On the Arithmetic of Recursively Run-length Compressed Natural Numbers

Paul Tarau
Dept. of Computer Science and Engineering
Univ. of North Texas
Denton, USA
tarau@cse.unt.edu

***Abstract*—We study arithmetic properties of a new tree-based number representation, *Recursively run-length compressed* natural numbers, defined by applying recursively a run-length encoding of their binary digits.**

Our representation supports novel algorithms that, in the best case, collapse the complexity of various computations by super-exponential factors and in the worse case are within a constant factor of their traditional counterparts.

As a result, it opens the door to a new world, where arithmetic operations are limited by the structural complexity of their operands, rather than their bitsizes.

***Keywords*-run-length compressed numbers, hereditary numbering systems, arithmetic algorithms for giant numbers, structural complexity of natural numbers**

I. Introduction

The set ℕ of natural numbers is in bijection with several naturally tree-represented constructs like hereditarily finite sets or countable ordinals. Their correspondence with some fundamental free algebras is well-known to people working in type theory and has also been investigated in [1] as the basis of arithmetic computations on the free algebra of binary trees.

Recent foundational developments like *homotopy type theory* use multiple isomorphic instances of ℕ as an argument for the *univalent view* of the foundations of mathematics [2], which is built around a type theory where "equivalent" types are identified.

Our extensive study of data type transformations described in [3], [4], [5], [6] is also centered around natural numbers used as shared encodings of various data structures and mathematical objects. Of particular interest, in this context, is the ability to compute with unconventional data types in a way that is equivalent to their natural number counterparts, coming mostly form a theoretical curiosity on how the recursive equations defining their operations correspond to their Peano arithmetic counterparts [7], [1]. More recently, on the edge of practicality, we have studied data types that turned out to be able to handle very large numbers, with a significant reduction in the case of "sparse" number representations [8].

Number representations have evolved over time from the unary "cave man" representation 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.

While *notations* like Knuth's "up-arrow" [9] or tetration are useful in describing 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 cannot be expressed with the notation anymore. More exotic notations like Conway's surreal numbers [10] involve uncountable cardinalities (they contain real numbers as a subset) and are more useful for modeling game-theoretical algorithms rather than common arithmetic computations.

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 worse 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 practically constant time and a number of other operations benefit from significant complexity reductions.

For the curious reader, it is basically a *hereditary number system* [11], based on recursively applied *run-length* compression of the usual binary digit notation.

A concept of structural complexity is introduced, based on the size of our tree representations and algorithms favoring large numbers of small structural complexity are designed for arithmetic operations.

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/rrl.hs . We hope that this will encourage the reader to experiment interactively and validate the technical correctness of our claims. The **Appendix** contains a quick overview of the subset of Haskell we are using as our executable function notation.

The paper is organized as follows. Section II introduces our tree represented recursively run-length compressed natural numbers. Section III describes practically constant time successor and predecessor operations on tree-represented numbers. Section IV describes novel algorithms for arithmetic operations taking advantage of our number representation. Section V defines a concept of structural complexity and studies best and worse cases. Section VI describes an example of computations with very large numbers using recursively run-length compressed numbers. Section VII discusses related work. Section VIII concludes the paper and discusses future work.

II. Recursively run-length compressed natural numbers as a data type

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

Definition 1: The data type $\mathbb{T}$ of the set of recursively run-length compressed numbers is defined by the Haskell declaration:

```
data T = F [T] deriving (Eq,Show,Read)
```

that automatically derives the equality relation "==", as well as reading and string representation. The data type $\mathbb{T}$ corresponds precisely to ordered rooted multiway trees with empty leaves, but for shortness, we will call the objects of type $\mathbb{T}$ *terms*. The "arithmetic intuition" behind the type $\mathbb{T}$ is the following:

- The term `F []` (empty leaf) corresponds to zero
- in the term `F xs`, each `x` on the list `xs` counts the number `x+1` of `0` digits followed by alternating counts of `1` and `0` digits
- the same principle is applied recursively for the counters, until the empty sequence is reached.

One can see this process as run-length compressed base-2 numbers, unfolded as trees with empty leaves, after applying the encoding recursively.

By convention, as the last (and most significant) digit is `1`, the last count on the list `xs` is for `1` digits. The following simple fact allows inferring parity from the number of subtrees of a tree.

Proposition 1: If the length of `xs` in `F xs` is odd, then `F xs` encodes an odd number, otherwise it encodes an even number.

Proof: Observe that as the highest order digit is always a `1`, the lowest order digit is also `1` when length of the list of counters is odd, as counters for `0` and `1` digits alternate. ■

This ensures the correctness of the Haskell definitions of the predicates `odd_` and `even_`, the last one defined to be true for non-empty trees only.

```
oddLen [] = False
oddLen [_]= True
oddLen (_:xs) = not (oddLen xs)

odd_ (F []) = False
odd_ (F (_:xs)) = not (oddLen xs)

even_ (F []) = False
even_ (F (_:xs)) = oddLen xs
```

Note that while these predicates work in time proportional to the length of the list `xs` in `F xs`, with a (dynamic) array based list representation or by keeping track of the length explicitly, one can assume that they are constant time, as we will do in the rest of the paper.

Definition 2: The function $n : \mathbb{T} \rightarrow \mathbb{N}$ shown in equation `(1)` defines the unique natural number associated to a term of type $\mathbb{T}$.

$$n(u) = \begin{cases} 0 & \text{if } u = \texttt{F []}, \\ 2^{n(\texttt{x})+1} n(\texttt{F xs}) & \text{if } u = \texttt{F (x:xs)} \text{ is even,} \\ 2^{n(\texttt{x})+1} n(\texttt{F xs}) - 1 & \text{if } u = \texttt{F (x:xs)} \text{ is odd.} \end{cases} \quad (1)$$

For instance, the computation of `n(F [F [],F [F [],F []]])` $= 14$ expands to $(2^{0+1}(2^{(2^{0+1}(2^{0+1}-1))+1}-1))$. The Haskell equivalent[1] of equation (1) is:

```
n (F []) = 0
n a@(F (x:xs)) | even_ a = 2^(n x + 1)*(n (F xs))
n a@(F (x:xs)) | odd_ a = 2^(n x + 1)*(n (F xs)+1)-1
```

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

```
0: F []
1: F [F []])
2: F [F [],F []]
3: F [F [F []]]
```

Definition 3: The function $t : \mathbb{N} \rightarrow \mathbb{T}$ defines the unique tree of type $\mathbb{T}$ associated to a natural number as follows:

```
t 0 = F []
t k | k>0 = F zs where
  (x,y) = if even k then split0 k else split1 k
  F ys = t y
  zs = if x==0 then ys else t (x-1) : ys
```

It uses the helper functions `split0` and `split1` that extract a block of contiguous `0` digits and, respectively, `1` digits from the lower end of a binary number.

```
split0 z | z> 0 && even z = (1+x,y) where
  (x,y) = split0  (z `div` 2)
split0 z = (0,z)

split1 z | z>0 && odd z = (1+x,y) where
  (x,y) = split1  ((z-1) `div` 2)
split1 z = (0,z)
```

Note that `div` is integer division. The following holds:

Proposition 2: Let `id` denote $\lambda x.x$ and $\circ$ function composition. Then, on their respective domains

$$t \circ n = id, \quad n \circ t = id \quad (2)$$

Proof: By induction, using the arithmetic formulas defining the two functions. ■

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

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

```
s (F []) = F [F []] -- 1
s (F [x]) = F [x,F []] -- 2

s a@(F (F []:x:xs)) | even_ a = F (s x:xs) -- 3
s a@(F (x:xs)) | even_ a = F (F []:s' x:xs) -- 4

s a@(F (x:F []:y:xs)) | odd_ a = F (x:s y:xs) -- 5
s a@(F (x:y:xs)) | odd_ a = F (x:F []:(s' y):xs) -- 6
```

[1]As an Haskell note, for the reader unfamiliar with the language, the pattern `a@p` indicates that the parameter `a` is the same as its expanded version matching the patten `p`.

```
s' (F [F []]) = F [] -- 1
s' (F [x,F []]) = F [x] -- 2

s' b@(F (x:F []:y:xs)) | even_ b = F (x:s y:xs) -- 6
s' b@(F (x:y:xs)) |even_ b = F (x:F []:s' y:xs) -- 5

s' b@(F (F []:x:xs)) | odd_ b = F (s x:xs) -- 4
s' b@(F (x:xs)) | odd_ b = F (F []:s' x:xs) -- 3
```

Note that the two functions work "a block of `0` or `1` digits at a time", and are based on simple arithmetic observations about the behavior of these blocks when incrementing or decrementing a binary number by `1`. The following holds:

Proposition 3: Denote $\mathbb{T}^+ = \mathbb{T} - \{$`F []`$\}$. 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 marked with the number `-- k` in `s` correspond one by one to patterns marked by `-- k` in `s'` and vice versa. ■

More generally, it can be shown that Peano's axioms hold and as a result $< \mathbb{T}, F[], s >$ is a *Peano algebra*.

Note also that calls to `s,s'` in `s` or `s'` happen on terms that are (roughly) logarithmic in the bitsize of their operands. One can therefore assume that their complexity, computed by an *iterated logarithm*, is practically constant, under the hypothesis that length information (and in accordance to Prop. 1, also parity information) is kept explicitly.

IV. Arithmetic operations

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

A. A few low complexity operations

Doubling a number `db` and reversing the `db` operation (`hf`) are quite simple. For instance, `db` proceeds by adding a new counter for odd numbers and incrementing the first counter for even ones.

```
db (F []) = F []
db a@(F xs) | odd_ a = F (F []:xs)
db a@(F (x:xs)) | even_ a = F (s x:xs)
```

```
hf (F []) = F []
hf (F (F []:xs)) = F xs
hf (F (x:xs)) = F (s' x:xs)
```

Note that such efficient implementations follow directly from simple number theoretic observations.

For instance, `exp2`, computing an exponent of 2 , has the following definition in terms of `s'`.

```
exp2 (F []) = F [F []]
exp2 x = F [s' x,F []]
```

Proposition 4: Assuming `s,s'` constant time, `db,hf` and `exp2` are also constant time.

Proof: It follows by observing that only at most 1 call to `s,s'` is made in each definition. ■

B. Reduced complexity addition and subtraction

We derive efficient addition and subtraction that *work on one run-length compressed block at a time*, rather than by individual `0` and `1` digit steps.

First we define the functions `leftshiftBy`, `leftshiftBy'` and `leftshiftBy''`corresponding to $2^n\ k$, $(\lambda x.2x+1)^n(k)$ and $(\lambda x.2x+2)^n(k)$.

```
leftshiftBy (F []) x = x
leftshiftBy _ (F []) = F []
leftshiftBy x k@(F xs) | odd_ k = F ((s' x):xs)
leftshiftBy x k@(F (y:xs)) | even_ k = F (add x y:xs)
```

```
leftshiftBy' x k = s' (leftshiftBy x (s k))

leftshiftBy'' x k = s' (s' (leftshiftBy x (s (s k))))
```

The last two correspond to the identities:

$$(\lambda x.2x+1)^n(k) = 2^n(k+1) - 1 \tag{3}$$

$$(\lambda x.2x+2)^n(k) = 2^n(k+2) - 2 \tag{4}$$

They are part of a chain of mutually recursive functions as they are already referring to the `add` function, to be implemented later. Note also that instead of naively iterating, they implement a more efficient algorithm, working "one block at a time". For instance, when detecting that its argument counts a number of `1`, `leftshiftBy'` just increments that count. As a result, performance is (roughly) logarithmic rather than linear in terms of the bitsize of argument `n`. We will also use this property for implementing a low complexity multiplication by exponent of 2 operation.

The following holds:

Proposition 5: Assuming `s,s'` constant time, `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. ■

We are now ready for defining addition. The base cases are

```
add (F []) y = y
add x (F []) = x
```

In the case when both terms represent even numbers, the two blocks add up to an even block of the same size.

```
add x@(F (a:as)) y@(F (b:bs)) |even_ x && even_ y =
 f (cmp a b) where
  f EQ = leftshiftBy (s a) (add (F as) (F bs))
  f GT = leftshiftBy (s b)
         (add (leftshiftBy (sub a b) (F as)) (F bs))
  f LT = leftshiftBy (s a) (add (F as)
         (leftshiftBy (sub b a) (F bs)))
```

In the case when the first term is even and the second odd, the two blocks add up to an odd block of the same size.

```
add x@(F (a:as)) y@(F (b:bs)) |even_ x && odd_ y =
 f (cmp a b) where
  f EQ = leftshiftBy' (s a) (add (F as) (F bs))
  f GT = leftshiftBy' (s b)
         (add (leftshiftBy (sub a b) (F as)) (F bs))
  f LT = leftshiftBy' (s a)
         (add (F as) (leftshiftBy' (sub b a) (F bs)))
```

In the case when the second term is even and the first odd the two blocks also add up to an odd block of the same size.

```
add x y |odd_ x && even_ y = add y x
```

In the case when both terms represent odd numbers, we use the identity (5):

$$(\lambda x.2x+1)^k(x) + (\lambda x.2x+1)(y) = (\lambda x.2x+2)^k(x+y) \tag{5}$$

```
add x@(F (a:as)) y@(F (b:bs)) | odd_ x && odd_ y =
 f (cmp a b) where
  f EQ = leftshiftBy'' (s a) (add (F as) (F bs))
  f GT = leftshiftBy'' (s b)
          (add (leftshiftBy' (sub a b) (F as)) (F bs))
  f LT = leftshiftBy'' (s a)
          (add (F as) (leftshiftBy' (sub b a) (F bs)))
```

Note the presence of 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 (F []) = x
sub x@(F (a:as)) y@(F (b:bs)) | even_ x && even_ y =
 f (cmp a b) where
  f EQ = leftshiftBy (s a) (sub (F as) (F bs))
  f GT = leftshiftBy (s b)
         (sub (leftshiftBy (sub a b) (F as)) (F bs))
  f LT = leftshiftBy (s a)
         (sub (F as) (leftshiftBy (sub b a) (F bs)))
```

The case when both terms represent `1` blocks the result is an `0` block

```
sub x@(F (a:as)) y@(F (b:bs)) | odd_ x && odd_ y =
 f (cmp a b) where
  f EQ = leftshiftBy (s a) (sub (F as) (F bs))
  f GT = leftshiftBy (s b)
         (sub (leftshiftBy' (sub a b) (F as)) (F bs))
  f LT = leftshiftBy (s a)
         (sub (F as) (leftshiftBy' (sub b a) (F bs)))
```

The case when the first block is `1` and the second is a `0` block:

```
sub x@(F (a:as)) y@(F (b:bs)) | odd_ x && even_ y =
 f (cmp a b) where
  f EQ = leftshiftBy' (s a) (sub (F as) (F bs))
  f GT = leftshiftBy' (s b)
         (sub (leftshiftBy' (sub a b) (F as)) (F bs))
  f LT = leftshiftBy' (s a)
         (sub (F as) (leftshiftBy (sub b a) (F bs)))
```

Finally, when the first block is `0` and the second is `1` an identity dual to (5) is used:

```
sub x@(F (a:as)) y@(F (b:bs)) | even_ x && odd_ y =
f (cmp a b) where
  f EQ = s (leftshiftBy (s a) (sub1 (F as) (F bs)))
  f GT = s (leftshiftBy (s b)
         (sub1 (leftshiftBy (sub a b) (F as)) (F bs)))
  f LT = s (leftshiftBy (s a)
         (sub1 (F as) (leftshiftBy' (sub b a) (F bs))))

sub1 x y = s' (sub x y)
```

C. Defining a total order: 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 binary digits 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 6: Let `bitsize` count the number of digits of a base-2 number, with the convention that it is `0` for `0`. Then `bitsize`(x) <`bitsize`$(y) \Rightarrow x < y$.

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

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

```
cmp (F []) (F []) = EQ
cmp (F []) _ = LT
cmp _ (F []) = GT
cmp (F [F []]) (F [F [],F []]) = LT
cmp  (F [F [],F []]) (F [F []]) = GT
cmp x y | x' /= y'  = (cmp x' y') where
  x' = bitsize x
  y' = bitsize y
cmp (F xs) (F ys) =
  compBigFirst True True
    (F (reverse xs)) (F (reverse ys))
```

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

Proposition 7: Assuming two terms of the same bitsizes, the one with its first before its highest digit `1` is larger than the one with its first before its highest digit 0.

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

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

```
compBigFirst _ _ (F []) (F []) = EQ
compBigFirst False False (F (a:as)) (F (b:bs)) =
  f (cmp a b) where
    f EQ = compBigFirst True True (F as) (F bs)
    f LT = GT
    f GT = LT
compBigFirst True True (F (a:as)) (F (b:bs)) =
  f (cmp a b) where
    f EQ = compBigFirst False False (F as) (F bs)
    f LT = LT
    f GT = GT
compBigFirst False True x y = LT
compBigFirst True False x y = GT
```

The function `bitsize` computes the number of digits, except that we define it as `F []` for `F []`, corresponding to `0`. It works by summing up (using Haskell's `foldr`) the counts of 0 and 1 digit blocks composing a tree-represented natural number.

```
bitsize (F []) = (F [])
bitsize (F (x:xs)) = s (foldr add1 x xs)

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

The base-2 integer logarithm is then computed as

```
ilog2 = s' . bitsize
```

D. Reduced complexity general multiplication

Devising a similar optimization as for add and sub for multiplication is actually easier.

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

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

  mul1 (F []) _ = F []
  mul1 a@(F (x:xs)) y | even_ a =
    leftshiftBy (s x) (mul1 (F xs) y)
  mul1 a y | odd_ a =
    add y (mul1 (s' a) y)
```

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

```
*RRL> let term1 = sub (exp2
   (exp2 (t 12345))) (exp2 (t 6789))
*RRL> let term2 = add (exp2
   (exp2 (t 123))) (exp2 (t 456789))
*RRL> bitsize (bitsize (mul term1 term2))
F [F [],F [],F [],F [F [],F []],
  F [F [],F [],F []],F [F []]]
*RRL> n it
12346
```

This hints toward a possibly new computational paradigm where arithmetic operations are not limited by the size of their operands, but only by their "structural complexity". We will make this concept more precise in section V.

E. Power

We first specialize our multiplication for a squaring operation,

```
square x = mul x x
```

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

```
pow _ (F []) = F [F []]
pow a@(F (x:xs)) b | even_ a =
 F (s' (mul (s x) b):ys) where
  F ys = pow (F xs) b
pow a b@(F (y:ys)) | even_ b =
 pow (superSquare y a) (F ys) where
  superSquare (F []) x = square x
  superSquare k x = square (superSquare (s' k) x)
pow x y = mul x (pow x (s' y)) -- ?
```

It works well with fairly large numbers, by also benefiting from efficiency of multiplication on terms with large blocks of 0 and 1 digits:

```
*RRL> n (bitsize (pow (t 10) (t 100)))
333
*RRL> pow (t 32) (t 10000000)
F [F [F [F [],F [F []]],F [F [F []],F []],
   F [F [F []]],F [],F [],F [],F [F [F []],
   F []],F [],F []],F []]
```

V. 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 (F xs) =foldr add1 (F []) (map tsize xs)
```

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

$$c(t) = \begin{cases} 0 & \text{if } \mathtt{t} = \ \mathtt{F}\ [], \\ \sum_{x \in \mathtt{xs}} (1 + c(x)) & \text{if } \mathtt{t} = \mathtt{F}\ \ \mathtt{xs}. \end{cases} \tag{6}$$

The following holds:

Proposition 8: For all terms $t \in \mathbb{T}$, tsize t $\leq$ bitsize t.

Proof: By induction on the structure of t, 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:

```
*RRL> map (n.tsize.t) [0,100,1000,10000]
[0,7,9,13]
*RRL> map (n.tsize.t) [2^16,2^32,2^64,2^256]
[5,6,6,6]
*RRL> map (n.bitsize.t) [2^16,2^32,2^64,2^256]
[17,33,65,257]
```

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

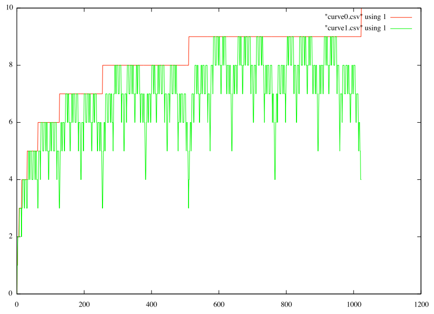

Fig. 1: Structural complexity (yellow line) bounded by bitsize (red line) from 0 to $2^{10} - 1$

After defining the higher order function iterated that applies k times the function f

```
iterated f (F []) 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 (F []) where
  wTree x = F [x]
```

and a worse case

```
worseCase k = iterated (s.db.db) k (F [])
```

The following examples illustrate these functions:

```
*RRL> bestCase (t 4)
F [F [F [F [F []]]]]
*RRL> n it
65535
*RRL> n (bitsize (bestCase (t 4)))
16
*RRL> n (tsize (bestCase (t 4)))
4

*RRL> worseCase (t 4)
F [F [],F [],F [],F [],F [],F [],F []]
*RRL> n it
85
*RRL> n (bitsize (worseCase (t 4)))
7
*RRL> n (tsize (worseCase (t 4)))
7
```

The function `bestCase` computes the iterated exponent of 2 (tetration) and then applies the predecessor to it. For $k = 4$ it corresponds to

$$(2^{(2^{(2^{(2^{0+1}-1)+1}-1)+1}-1)+1}-1) = 2^{2^{2^2}}-1 = 65535.$$

The average space-complexity of the representation is related to the average length of the *integer partitions of the bitsize of a number* [12]. Intuitively, the shorter the partition in alternative blocks of `0` and `1` digits, the more significant the compression is, but the exact study, given the recursive application of run-length encoding, is likely to be quite intricate.

Note also that our concept of structural complexity is only a weak approximation of Kolmogorov complexity [13]. For instance, the reader might notice that our worse case example is computable by a program of relatively small size. However, as `bitsize` is an upper limit to `tsize`, we can be sure that we are within constant factors from the corresponding bitstring computations, even on random data of high Kolmogorov complexity.

Note also that an alternative concept of structural complexity can be defined by considering the (vertices+edges) size of the DAG obtained by folding together identical subtrees.

VI. Computing the Collatz/Syracuse sequence for huge numbers

As an interesting application, that achieves something one cannot do with traditional arbitrary bitsize Integers is to explore the behavior of interesting conjectures in the new world of numbers limited not by their sizes but by their structural complexity. The Collatz conjecture states that the function

$$collatz(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{7}$$

reaches 1 after a finite number of iterations. An equivalent formulation, by grouping together all the division by 2 steps, is the function:

$$collatz'(x) = \begin{cases} \frac{x}{2^{\nu_2(x)}} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{8}$$

where $\nu_2(x)$ denotes the *dyadic valuation of x*, i.e., the largest exponent of 2 that divides x. One step further, the *syracuse function* is defined as the odd integer k' such that $n = 3k+1 = 2^{\nu(n)}k'$. One more step further, by writing $k' = 2m+1$ we get a function that associates $k \in \mathbb{N}$ to $m \in \mathbb{N}$.

The function `tl` computes efficiently the equivalent of

$$tl(k) = \frac{\frac{k}{2^{\nu_2(k)}}-1}{2} \tag{9}$$

Together with its `hd` counterpart, it is defined as

```
hd = fst.decons

tl = snd.decons

decons (F []) = undefined
decons a@(F (x:xs)) | even_ a = (s x,hf (s' (F xs)))
decons a = (F [],hf (s' a))
```

where the function `decons` is the inverse of

```
cons (x,y) = leftshiftBy x (s (db y))
```

corresponding to $2^x\ (2y+1)$.

Then our variant of the *syracuse function* corresponds to

$$syracuse(n) = tl(3n+2) \tag{10}$$

which is defined from $\mathbb{N}$ to $\mathbb{N}$ and we can be implemented as

```
syracuse n = tl (add n (db (s n)))
```

The function `nsyr` computes the iterates of this function, until (possibly) stopping:

```
nsyr (F []) = [F []]
nsyr n = n : nsyr (syracuse n)
```

It is easy to see that the Collatz conjecture is true if and only if `nsyr` terminates for all n, as illustrated by the following example:

```
*RRL> map n (nsyr (t 2014))
[2014,755,1133,1700,1275,1913,2870,1076,807,
   1211,1817,2726,1022,383,575,863,1295,1943,
   2915,4373,6560,4920,3690,86,32,24,18,3,5,
   8,6,2,0]
```

The next examples will show that computations for `nsyr` can be efficiently carried out for giant numbers that, with the traditional bitstring representation, would easily overflow the memory of a computer with as many transistors as the atoms in the known universe.

And finally something we are quite sure has never been computed before, we can also start with a *tower of exponents 100 levels tall*:

```
*RRL> take 100 (map(n.tsize)(nsyr (bestCase (t 100))))
[100,199,297,298,300,...,440,436,429,434,445,439]
```

Note that we have only computed the decimal equivalents of the structural complexity `tsize` of these numbers, that obviously would not fit themselves in a decimal representation.

VII. Related work

We will briefly describe here some related work that has inspired and facilitated this line of research and will help to put our past contributions and planned developments in context.

This paper is an adaptation of our online draft at the Cornell `arxiv` repository [14], which describes a more complex hereditary number system (based on run-length encoded "bijective base 2" numbers, first introduced in [15] pp. 90-92 as "m-adic" numbers). It is also similar in purpose to [8] where a binary tree representation enables arithmetic operations which are simpler but limited in efficiency to to a small set of "sparse" numbers.

In contrast to [14], we are using here the familiar binary number system, and we represent our numbers as the *free algebra* of ordered rooted multiway threes, rather than the more complex data structure used in [14]. The focus on alternative free algebras as implementation of natural numbers is explored in detail in [1].

Several notations for very large numbers have been invented in the past. Examples include Knuth's *arrow-up* notation [9] 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 [11], 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.

Like our trees of type $\mathbb{T}$, Conway's surreal numbers [10] 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.

Arithmetic computations based on recursive data types like the free magma of binary trees (isomorphic to the context-free language of balanced parentheses) are described in [8], where they are seen as Gödel's `System T` types. In [5] a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite functions.

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [16]. 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 [17] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types.

VIII. 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 correctness and complexity estimates 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 shown that *computations* like addition, subtraction, multiplication, bitsize, exponent of 2, that favor giant numbers with *low structural complexity*, are performed in constant time, or time proportional to their structural complexity. We have also studied the best and worse case structural complexity of our representations and shown that, as structural complexity is bounded by bitsize, computations and data representations are within constant factors of conventional arithmetic even in the worse case.

The fundamental theoretical challenge raised at this point is the following: *can other number-theoretically interesting operations expressed succinctly in terms of our tree-based data type? Is it possible to reduce the complexity of some other important operations, besides those found so far?* In particular, is it possible to devise comparably efficient division and modular arithmetic operations favoring giant low structural complexity numbers? Would that have an impact on primality and factoring algorithms?

Another aspect of future work is 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. Easier developments with practicality in mind would involve extensions to signed integers and rational numbers.

Acknowledgement

This research has been supported by NSF research grant 1018172.

References

[1] P. Tarau, "Computing with Free Algebras," in *Proceedings of SYNASC 2012*, A. V. et al, Ed. IEEE, 2013, pp. 15–22, invited talk.

[2] The Univalent Foundations Program, *Homotopy Type Theory*. Institute of Advanced Studies, Princeton, 2013. [Online]. Available: http://homotopytypetheory.org/2013/06/20/the-hott-book/

[3] P. Tarau, ""Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms," *Complex Systems*, no. 18, pp. 475–493, 2010.

[4] ——, "An Embedded Declarative Data Transformation Language," in *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*. Coimbra, Portugal: ACM, Sep. 2009, pp. 171–182.

[5] ——, "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.

[6] ——, "Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell," Jan. 2009, unpublished draft, http://arXiv.org/abs/0808.2953, updated version at http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf, 150 pages.

[7] ——, "A Unified Formal Description of Arithmetic and Set Theoretical Data Types," in *Intelligent Computer Mathematics, 17th Symposium, Calculemus 2010, 9th International Conference AISC/Calculemus/MKM 2010*, S. Auxtier, Ed. Paris: Springer, LNAI 6167, Jul. 2010, pp. 247–261.

[8] 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.

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

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

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

[12] S. Corteel, B. Pittel, C. D. Savage, and H. S. Wilf, "On the multiplicity of parts in a random partition," *Random Struct. Algorithms*, vol. 14, no. 2, pp. 185–197, 1999.

[13] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*. New York, NY, USA: Springer-Verlag New York, Inc., 1993.

[14] P. Tarau, "Arithmetic Algorithms for Hereditarily Binary Natural Numbers," Jun. 2013. [Online]. Available: http://arxiv.org/abs/1306.1128
[15] A. Salomaa, *Formal Languages*. Academic Press, New York, 1973.
[16] 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.
[17] J. Vuillemin, "Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, june 2009, pp. 7 –14.

APPENDIX

A subset of Haskell as an executable function notation

We mention, for the benefit of the reader unfamiliar with Haskell, that a notation like `f x y` stands for $f(x,y)$, `[t]` represents sequences of type `t` and a type declaration like `f :: s -> t -> u` stands for a function $f : s\times t \to u$ (modulo Haskell's "currying" operation, given the isomorphism between the function spaces $s \times t \to u$ and $s \to t \to u$). Our Haskell functions are always represented as sets of recursive equations guided by pattern matching, conditional to constraints (simple relations following `|` and before the `=` symbol). Locally scoped helper functions are defined in Haskell after the `where` keyword, using the same equational style. The composition of functions `f` and `g` is denoted `f . g`. It is also customary in Haskell, when defining functions in an equational style (using =) to write $f = g$ instead of $f\ x = g\ x$ ("point-free" notation). We also make some use of Haskell's "call-by-need" evaluation that allows us to work with infinite sequences, like the `[0..]` infinite list notation, corresponding to the set of natural numbers $\mathbb{N}$. Note also that the result of the last evaluation is stored in the special Haskell variable `it`. By restricting ourselves to this *Haskell–* subset, our code can also be easily transliterated into a system of rewriting rules, other pattern-based functional languages as well as deterministic Horn Clauses.

Division operations

A fairly efficient integer division algorithm is given here, but it does not provide the same complexity gains as, for instance, multiplication, addition or subtraction. Finding a "one block at a time" division algorithm, if possible at all, is subject of future work.

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

  divstep n m = (q, sub n p) where
    q = try_to_double n m (F [])
    p = leftshiftBy q m

  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)
```