Scientific Annals of Computer Science vol. xx, yyyy, pp. 1–37

Arithmetic and Boolean Operations on Recursively Run-length Compressed Natural Numbers

Paul TARAU[1]

Abstract

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

We design arithmetic and boolean operations with recursively run-length compressed natural numbers that work a block of digits at a time and are limited only by the representation complexity of their operands, rather than their bitsizes.

As a result, operations on very large numbers exhibiting a regular structure become tractable.

In addition, we ensure that the average complexity of our operations is still within constant factors of the usual arithmetic operations on binary numbers.

Arithmetic operations on our recursively run-length compressed are specified as pattern-directed recursive equations made executable by using a purely declarative subset of the functional language Haskell.

Keywords: run-length compressed numbers, hereditary numbering systems, arithmetic algorithms for giant numbers, representation complexity of natural numbers.

[1] Department of Computer Science and Engineering, University of North Texas, Denton, 76203 TX, USA. Email: `tarau@cse.unt.edu`

1 Introduction

Notations like Knuth's "up-arrow" [6] have been shown to be useful in describing very large numbers. However, 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.

The main focus of this paper is a new tree-based numbering system that allows computations with numbers comparable in size with Knuth's "up-arrow" notation. Moreover, these computations have worst and average 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.

For the curious reader, it is basically a *hereditary number system* [3], based on recursively applied run-length compression of the usual binary digit notation. It favors giant numbers in neighborhoods of towers of exponents of two, with super-exponential gains on their arithmetic operations. Moreover, the proposed notation is *canonical* i.e., each number has a unique representation (contrary to the traditional one where any number of leading zeros can be added).

We adopt a literate programming style, i.e. the code described in the paper forms a self-contained Haskell module (tested with ghc 7.6.3), also available as a separate file at http://www.cse.unt.edu/~tarau/research/2014/RCN.hs .

Our literate Haskell program is organized as the module RCN with the declaration:

```
module RCN where
import System.Random
```

The code in this paper can be seen as a compact and mathematically obvious specification rather than an implementation fine-tuned for performance. Faster but more verbose equivalent code can be derived in procedural or object oriented languages by replacing lists with (dynamic) arrays and some instances of recursion with iteration.

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 elements of type `t` and a type declaration like `f :: s -> t -> u` stands for a function $f : s \times t \to u$.

Our Haskell functions are always represented as sequences 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 $f.g$. Note also that the result of the last evaluation is stored in the special Haskell variable `it`.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces our tree-represented recursively run-length compressed natural numbers. Section 4 describes constant time successor and predecessor operations on tree-represented numbers. Section 5 describes novel algorithms for arithmetic operations taking advantage of our number representation. Section 6 defines several specialized operations and primality tests. Section 7 introduces bitwise operations taking advantage of our representation and applies them to boolean evaluation. Section 8 defines a concept of representation complexity and studies best and worst cases. Section 9 describes an example of computation with very large numbers using recursively run-length compressed numbers. Section 10 concludes the paper.

This is an extended and improved version of the paper [18] with most of the new material concentrated in sections 6 and 7.

2 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.

The first instance of a hereditary number system, at our best knowledge, occurs in the proof of Goodstein's theorem [3], 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 [2] can also be seen as inductively constructed trees. While our focus will be on efficient large natural number arithmetic, surreal numbers model games, transfinite ordinals and generalizations of real numbers.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [6], covering operations like the tetration (a notation for towers of exponents). In contrast to the tree-based natural numbers we propose in this paper, such notations are not closed under addition and multiplication, and consequently they cannot be used as a replacement for ordinary binary or decimal numbers.

This paper is similar in purpose with [19] which describes a more complex hereditary number system (based on run-length encoded "bijective base 2" numbers, introduced in [14] pp. 90-92 as "m-adic" numbers). In contrast to [19], we are using here the familiar binary number system, and we represent our numbers as the free algebra of ordered rooted multiway trees, rather than the more complex data structure used in [19].

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

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [5]. 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 [20] a binary tree representation enables arithmetic operations which are simpler but limited in efficiency to a small set of "sparse" numbers.

In [22] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. However likewise [20] and [16], and in contrast to those proposed in this paper, they only compress "sparse" numbers, consisting of relatively few `1` bits in their binary representation.

The tree representation that we will use is an instance of the Catalan family of combinatorial objects [15], on which, in [17], arithmetic operations are seen as operating on balanced parenthesis languages. While combinatorial enumeration and combinatorial generation, for which a vast literature exists (see for instance [15], [4], [10], [8], [12] and [13]), can be seen as providing unary Peano arithmetic operations implicitly, we are not aware of any work enabling arithmetic computations of efficiency comparable to the usual binary numbers (or better) using an instance of a combinatorial family. In fact, this is the main motivation and the most significant contribution of this paper.

3 The Data Type of Recursively Run-length Compressed Natural Numbers

First, we define a data type for our tree-represented natural numbers, that we call *recursively run-length compressed numbers* to emphasize that this encoding is recursively used in their representation. Through the paper, we assume a "big-endian" notation, with the least significant digit first for binary strings.

Definition 1 *The data type* `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* `T` *corresponds precisely to ordered rooted multiway trees with empty leaves, but for shortness, we will call the objects of type* `T` terms*. The "arithmetic intuition" behind the type* `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* $b \in \{0,1\}$ *digits, followed by* alternating *counts of* `1-b` *and* `b` *digits, with the convention that the most significant digit is* 1
- *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. Note that we count `x+1` as we start at `0`. By convention, as the last (and most significant) digit is `1`, the last count on the list `xs` is for `1` digits. For instance, the first level of the encoding of `123` as the (big-endian) binary number `1101111` is `[1,0,3]`.

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_`.

```
odd_ ::  T → Bool
odd_ (F []) = False
odd_ (F (_:xs)) = even_ (F xs)

even_ :: T → Bool
even_ (F []) = True
even_ (F (_:xs)) = odd_ (F 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 that keeps track of the length or keeps track of the parity bit explicitly, one can assume that they are constant time, as we will do in the rest of the paper.

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

$$n(a)=\begin{cases}0 & \textit{if } a= \ \mathtt{F\ []},\\ 2^{n(\mathtt{x})+1}n(\mathtt{F\ xs}) & \textit{if } a= \ \mathtt{F\ (x:xs)}\ \textit{is}\ \mathtt{even\_},\\ 2^{n(\mathtt{x})+1}(1+n(\mathtt{F\ xs}))-1 & \textit{if } a= \ \mathtt{F\ (x:xs)}\ \textit{is}\ \mathtt{odd\_}.\end{cases} \tag{1}$$

For instance, the computation of `n(F [F [],F [F [],F []]])` using equation (1) expands to $(2^{0+1}(2^{(2^{0+1}(2^{0+1}-1))+1}-1))=14$, which, in binary, is `[0,1,1,1]` where the first level expansion `[0,2]`, corresponds to `F []` → `0` and `F [F [],F []]` → `2`. After defining the type of natural numbers as

```
type N = Integer
```

the Haskell equivalent[2] of equation (1) is:

```
n :: T → N
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 []]]
```

[2]As a Haskell note, the pattern `a@p` indicates that the parameter `a` has the same value as its expanded version matching the patten `p`.

One might notice that our trees are in bijection with objects of the *Catalan family*, e.g., strings of balanced parentheses, for instance $0 \rightarrow$ `F []` $\rightarrow ()$, $1 \rightarrow$ `F [F []]` $\rightarrow (())$, $14 \rightarrow$ `F [F [],F [F [],F []]]` $\rightarrow (()(()()))$.

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

```
t :: N → T
t 0 = F []
t k | k>0 = F zs where
  (x,y) = split_on (parity k) k
  F ys = t y
  zs = if x==0 then ys else t (x-1) : ys

  parity x = x `mod` 2

  split_on b z | z>0 && parity z == b = (1+x,y) where
    (x,y) = split_on b  ((z-b) `div` 2)
  split_on _ z = (0,z)
```

It uses the helper function `split_on`, which, depending on parity `b`, extracts a block of contiguous `0` or `1` digits from the lower end of a binary number. It returns a pair `(x,y)` consisting of a count `x` of the number of digits in the block, and the natural number `y` representing the digits left over after extracting the block. Note that `div`, occurring in both functions, is integer division.

The following holds:

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

$$t \circ n = id, \quad n \circ t = id \ . \tag{2}$$

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

The following example illustrates the correctness of our definitions:

```
*RCN> map (n.t) [0..15]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

4 Successor (s) and Predecessor (s')

We will now specify successor and predecessor on data type `T` through two mutually recursive functions, `s` and `s'`.

```
s :: T → T
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
```

```
s' :: T → T
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 *on a block of* `0` *or* `1` *digits at a time*. They 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* $\mathtt{T}^+ = \mathtt{T} - \{\mathtt{F\ []}\}$. *The functions* $s : \mathtt{T} \to \mathtt{T}^+$ *and* $s' : \mathtt{T}^+ \to \mathtt{T}$ *are inverses.*

Proof: It follows by structural induction after observing that patterns for rules 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 $< \mathtt{T}, \mathtt{F}[], \mathtt{s} >$ is a Peano algebra.

Note also that if parity information is kept explicitly, the calls to `odd_` and `even_` are constant time, as we will assume in the rest of the paper.

The following examples illustrate the correctness of our definitions of `s` and `s'`:

```
*RCN> map (n.s'.s.t) [0..15]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Proposition 4 *The worst case time complexity of the* `s` *and* `s'` *operations on n is given by the iterated logarithm $O(log_2^*(n))$, where log_2^* counts the number of times log_2 can be applied before reaching 0.*

Proof: Note that calls to `s` and `s'` in `s` or `s'` happen on terms at most logarithmic in the bitsize of their operands. The recurrence relation counting the worst case number of calls to `s` or `s'` is: $T(n) = T(log_2(n)) + O(1)$, which solves to $T(n) = O(log_2^*(n))$. □

Note that this is much better than the logarithmic worst case for binary umbers (when computing, for instance, binary 111...111+1=1000...000).

Proposition 5 `s` *and* `s'` *are constant time, on the average.*

Proof: Observe that the average size of a contiguous block of `0`s or `1`s in a number of bitsize n has the upper bound 2 as $\sum_{k=0}^{n} \frac{1}{2^k} = 2 - \frac{1}{2^n} < 2$. As on 2-bit numbers we have an average of 0.25 more calls, we can conclude that the total average number of calls is constant, with upper bound $2 + 0.25 = 2.25$. □

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

5 Arithmetic Operations

Clearly one could use the successor and predecessor operations `s` and `s'` to implement unary Peano arithmetic. However, that would be exponentially less efficient than the usual binary arithmetic.

The interesting thing about our representation and our successor/predecessor definitions is that we can do much better.

We will now describe algorithms for basic arithmetic operations that take advantage of our number representation and provide equivalent or better performance than the usual binary numbers.

5.1 A few Other Average Constant Time 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 :: T → T
db (F []) = F []
db a@(F xs) | odd_ a = F (F []:xs)
db a@(F (x:xs)) | even_ a = F (s x:xs)
```

```
hf :: T → T
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 :: T → T
exp2 (F []) = F [F []]
exp2 x = F [s' x,F []]
```

As `log2` shows, `exp2` is also easy to invert with a similar amount of work:

```
log2 :: T → T
log2 (F [F []]) = F []
log2 (F [y,F []]) = s y
```

Note that this definition works on powers of 2, see Subsection 5.4 for a general version.

The following examples illustrate these operations:

```
*RCN> map (n.db.t) [0..15]
[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
*RCN> map (n.hf.db.t) [0..15]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
*RCN> map (n.exp2.t) [0..15]
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768]
*RCN> map (n.log2.exp2.t) [0..15]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Proposition 6 *The operations* `db`*,* `hf`*,* `exp2` *and* `log2` *are average constant-time and iterated logarithm in the worst case.*

Proof: At most 1 call to `s` or `s'` is made in each definition. Therefore these operations have the same worst and average complexity as `s` and `s'`. □

5.2 Optimizing Addition and Subtraction for Numbers with Few Large Blocks of 0s and 1s

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

```
leftshiftBy :: T → T → T
leftshiftBy (F []) k = k
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' :: T → T → T
leftshiftBy' x k = s' (leftshiftBy x (s k))

leftshiftBy'' :: T → T → T
leftshiftBy'' x k = s' (s' (leftshiftBy x (s (s k))))
```

The last two are derived from 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 1s, leftshiftBy' just increments that count. As a result, the algorithm favors numbers with relatively few large blocks of 0 and 1 digits.

The following examples illustrate these operations:

```
*RCN> n (leftshiftBy (t 5) (t 3))
96
*RCN> n (leftshiftBy' (t 5) (t 3))
127
*RCN> n (leftshiftBy'' (t 5) (t 3))
158
```

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

```
add :: T → T → T
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):

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

```
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 :: T → T → T
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 a `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)
```

Note that these algorithms collapse to the ordinary binary addition and subtraction most of the time, given that the average size of a block of contiguous `0`s or `1`s is 2 bits (as shown in Prop. 5), so their average performance is within a constant factor of their ordinary counterparts. On the other hand, the algorithms favor deeper trees made of large blocks, representing giant “towers of exponents”-like numbers by working (recursively) one block at a time rather than 1 bit at a time, resulting in possibly super-exponential gains.

5.3 Comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our type `T`. It relies on `bitsize` computing the number of binary digits constructing a term in `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 7 *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 operation also proceeds one block at a time, and it also takes some inferential shortcuts, when possible.

```
cmp :: T → T → Ordering
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 8 *Assuming two terms of the same bitsizes, the one with its first before its highest order digit* 1 *is larger than the one with its first before its highest order 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 lead the sequence of blocks, otherwise it infers the `LT` or `GT` relation.

The function `compBigFirst` is driven by two boolean arguments encoding the parity of the alternating blocks, initially set to `True`, as the highest order block is always made of 1s.

```
compBigFirst :: Bool → Bool → T → T → Ordering
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
```

Note that when parities are distinct, `False` and `True` results in `LT` indicating that the first term (headed by 0s) is smaller than the second. Conversely, `True` and `False` results in `GT` indicating that the first term (headed by 1s) is greater than the second. The following examples illustrate the comparison operation `cmp`:

```
*RCN> cmp (t 100) (t 200)
LT
*RCN> cmp (t 300) (t 200)
GT
*RCN> cmp (t 200) (t 200)
EQ
```

5.4 Bitsize

The function `bitsize` computes the number of digits, except that we define it as `F []` for `F []`, corresponding to `0`. It concludes the chain of mutually recursive functions starting with the addition operation `add`. It works by summing up (using Haskell’s `foldr`) the counts of `0` and `1` digit blocks composing a tree-represented natural number.

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

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

It follows that the base-2 integer logarithm is then computed as

```
ilog2 :: T → T
ilog2 = s' . bitsize
```

The iterated logarithm log_2^* can be also defined as

```
ilog2star :: T → T
ilog2star (F []) = F []
ilog2star x = s (ilog2star (ilog2 x))
```

The following examples illustrate these operations:

```
*RCN> map (n.ilog2.t) [1..15]
[0,1,1,2,2,2,2,3,3,3,3,3,3,3,3]
*RCN> (n.bitsize.exp2.exp2.exp2.exp2.t) 2
65537
*RCN> (n.ilog2star.exp2.exp2.exp2.exp2.t) 2
6
```

5.5 Multiplication, Optimized for Large Blocks of `0`s and `1`s

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 `leftshiftBy` operation and we reduce the other case to this one.

```
mul :: T → T → T
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 its first argument rather than the the number of digits. The following example illustrates a blend of arithmetic operations benefiting from complexity reductions on giant tree-represented numbers:

```
*RCN> let term1 = sub (exp2 (exp2 (t 12345))) (exp2 (t 6789))
*RCN> let term2 = add (exp2 (exp2 (t 123))) (exp2 (t 456789))
*RCN> bitsize (bitsize (mul term1 term2))
F [F [],F [],F [],F [F [],F []],F [F [],F [],F []],F [F []]]
*RCN> 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 representation complexity. We will make this concept more precise in section 8.

5.6 Power

After specializing our multiplication for a squaring operation,

```
square :: T → T
square x = mul x x
```

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

```
pow :: T → T → T
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:

```
*RCN> n (bitsize (pow (t 10) (t 100)))
333
*RCN> 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 []]
```

5.7 Division Operations

We start by defining an efficient special case.

5.7.1 A Special Case: Division by a Power of 2

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

```
rightshiftBy :: T → T → T
rightshiftBy (F [])  y = y
rightshiftBy _ (F []) = F []
rightshiftBy x (F (a:xs)) = f (cmp x a')  where
  b = F xs
  a' = s a
  f LT = F (sub a x:xs)
  f EQ = b
  f GT = rightshiftBy (sub  x a') b
```

5.7.2 General division

An integer division algorithm is given here, but it does not provide the same complexity gains as, for instance, multiplication, addition or subtraction.

```
div_and_rem :: T → T → (T, T)
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
```

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

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

Division and remainder are obtained by specializing `div_and_rem`.

```
divide :: T → T → T
divide n m = fst (div_and_rem n m)

remainder :: T → T → T
remainder n m = snd (div_and_rem n m)
```

The following examples illustrate these operations:

```
*RCN> n (rightshiftBy (t 3) (t 50))
6
*RCN> n (divide (t 100) (t 9))
11
*RCN> n (remainder (t 100) (t 9))
1
```

6 Specialized Arithmetic Operations and Primality Tests

We describe in this section a number of special purpose arithmetic operations showing the practical usefulness of our number representation.

6.1 Integer Square Root

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

```
isqrt :: T → T
isqrt (F []) = F []
isqrt n = if cmp (square k) n==GT then s' k else k where
  two = F [F [],F []]
  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
```

6.2 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 :: T → T → T → T
modPow m base expo = modStep expo (F [F []]) base where
  modStep(F [F []]) r b  = (mul r b) `remainder` m
  modStep x r b | odd_ x =
    modStep (hf (s' x)) (remainder (mul r b) m)
                        (remainder (square b)  m)
  modStep x r b = modStep (hf x) r (remainder (square b) m)
```

The following examples illustrate the correctness of these operations:

```
*RCN> n (isqrt (t 103))
10
*RCN> modPow (t 10) (t 3) (t 4)
F [F []]
*RCN> n it
1
```

6.3 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. 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 [1]. The function `ll_iter` implements this iteration.

```
ll_iter :: T → T → T → T
ll_iter (F []) n m = 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 :: T → T → T
fastmod k m | k == s' m = F []
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 primality test is implemented as follows:

```
lucas_lehmer :: T → Bool
lucas_lehmer p | p == s (s (F [])) = True
lucas_lehmer p = F [] ==  (ll_iter p_2 four m) where
```

```
  p_2 = s' (s' p)
  four = F [F [F []],F []]
  m  = exp2 p
```

We illustrate its use for detecting a few Mersenne primes:

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

Note that the last line contains the Mersenne primes corresponding to $2p+1$.

6.4 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 (6)

$$dyadicSplit(k) = (k, \frac{k}{2^{\nu_2(k)}}) \tag{6}$$

can be implemented as an average constant time operation as:

```
dyadicSplit :: T → (T, T)
dyadicSplit z | odd_ z = (F [],z)
dyadicSplit z | even_ z = (s x, s (g xs)) where
  F (x:xs) = s' z

  g [] =  F []
  g (y:ys) = F (y:ys)
```

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

```
randomNats :: Int → Int → T → T → [T]
randomNats seed k smallest largest = map t ns  where
  ns :: [N]
  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 :: Int → T → Bool
oddPrime k m = all strongLiar as where
  m' = s' m
  as = randomNats k k (F [F [],F []]) m'
  (l,d) = dyadicSplit m'

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

    squaringSteps (F []) _ = []
    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 :: T → Bool
isProbablyPrime (F [F [],F []])  = True
isProbablyPrime x | even_ x = False
isProbablyPrime p = oddPrime 42 p
```

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

```
*RCN>  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]
```

7 Boolean Operations on Tree-represented Bitvectors

We implement bitvector operations (also seen as efficient bitset operations) to work "one block of binary digits at a time", to facilitate their use on large but sparse boolean formulas involving a large numbers of variables. One will be able to evaluate such formulas "all value-combinations at a time" when represented as bitvectors of size 2^{2^n}. Note that such operations will be tractable with our trees, provided that they have a relatively small representation complexity, despite their large bitsize.

7.1 Bitwise Operations One Block of Digits at a time

We implement a generic `bitwise` operation that takes a boolean function `bf` as its first parameter.

First, when an argument is `F []`, corresponding to `0` the behavior is derived from that of the boolean function `bf`.

```
bitwise :: (Bool → Bool → Bool) → T → T → T
bitwise bf (F []) (F []) = F []
bitwise bf (F []) y = if bf False True then y else F []
bitwise bf x (F []) = if bf True False then x else F []
```

Next, the parities of the arguments `px` and `py` are used to derive the parity of the result `pz`, by applying the boolean function `bz`.

```
bitwise bf x@(F (a:as)) y@(F (b:bs))  = f (cmp a b) where
  px = odd_ x
  py = odd_ y
  pz = bf px py
```

Based on the parity `pz` the local function `f` (also parameterized by the result of the comparison between arguments `x` and `y`) is called.

```
  f EQ = fApply bf pz (s a) (F as) (F bs)
  f GT = fApply bf pz (s b) (fromB px (sub a b) (F as)) (F bs)
  f LT = fApply bf pz (s a) (F as) (fromB py (sub b a) (F bs))
```

The function `f` calls `fromB` to derive from the parities `px` and `py` the appropriate left-shifting operation.

```
  fromB False = leftshiftBy
  fromB True  = leftshiftBy'
```

Finally, the function `f` calls the helper function `fApply`, which, depending on the expected parity of the result `pz`, applies the appropriate left-shift operation to the result of the recursive application of `bitwise` to the remaining blocks of digits `u` and tt v.

```
  fApply bf False k u v =  leftshiftBy k (bitwise bf u v)
  fApply bf True k u v =  leftshiftBy' k (bitwise bf u v)
```

The actual bitwise operations are obtained by parameterizing the generic `bitwise` function with the appropriate Haskell boolean functions:

```
bitwiseOr :: T → T → T
bitwiseOr = bitwise (||)

bitwiseXor :: T → T → T
bitwiseXor = bitwise (/=)

bitwiseAnd :: T → T → T
bitwiseAnd = bitwise (&&)
```

Bitwise negation (requiring the additional parameter `k` to specify the intended bitlength of the operand) corresponds to the complement w.r.t. the "universal set" of all natural numbers up to $2^k - 1$. It is defined as usual, by subtracting from the "bitmask" corresponding to $2^k - 1$:

```
bitwiseNot :: T → T → T
bitwiseNot k x = sub y x where y = s' (exp2 k)
```

The function `bitwiseAndNot` combines `bitwiseOr`, `bitwiseNot` the usual way, except that it uses the helper function `bitsOf` to ensure enough mask bits are made available when negation is applied.

```
bitwiseAndNot :: T → T → T
bitwiseAndNot x y = bitwiseNot l d  where
  l = max2 (bitsOf x) (bitsOf y)
  d = bitwiseOr (bitwiseNot l x) y
```

The function `max2` is defined in terms of comparison operation `cmp` as follows:

```
max2 :: T → T → T
max2 x y = if LT==cmp x y then y else x
```

The function `bitsOf` adapts our definition for `bitsize` to compute the number of bits of a bitvector (considering 0 to be 1 bit).

```
bitsOf :: T → T
bitsOf (F []) = s (F [])
bitsOf x = bitsize x
```

The following example illustrates that our bitwise operations can be efficiently applied to giant numbers:

```
*RCNx> bitwiseXor (s (exp2 (exp2 (t 12345))))
                  (s' (exp2 (exp2 (t 6789))))
F [F [],F [F [],F [F [F []],F [F [F []],F []],F [],F [],F [],F [],
  F [F []]]],F [F [F [F []],F [],F [F [F []]],F [],F [],F [],F [],
  F [F []]],F [],F [F [],F [],F [F []],F [F []],F [],F [F []],F [],
  F [],F [],F []]],F []]
```

Note that while the size of the term representing this result is `46 F` nodes the bitsize of the result is 2^{12346} showing clearly that such an operation is intractable with a bitstring representation.

7.2 Boolean Formula Evaluation

Besides definitions for the bitwise boolean functions, we also need definitions of the projection variables $var(n,k)$ corresponding to column k of a truth table, for a function with n variables. A compact formula for them, as given in [9] or [21], is

$$var(n,k) = (2^{2^n} - 1) \ / \ (2^{2^{n-k-1}} + 1) \tag{7}$$

However, instead of doing the division, one can compute them as a concatenation of alternating blocks of 1 and 0 bits to take advantage of our efficient block operations.

```
var :: T → T → T
var n k = repeatBlocks nbBlocks blockSize mask where
  k' = s k
  nbBlocks = exp2 k'
  blockSize = exp2 (sub n k')
  mask = s' (exp2 blockSize)
```

The alternating blocks are put together by the function `repeatBlocks` that shifts to the left by the size of a block, at each step, and adds the `mask` made of 2^{n-k} ones, at each even step.

```
  repeatBlocks (F []) _ _ = F []
  repeatBlocks k l mask =
   if odd_ k then r else add mask r where
    r = leftshiftBy l (repeatBlocks (s' k) l mask)
```

The following examples illustrate these operations:

```
*RCN> map n (map (var (t 3)) (map t [0..2]))
[15,51,85]
*RCN> map n (map (var (t 4)) (map t [0..3]))
[255,3855,13107,21845]
*RCN> map n (map (var (t 5)) (map t [0..4]))
[65535,16711935,252645135,858993459,1431655765]
```

The following example illustrates the evaluation of a boolean formula in conjunctive normal form (CNF). The mechanism is usable as a simple satisfiability or tautology tester, for formulas resulting in possibly large but sparse or dense, low structural complexity bitvectors.

```
cnf :: T
cnf = andN (map orN cls) where
```

```
cls = [[v0',v1',v2],[v0,v1',v2],
       [v0',v1,v2'],[v0',v1',v2'],[v0,v1,v2]]

v0 = var (t 3) (t 0)
v1 = var (t 3) (t 1)
v2 = var (t 3) (t 2)

v0' = bitwiseNot (exp2 (t 3)) v0
v1' = bitwiseNot (exp2 (t 3)) v1
v2' = bitwiseNot (exp2 (t 3)) v2

orN (x:xs) = foldr bitwiseOr x xs
andN (x:xs) = foldr bitwiseAnd x xs
```

The execution of the function `cnf` evaluates the formula, the result corresponding to bitvector `88 = [0,0,0,1,1,0,1,0]`.

```
*RCN> cnf
F [F [F [],F []],F [F []],F [],F []]
*RCN> n it
88
```

8 Representation Complexity

While a detailed analysis of all our algorithms is beyond the scope of this paper, arguments similar to those about the average behavior of `s` and `s'` can be carried out to prove that *their average complexity matches their traditional counterparts*, using the fact, shown in the proof of Prop. 5, that the average size of a block of contiguous `0` or `1` bits is at most `2`.

8.1 Complexity as Representation Size

To evaluate the best and worst case space requirements of our number representation, we introduce here a measure of *representation complexity*, defined by the function `tsize` that counts the nodes of a tree of type `T` (except the root).

```
tsize :: T → T
tsize (F xs) = foldr add1 (F []) (map tsize xs)
```

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

$$c(t) = \begin{cases} 0 & \text{if } \texttt{t} = \text{ F []}, \\ \sum_{x \in \texttt{xs}} (1 + c(x)) & \text{if t} = \texttt{F xs}. \end{cases} \tag{8}$$

The following holds:

Proposition 9 *For all terms $t \in$ 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 inductively assumed smaller than those of `bitsize`. □

The following example illustrates their use:

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

8.2 Best and Worst Cases

Next we define the higher order function `iterated` that applies `k` times the function `f`.

```
iterated :: (T → T) → T → T → T
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 :: T → T
bestCase k = iterated wTree k (F []) where wTree x = F [x]
```

and a worst case

```
worstCase :: T → T
worstCase k = iterated f k (F []) where f (F xs) = F (F []:xs)
```

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

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

Given the time complexity of `s` (Props. 4 and 5) and the k applications of function `iterated`, the terms `bestCase k` and `worstCase k` are built in average time proportional to k and worst case time $O(k * log^{*}k)$.

The following examples illustrate these functions:

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

*RCN> worstCase (t 4)
F [F [],F [],F [],F []]
*RCN> n it
10
*RCN> n (bitsize (worstCase (t 4)))
4
*RCN> n (tsize (worstCase (t 4)))
4
```

Note that the worst case corresponds to alternation of 0 and 1 bits while the best case corresponds to a tower of exponents of 2 minus 1 resulting in a large block of 1s that are recursively described the same way.

Our concept of representation complexity is only a weak approximation of Kolmogorov complexity [11]. Kolmogorov complexity is given by the size of the smallest program that computes a given bitstring. As such, it is uncomputable but it is often approximated by incompressibility - a property that can also be used to test the quality of randomly generated bitstrings. For instance, the reader might notice that our worst 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 representation complexity can be defined by considering the (vertices+edges) size of the DAG obtained by folding together identical subtrees.

8.3 A Concept of Duality

We will discuss here a concept of “duality” that connects our worst and best cases. We will define it through a simple transformation between “shallow” and “deep” trees representing our numbers.

Looking back at the worst and best cases for $n = 4$, `t 10 = F [F [],F [],F [],F []]` and `t 65535 = F [F [F [F [F []]]]]`, note that the shallow (1-level) tree corresponding to `10` shares the same tree size with the deep (4-level) tree corresponding to `65535`. We will generalize this observation by defining a tree transformation that puts such trees into a bijection.

As our multiway trees with empty leaves are members of the Catalan family of combinatorial objects, they can be seen as binary trees with empty leaves, as defined by the bijection `toBinView` and its inverse `fromBinView`.

```
toBinView :: T → (T, T)
toBinView (F (x:xs)) = (x,F xs)

fromBinView :: (T, T) → T
fromBinView (x,F xs) = F (x:xs)
```

Therefore, we can transform the tree-representation of a natural number by swapping left and right branches under a binary tree view, recursively. The corresponding Haskell code is:

```
dual :: T → T
dual (F []) = F []
dual x = fromBinView (dual b, dual a) where (a,b) = toBinView x
```

As clearly `dual` is an *involution* (i.e., `dual` ∘ `dual` is the identity of `T`), the corresponding permutation of $\mathbb{N}$ will put in bijection huge and small natural numbers sharing representations of the same size, as illustrated by the following example.

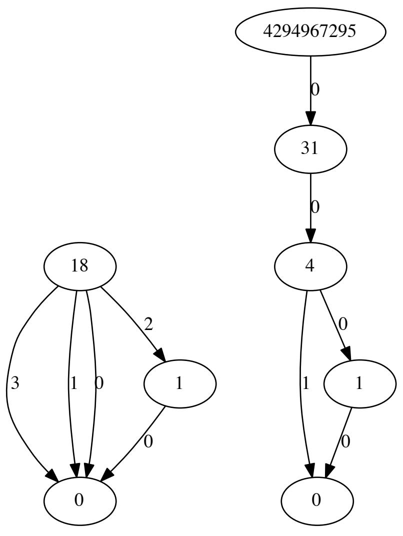
```
*RCN> map (n.dual.t) [0..20]
[0,1,3,2,4,15,7,6,12,31,65535,16,8,255,127,5,11,8191,
                                              4294967295,32,65536]
```

For instance, `18` and `4294967295` have dual representations of the same size, except that the wide tree associated to `18` maps to the tall tree associated to `4294967295`, as illustrated by Fig. 1, with trees folded to DAGs by merging together shared subtrees. As a result, significantly different bitsizes can result for a term and its dual.

```
*RCN> t 18
F [F [],F [],F [F []],F []]
*RCN> dual (t 18)
F [F [F [F [F []],F []]]]
*RCN> n (bitsize (t 18))
5
*RCN> n (bitsize (dual (t 18)))
32
```

It follows immediately from the definitions of the respective functions, that as an extreme case, the following holds:

Proposition 10 $\forall$ `x dual (bestCase x)` = `worstCase x`.



(a) t 18 (b) dual (t 18)

Figure 1: Duals, with trees folded to DAGs, with numbers on edges indicating their order

The following example illustrates this equality, with a tower of exponents `1000` tall, reached by `bestCase`.

```
*RCN> dual (bestCase (t 10000)) == worstCase (t 10000)
True
```

Note that these computations run easily on objects of type `T`, while their equivalents would dramatically overflow memory on bitstring-represented numbers.

Another interesting property of `dual` is illustrated by the following examples:

```
*RCN> [x|x<-[0..2^5-1],cmp (t x) (dual (t x)) ==  LT]
[2,5,6,8,9,10,11,13,14,17,18,19,20,21,22,23,25,26,27,28,29,30]
*RCN> [x|x<-[0..2^5-1],cmp (t x) (dual (t x)) ==  EQ]
[0,1,4,24]
```

```
*RCN> [x|x<-[0..2^5-1],cmp (t x) (dual (t x)) ==  GT]
[3,7,12,15,16,31]
```

The discrepancy between the number of elements for which `x` is smaller than `(dual x)` and those for which it is greater or equal, is growing as numbers get larger, contrary to the intuition that, as `dual` is an involution, the grater and smaller sets would have similar sizes for an initial interval of $\mathbb{N}$. For instance, between 0 and $2^{16}-1$ one will find only 68 numbers for which the dual is smaller and 11 for which it is equal.

Note that random elements of $\mathbb{N}$ tend to have relatively *shallow* (and wide) multiway tree representations, given that the average size of a contiguous block of `0`s or `1`s is `2`. Consequently, `dual` provides an interesting bijection between "incompressible" natural numbers (of high Kolmogorov complexity) and their *deep*, highly compressible, duals.

9 A Case Study: Computing the Collatz/Syracuse Sequence for Huge Numbers

An 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 representation complexity. The Collatz conjecture states that the function

$$collatz(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{9}$$

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

$$collatz'(x) = \begin{cases} \frac{x}{2^{\nu_2(x)}} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{10}$$

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 $n = 3k+1 = 2^{\nu_2(n)}k'$. 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

$$tl(k) = \frac{\frac{k}{2^{\nu_2(k)}} - 1}{2} \tag{11}$$

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

```
hd :: T → T
hd = fst . decons

tl :: T → T
tl = snd . decons

decons :: T → (T, T)
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 :: (T, T) → T
cons (x,y) = leftshiftBy x (s (db y))
```

corresponding to $2^x\ (2y+1)$. Then our variant of the *syracuse function* corresponds to

$$syracuse(n) = tl(3n+2) \tag{12}$$

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

```
syracuse :: T → T
syracuse n = tl (add n (db (s n)))
```

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

```
nsyr :: T → [T]
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:

```
*RCN> 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*:

```
*RCN> 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 representation complexity `tsize` of these numbers, that obviously would not fit themselves in a decimal representation.

10 Conclusion

We have provided in the form of a literate Haskell program a specification of a tree-based number system where trees are built by recursively applying run-length encoding on the usual binary representation until the empty leaves corresponding to `0` are reached.

The resulting numbering system, based on a bijection between natural numbers and trees, is *canonical* - each natural number is represented as a unique object. Besides unique decoding, such canonical representations ensure that equality testing reduces to *syntactic equality*.

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

We have shown that realistic computations (e.g.; advanced primality tests) can be performed with our numbers and that bitvector boolean operations can benefit from our representation when they contain large contiguous blocks of `0` and `1` digits.

The conditions for lower time and space complexity for algorithms working on numbers consisting of large contiguous blocks of `0`s and `1`s are also likely to apply to several practical data representations ranging from quad-trees to audio/video encoding formats.

Acknowledgement

This research has been supported by NSF grant 1423324. The author is grateful to the anonymous reviewers of ICTAC'14 and SACS for the thoughtful comments and suggestions that have improved the final version of this paper.

References

[1] J. W. Bruce. A Really Trivial Proof of the Lucas-Lehmer Test. *The American Mathematical Monthly*, 100(4):370–371, 1993.

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

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

[4] Norbert Hungerbühler. The isomorphism problem for catalan families. *J. Combin. Inform. System Sci*, 20:129–139, 1995.

[5] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *FLOPS*, pages 64–80, 2008.

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

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

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming).* Addison-Wesley Professional, 2006.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 2009.

[10] Donald L. Kreher and D.R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* The CRC Press Series on Discrete Mathematics and its Applications. CRC PressINC, 1999.

[11] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[12] J. Liebehenschel. Ranking and unranking of a generalized Dyck language and the application to the generation of random trees. *Séminaire Lotharingien de Combinatoire*, 43:19, 2000.

[13] Conrado Martinez and Xavier Molinero. Generic algorithms for the generation of combinatorial objects. In Branislav Rovan and Peter Vojtas, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 572–581, Berlin Heidelberg, 2003. Springer.

[14] Arto Salomaa. *Formal Languages*. Academic Press, New York, 1973.

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

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

[17] Paul Tarau. Computing with Catalan Families. In Adrian-Horia Dediu, Carlos Martin-Vide, José-Luis Sierra, and Bianca Truthe, editors, *Proceedings of Language and Automata Theory and Applications, 8th International Conference, LATA 2014*, pages 564–576, Madrid, Spain,, March 2014. Springer, LNCS.

[18] Paul Tarau. The Arithmetic of Recursively Run-length Compressed Natural Numbers. In Gabriel Ciobanu and Doninique Méry, editors, *Proceedings of 8th International Colloquium on Theoretical Aspects of Computing, ICTAC 2014*, pages 406–423, Bucharest, Romania, September 2014. Springer, LNCS 8687.

[19] Paul Tarau and Bill Buckles. Arithmetic Algorithms for Hereditarily Binary Natural Numbers. In *Proceedings of SAC'14, ACM Symposium on Applied Computing, PL track*, pages 1593–1600, Gyeongju, Korea, March 2014. ACM.

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

[21] Paul Tarau and Brenda Luderman. Boolean Evaluation with a Pairing and Unpairing Function. In Andrei Voronkov, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen Watt, and Daniela Zaharie, editors, *Proceedings of SYNASC 2012*, pages 384–392. IEEE, January 2013.

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