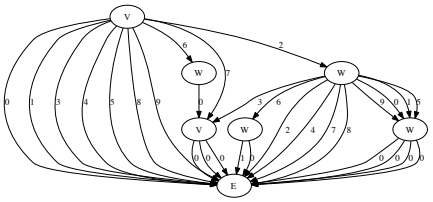

Fig. 5: Generalized Fermat prime

The largest known Cullen prime $6679881*2^{6679881}+1$ computed as tree (with 6 shared nodes and structural complexity 43) is:

```
cullenPrime = s (leftshiftBy n n) where
  n = t (6679881::Integer)
```

```
> cullenPrime
V E [E,W (W E []) [W E [],E,E,E,E,V E [],E,
  V (V E []) [],E,E,V E [],E],E,V E [],E,V E [],
  E,E,E,E,V E [],E,V (V E []) [],E,E,V E [],E]
```

The largest known Woodall prime $3752948 * 2^{3752948} - 1$ computed as a tree (with 6 shared nodes and structural complexity 33) is:

```
woodallPrime = s' (leftshiftBy n n) where
  n = t (3752948::Integer)
```

```
> woodallPrime
V (V E [V E [],E,V E [E],V (V E []) [],
 E,E,E,V E [],V E []]) [E,E,V E [E],
 V (V E []) [],E,E,E,V E [],V E []]
```

The largest known Proth prime $19249 * 2^{13018586} + 1$ computed as a tree is:

```
prothPrime = s (leftshiftBy n k) where
  n = t (13018586::Integer)
  k = t (19249::Integer)
```

```
> prothPrime
V E [E,V (W E []) [V E [],E,W E [],E,E,
  V E [],E,E,E,E,V E [],W E [],E],E,W E [],
  V E [],V E [],V E [],E,E,V E []]
```

The DAG representation of this Proth prime, the largest non-Mersenne prime known by March 2013 has 5 shared nodes and structural complexity 36.

The largest known Sophie Germain prime $18543637900515 * 2^{666667} - 1$ computed as a tree (with 6 shared nodes and structural complexity 56) is:

```
sophieGermainPrime = s' (leftshiftBy n k) where
  n = t (666667::Integer)
  k = t (18543637900515::Integer)
```

```
> sophieGermainPrime
V (W (V E []) [E,E,E,E,V (V E []) [],V E [],E,
   E,W E [],E,E]) [V E [],W E [],W E [],V E [],
   V E [],E,E,V E [],V E [],V E [],V (V E [])
   [],E,V E [],V (V E []) [],V E [],E,W E [],E,
   V E [],V (V E []) []]
```

Figure 6 shows the DAG representation of this prime.

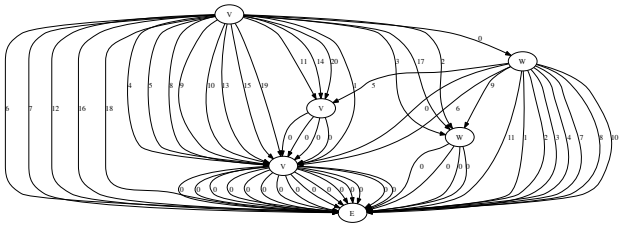

Fig. 6: Largest known Sophie Germain prime

An interesting application might come from the following observation: *Sophie Germain primes* are such that both p and $2p + 1$ are primes. Their generalization, called a *Cunningham chain* is a maximal sequence of primes such that $p_{k+1} = 2p_k + 1$, for example, the sequence chain: 2, 5, 11, 23, 47. As one can notice, they are built with iterated o^k operations, therefore all members of a Cunningham chain are of the form `V k xs`, `V (s k) xs` etc. Interestingly, *primecoins* [7] a digital currency similar to *bitcoins* that "mints" Cunningham chains using Fermat's pseudo-primality test. A topic to explore further would be to see if our representation could help minting primecoins faster, or storing them in a compact form.

The largest known twin primes $3756801695685 * 2^{666669} \pm 1$ computed as a pair of trees (with 7 shared nodes both and structural complexities of 54 and 56) are:

```
twinPrimes = (s' m,s m) where
  n = t (666669::Integer)
  k = t (3756801695685::Integer)
  m = leftshiftBy n k
```

```
> fst twinPrimes
V (W E [E,V E [],E,E,V (V E []) [],
    V E [],E,E,W E [],E,E])
    [E,E,E,W E [],W (V E []) [],V E [],E,V E [],
    E,E,E,E,V E [],E,E,
    V E [],V E [],E,E,E,E,E,E,E,V E [],E,E]
> snd twinPrimes
V E [E,W (V E []) [E,E,E,E,V (V E []) [],
  V E [],E,E,W E [],E,E],E,E,E,
  W E [],W (V E []) [],V E [],E,V E [],
  E,E,E,E,V E [],E,E,V E [],
  V E [],E,E,E,E,E,E,E,V E [],E,E]
```

Figures 7 and 8 show the DAG representation of these twin primes. One can appreciate the succinctness of our representations, given that all these numbers have hundreds of thousands or millions of decimal digits.

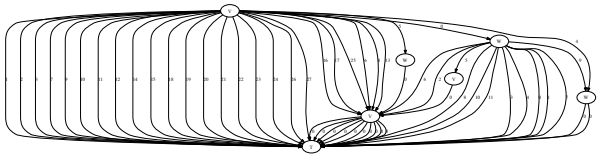

Fig. 7: Largest known twin prime 1

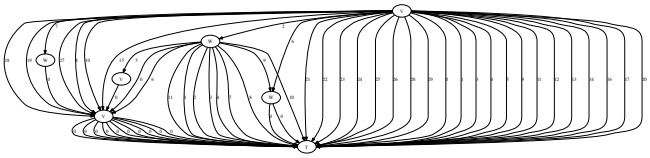

Fig. 8: Largest known twin prime 2

An interesting challenge would be to (re)focus on discovering primes with significantly larger structural complexity than the current record holders, obtained by comparing exclusively their bitsizes.

More importantly, as the following examples illustrate it, *computations* like addition, subtraction and multiplication of such numbers are possible:

```
> sub genFermatPrime (t 2014)
V (V E []) [E,V E [],E,W E [E],V E [W E [E],
  W E [],E,W E [],W E [E],E,E,W E []],V E [],
  E,W (V E []) [],V E [],E,E]
> bitsize (sub prothPrime (t 1234567890))
W E [V E [],E,E,V (V E []) [],E,E,
```

```
  V E [],E,E,E,E,V E [],W E [],E]
> tsize (exp2 (exp2 mersenne48))
V E [E,E,E]
> tsize(leftshiftBy mersenne48 mersenne48)
V E [W E [],E]
> add (t 2) (fst twinPrimes) ==
           (snd twinPrimes)
True
> ilog2 (ilog2 (mul prothPrime cullenPrime))
W E [V E [],E]
> n it
24
```

VI. Related work

This paper is a sequel to [1] where hereditary binary numbers are introduced and "one block of iterated o and i operations at a time" algorithms are described, together with the number-theoretical identities they are relying on.

As an extension of [1], where basic arithmetic algorithms that take advantage of our tree based number representation like multiplication, addition and comparison are described, this paper covers several other arithmetic algorithms and an empirical validation of various performance bounds.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [8] 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, at our best knowledge, occurs in the proof of Goodstein's theorem [9], 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.

Conway's surreal numbers [10] can also be seen as inductively constructed trees. While our focus is on efficient large natural number arithmetic, surreal numbers model games, transfinite ordinals and generalizations of real numbers.

Numeration systems on regular languages have been studied recently, e.g. in [11] and specific instances of them are also known as bijective base-k numbers. Arithmetic packages similar to our bijective base-2 view of arithmetic operations are part of libraries of proof assistants like Coq [12].

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [13]. Their approach is similar as far as a symbolic representation is used. The key difference with our work is that our operations work on tree structures, and as such, they are not based on previously known algorithms.

In [14] and in [15] a binary and respectively multiway tree representation enables arithmetic operations which are simpler but limited in efficiency to a smaller set of "sparse" numbers. In [16] this number representation's connexion to free algebras is explored.

In [17] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. However likewise [14] and [15], and in contrast to those proposed in this paper, they only compress "sparse" numbers, consisting of relatively few `1` bits in their binary representation.

VII. Conclusion and future work

We have validated the complexity bounds of our arithmetic algorithms on hereditarily binary numbers introduced in [1] and we have defined several new arithmetic algorithms for them. Our performance analysis has shown the wide spectrum of best and worst case behaviors of our arithmetic algorithms when compared to Haskell's GMP-based `Integer` operations.

As planned in [1], future work will focus on developing a practical arithmetic library based on a hybrid representation, where the empty leaves of our trees will be replaced with 64-bit integers, to benefit from fast hardware arithmetic on small numbers. In addition, we plan to also cover signed integer as well as rational arithmetic with this hybrid representation.

References

[1] P. Tarau and B. Buckles, "Arithmetic Algorithms for Hereditarily Binary Natural Numbers," in *Proceedings of SAC'14, ACM Symposium on Applied Computing, PL track*. Gyeongju, Korea: ACM, Mar. 2014.

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

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

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

[5] J. W. Bruce, "A Really Trivial Proof of the Lucas-Lehmer Test," *The American Mathematical Monthly*, vol. 100, no. 4, pp. 370–371, 1993.

[6] "Great Internet Mersenne Prime Search," 2013, http://www.mersenne.org/.

[7] V. Buterin, "Primecoin: the cryptocurrency whose mining is actually useful." *Bitcoin Magazine*, July 8 2013. [Online]. Available: http://bitcoinmagazine.com/primecoin-the-cryptocurrency-whose-mining-is-actually-useful

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

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

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

[11] M. Rigo, "Numeration systems on a regular language: arithmetic operations, recognizability and formal power series," *Theoretical Computer Science*, vol. 269, no. 1-2, pp. 469 – 498, 2001.

[12] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2012, version 8.4. [Online]. Available: http://coq.inria.fr

[13] O. Kiselyov, W. E. Byrd, D. P. Friedman, and C.-c. Shan, "Pure, declarative, and constructive arithmetic relations (declarative pearl)," in *FLOPS*, 2008, pp. 64–80.

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

[15] P. Tarau, "Declarative Modeling of Finite Mathematics," in *PPDP '10: Proceedings of the 12th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA: ACM, 2010, pp. 131–142.

[16] ——, "Computing with Free Algebras," in *Proceedings of SYNASC 2012*, A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. Watt, and D. Zaharie, Eds. IEEE, Jan. 2013, pp. 15–22, invited talk.

[17] 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.

APPENDIX

Implementing Haskell's Number and Order-related Type Classes

In [1] algorithms working "one block of o^n or i^m applications at a time" are introduced for various arithmetic operations on type `T`, and implemented as the following functions:

- `add` : addition
- `sub` : subtraction
- `mul` : multiplication
- `divide` : integer division
- `remainder` : remainder of integer division
- `cmp`: comparison operation, returning `LT,EQ,GT`
- `exp2`: exponent of 2
- `leftshiftBy x y`: specialized multiplication $2^x y$
- `rightshiftBy x y`: specialized integer division $\frac{y}{2^x}$
- `pow`: power operation
- `bitsize`: computing the bitsize of a bijective base-2 representation
- `tsize`: computing the structural complexity of a tree-represented number
- `dual`: flipping of topmost `V` and `W` constructors, corresponding to flipping of all `o` and `i` applications.

We will make here the type `T` an instance of Haskell's predefined number and order-related type classes like `Ord` and `Num`, etc., to enable the usual arithmetic operation and comparison syntax.

```
instance Ord T where compare = cmp
instance Num T where
  (+) = add
  (-) = sub
  (*) = mul
  fromInteger = t
  abs = id
  signum E = E
  signum _ = V E []
instance Integral T where
  quot = divide
  div = divide
  rem = remainder
  mod = remainder
  quotRem = div_and_rem
  divMod = div_and_rem
  toInteger = n
instance Real T where
  toRational = toRational . n
instance Enum T where
  fromEnum = fromEnum . n
  toEnum = t . f where
    f:: Int→Integer
    f = toEnum
  succ = s
  pred = s'
```

Note that as Haskell does not have a built-in natural number class, we mapped `abs` and `signum` to identity `id` and the constant value corresponding to 1, `V E []`.

The following example illustrates the use of the + operation directly on objects of type `T`:

```
> t 3
V (V E []) []
> t 4
W E [E]
> (V (V E []) []) + (W E [E])
V (W E []) []
> n it
7
```

Haskell definitions of arithmetic operations on Hereditarily Binary Numbers from [1]

As a convenience to the reviewers, we include here the Haskell code imported from the paper [1], a literate program explaining it in full detail (see draft at http://www.cse.unt.edu/~tarau/research/2014/HBin.pdf). We have trimmed the code to only cover the functions actually used in this paper and their call graph.

```
module HBin where

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

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

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)

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

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

e_ E = True
e_ _ = False

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

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

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)))

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

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

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)

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)))))

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)

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)

add E y = y
add x E = x

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)

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)

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)

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)

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)

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)

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)

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

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)

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

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)

ilog2 x = bitsize (s' x)

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

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))
```