New Arithmetic Algorithms for Hereditarily Binary Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

SYNASC'2014

– research supported by NSF grant 1423324 –

Overview

- our tree-based *hereditarily binary numbers* apply recursively a run-length compression mechanism
- they enable performing arithmetic computations symbolically and lift tractability of computations to be limited by the representation size of their operands rather than by their bitsizes
- this paper describes several new arithmetic algorithms on hereditarily binary numbers
  1. that are within constant factors from their traditional counterparts for their average case behavior
  2. are super-exponentially faster on some “interesting” giant numbers
  3. $\Rightarrow$ make tractable important computations that are impossible with traditional number representations

Outline

Related work

- a hereditary number system occurs in the proof of Goodstein's theorem (1947) , where replacement of finite numbers on a tree's branches by the ordinal ω allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates
- notations vs. computations
  - notations for very large numbers have been invented in the past ex: Knuth's up-arrow
  - in contrast to our tree-based natural numbers, such notations are not closed under successor, addition and multiplication
- this paper is a sequel to our ACM SAC'14 where computations with hereditarily binary numbers are introduced
- in our PPDP'14 paper: boolean operations, encodings of hereditarily finite sets and multisets with hereditarily binary numbers are described as well as size-proportionate bijective Gödel numberings of term algebras

Bijective base-2 numbers as iterated function applications

Natural numbers can be seen as iterated applications of the functions

- $o(x) = 2x + 1$
- $i(x) = 2x + 2$

corresponding the so called *bijective base-2* representation.

- $1 = o(0)$,
- $2 = i(0)$,
- $3 = o(o(0))$,
- $4 = i(o(0))$,
- $5 = o(i(0))$

Iterated applications of *o* and *i*: some useful identities

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

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

and in particular

$$o^n(0) = 2^n - 1 \tag{3}$$

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

Hereditarily binary numbers

Hereditarily binary numbers are defined as the Haskell type $\mathbb{T}$:

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

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

- 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
- note: `x` counts `x+1` applications, as we start at `0`

The arithmetic interpretation of hereditarily binary numbers

Definition

The bijection $n : \mathbb{T} \to \mathbb{N}$ defines the unique natural number associated to a term of type $\mathbb{T}$. Its inverse is denoted $t : \mathbb{N} \to \mathbb{T}$.

$$n(t) = \begin{cases} 0 & \text{if } t = \texttt{E}, \\ 2^{n(x)+1} - 1 & \text{if } t = \texttt{V x []}, \\ (n(u)+1)2^{n(x)+1} - 1 & \text{if } t = \texttt{V x (y:xs)} \text{ and } u = \texttt{W y xs}, \\ 2^{n(x)+2} - 2 & \text{if } t = \texttt{W x []}, \\ (n(u)+2)2^{n(x)+1} - 2 & \text{if } t = \texttt{W x (y:xs)} \text{ and } u = \texttt{V y xs}. \end{cases} \tag{5}$$

ex: the computation of $n(\texttt{W (V E []) [E,E,E]})$ expands to $(((2^{0+1}-1+2)2^{0+1}-2+1)2^{0+1}-1+2)2^{2^{0+1}-1+1}-2 = 42$.

Examples

- each term canonically represents the corresponding natural number
- the first few natural numbers are:

```
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])
```

An overview of constant average time and worst case constant or log^* time operations with hereditarily binary numbers

- introduced in our ACM SAC'14 paper:
- mutually recursive successor s and predecessor s'
- defined on top of s and s':
  - $o(x) = 2x + 1$ and $i(x) = 2x + 2$
  - their inverses o' and i'
  - recognizers of odd and even numbers *o_* and *i_*
  - double *db* and its left inverse *hf*
  - power of two *exp*2
- $\Rightarrow$ computations favoring towers of exponents and numbers in their "neighborhood"
- $\Rightarrow$ computations favoring sparse numbers (with a lot of 0s) or dense numbers (with a lot of 1s)

Other operations on hereditarily binary numbers

algorithms working "one block of *o* or *i* applications at a time" for:

- `add` : addition
- `sub` : subtraction
- `cmp`: comparison operation, returning `LT,EQ,GT`
- `leftshiftBy x y`: specialized multiplication $2^x y$
- `rightshiftBy x y`: specialized integer division $\frac{y}{2^x}$
- `bitsize`: computing the bitsize of a bijective base-2 representation
- `tsize`: computing the structural complexity of a tree-represented number

Towers of exponents can grow tall, provided they are finite ... (credit: Bruegel's Tower of Babel)

General multiplication

- 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

The following holds:

$$o^n(a)o^m(b) = o^{n+m}(ab+a+b) - o^n(a) - o^m(b) \tag{6}$$

Proof.

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

Another identity used for multiplication

Proposition

$$i^n(a)i^m(b) = i^{n+m}(ab+2(a+b+1))+2-i^{n+1}(a)-i^{m+1}(b) \tag{7}$$

Proof.

By (2), 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 Haskell code follows these identities closely
- we use a small subset of Haskell as an executable notation for our functions
- $\Rightarrow$ the paper is a *literate program*

Power

- we specialize our multiplication for a faster squaring operation:

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

$$(i^n(a))^2 = i^{2n}(a^2 + 2(2a+1)) + 2 - 2i^{n+1}(a) \tag{9}$$

- power by squaring, in Haskell:

```
pow :: T→T→T

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
```

Division,Integer square root

- division - the traditional algorithm - see paper
- integer square root - more interesting (with Newton's method):

```
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
```

Modular power

- the modular power operation $x^y (mod\ m)$ is optimized to avoid the creation of large intermediate results
- we combine “power by squaring” and pushing the modulo operation inside the inner function

```
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)
```

Primality tests

Lucas-Lehmer primality test - good at finding Mersenne primes

- used for the discovery of all the record holder largest known prime numbers of the form $2^p - 1$ with p prime
- 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

Miller-Rabin probabilistic primality test

- most of the code is routine (see paper) - uses in cryptography
- $\nu_2(x)$: *dyadic valuation of* x, i.e., the largest exponent of 2 that divides x
- interestingly, $dyadicSplit(k) = (k, \frac{k}{2^{\nu_2(k)}})$ used in the algorithms, can be implemented as an average constant time operation:

```
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
```

Performance evaluation

Benchmark	Integer	tree type $\mathbb{T}$
$2^{2^{30}}$	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

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

Compact representation of some record-holder giant numbers

`mersenne48 = s' (exp2 (t 57885161))`

it has a bit-size of 57885161, but its compressed tree representation is: `V (W E [V E [],E,E,V (V E []) [],W E [E],E,E,V E [],V E [],W E [],E,E]) []`

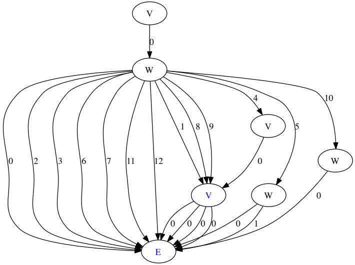

Figure: Largest known prime number discovered in January 2013: the 48-th Mersenne prime, represented as a DAG

Catalan conjecture: they are all primes – intractable

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

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

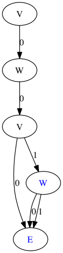

Figure: Catalan Mersenne number 5

Largest known Sophie Germain prime

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

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 []) []]

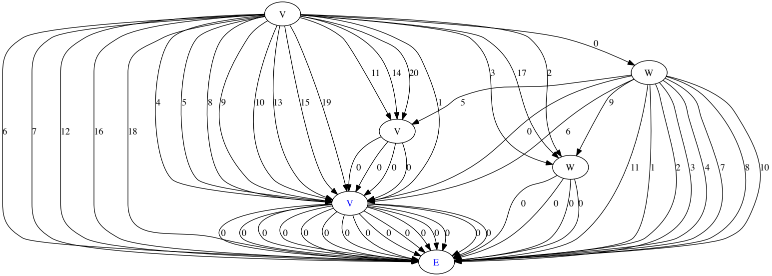

Figure: Largest known Sophie Germain prime

Cunningham Chains

- *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`
- 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 ...`
- *primecoins* : a digital currency similar to *bitcoins* that "mints" Cunningham chains using Fermat's pseudo-primality test
- open problem: could our representation could help minting primecoins faster, or storing them in a compact form?

Conclusion and future work

- we have shown previously that hereditarily binary numbers favor by a super-exponential factor, arithmetic operations on numbers in neighborhoods of towers of exponents of two
- we have validated the complexity bounds of our arithmetic algorithms on hereditarily binary numbers
- 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
- future work
  - 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
  - we plan to also cover signed integer as well as rational arithmetic with this hybrid representation

Links

- the paper is a literate program, our Haskell code is at `http://www.cse.unt.edu/~tarau/research/2014/HBinX.hs`
- it imports code from the our ACM SAC'14 paper at `http://www.cse.unt.edu/~tarau/research/2014/HBin.hs`
- a draft version of the ACM SAC'14 paper is at `http://www.cse.unt.edu/~tarau/research/2014/HBin.pdf`
- collection encoding and boolean operations with HBNs at `http://www.cse.unt.edu/~tarau/research/2014/HBS.pdf`
- an alternative Scala based implementation of HBNs is at at: `http:/code.google.com/p/giant-numbers/`