A Generic Numbering System based on Catalan Families of Combinatorial Objects

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

Abstract

We study novel arithmetic algorithms on a canonical number representation based on the Catalan family of combinatorial objects.

Our algorithms work on a generic representation that we illustrate on instances like ordered binary and multiway trees, balanced parentheses languages as well as the usual bitstring-based natural numbers seen through the same generic interface as members of the Catalan family.

For numbers corresponding to Catalan objects of low representation complexity, our algorithms provide super-exponential gains while their average and worst case complexity is within constant factors of their traditional counterparts.

Keywords: *hereditary numbering systems, run-length compressed numbers, arithmetic with combinatorial objects, Catalan families, representation complexity of natural numbers, generic functional algorithms*

1. Introduction

This paper is an extended and generalized version of [1], where a special instance of the Catalan family of combinatorial objects, the language of balanced parentheses, has been endowed with basic arithmetic operations corresponding to those on bitstring-represented natural numbers.

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 alternative *notations* like Knuth's "up-arrow" [2] are useful in describing very large numbers, they do not provide the ability to actually *compute* with them, as addition or multiplication with a natural number results in a number that cannot be expressed with the notation anymore.

Email address: tarau@cse.unt.edu (Paul Tarau)

The main contribution of this paper is a Catalan family based numbering system that *allows computations* with numbers comparable in size with Knuth's "arrow-up" notation. Moreover, these computations have a worst case 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.

To evaluate best and worst cases, a concept of *representation complexity* is introduced, based on the size of representations and algorithms favoring large numbers of small representation complexity are designed for arithmetic operations. Simple operations like successor, multiplication by 2, exponent of 2 are constant time when using tree-representations for Catalan objects, and a number of other operations benefit from significant complexity reductions on objects with a low representation size.

As the Catalan family [4, 5] contains a large number of computationally isomorphic but structurally distinct combinatorial objects, we will describe our arithmetic computations generically, using Haskell's *type classes* [6], of which typical members of the Catalan family, like binary trees, multiway trees and balanced parentheses languages will be described as instances.

At the same time, an *atypical instance* will be derived, representing the set of *natural numbers* $\mathbb{N}$, which will be used to validate the correctness of our generically defined arithmetic operations.

We have adopted 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 at `http://www.cse.unt.edu/~tarau/research/2014/Cats.hs` as a separate file. We hope that this will encourage the reader to experiment interactively and validate the technical correctness of our claims.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces a generic view of Catalan families as a Haskell type class, with subsection 3.5 embedding the set of natural numbers as an instance of the family. Section 4 introduces basic algorithms for arithmetic operations taking advantage of our number representation, with subsection 4.2 focusing on constant time successor and predecessor operations. Section 5 discusses a chain of mutually recursive definitions centered around addition and subtraction. Section 6 covers more advanced operations, centered around multiplication 6.1 and division 6.3. Section 7 defines a concept of representation complexity and studies best and worst cases. Section 8 describes examples of computation with very large numbers favored by our numbering system. Section 9 discusses some open problems and suggest future work. Section 10 concludes the paper. The **Appendix** overviews the subset of Haskell used in the paper as an executable mathematical notation.

2. Related work

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", after visiting arbitrarily large numbers, eventually turns around and terminates.

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 $\mathbb{N} \times \mathbb{N}$ and therefore the representation is not canonical. Moreover, the literate C-program that defines it only implements successor, addition, comparison and multiplication and does not provide a constant time exponent of 2 and low complexity leftshift / rightshift operations.

Conway's surreal numbers [8] 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 [2], 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 an extended and generalized version of [1] where a similar treatment of arithmetic operations is specialized to the language of balanced parentheses. It is also similar in purpose to [9], which describes a more complex hereditary number system (based on run-length encoded "bijective base 2" numbers, first introduced in [10] pp. 90-92 as "m-adic" numbers). Like in [1] and in contrast to [9], we are using here the familiar binary number system, and we represent our numbers as generic members of the Catalan family [4], rather than the more complex data structure used in [9].

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [11]. 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 [12] a binary tree representation enables arithmetic operations which are simpler but limited in efficiency to a smaller set of "sparse" numbers. In [13] this number representation's connexion to free algebras is explored.

In [14] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. However likewise [12] and [15], and in contrast to those proposed in this paper, they only compress "sparse" numbers, consisting of relatively few `1` bits in their binary representation.

While combinatorial enumeration and combinatorial generation, for which a vast literature exists (see for instance [4], [16], [17], [18], [19] and [20]), 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 combinatorial families. In fact, this is the main motivation and the most significant contribution of this paper.

3. The Catalan family of combinatorial objects

The Haskell data type `T` representing ordered rooted binary trees with empty leaves `E` and branches provided by the constructor `C` is a typical member of the Catalan family of combinatorial objects [4].

```
data T = E | C T T deriving (Eq,Show,Read)
```

Note the use of the type classes `Eq, Show` and `Read` to derive structural equality and respectively human readable output and input for this data type.

The data type `M` is another well-known member of the Catalan family, defining multiway ordered rooted trees with empty leaves.

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

Another representative of the Catalan family is the language of balanced parentheses. We fix our set of two parentheses to be {`L`,`R`} corresponding to the Haskell data type `Par`.

```
data Par = L | R deriving (Eq,Show,Read)
```

Definition 1. *A Dyck word on the set of parentheses* {`L`,`R`} *is a list consisting of n* `L`*'s and* `R`*'s such that no prefix of the list has more* `L`*'s than* `R`*'s.*

The set of Dyck words is a member of the Catalan family of combinatorial objects [4]. Let $\mathcal{P}$ be the language obtained from the set Dyck words on {L,R} with an extra `L` parenthesis added at the beginning of each word and an extra `R` parenthesis added at the end of each word. We represent the language $\mathcal{P}$ in Haskell as the type `P`.

```
data P = P [Par] deriving (Eq,Show,Read)
```

We will leave the enforcement of the balancing constraints to subsection 3.4 where data type `P` will be made an instance of the type class representing generically objects of the Catalan family.

3.1. A generic view of Catalan families as a Haskell type class

We will work through the paper with a generic data type ranging over instances of the type class `Cat`, representing a member of the Catalan family of combinatorial objects [4].

```
class (Show a,Read a,Eq a) ⇒ Cat a where
  e :: a

  c :: (a,a) → a
  c' :: a → (a,a)
```

The zero element is denoted `e` and we inherit from classes `Read` and `Show` which ensure derivation of input and output functions for members of type class `Cat` as well as from type class `Eq` that ensures derivation of the structural equality predicate `==` and its negation `/=`.

We will also define the corresponding recognizer predicates `e_` and `c_`, relying on the derived equality relation inherited from the Haskell type class `Eq`.

```
  e_ :: a → Bool
  e_ a = a == e

  c_ :: a → Bool
  c_ a = a /= e
```

For each instance, we assume that `c` and `c'` are inverses on their respective domains `Cat` × `Cat` and `Cat - {e}`, and `e` is distinct from objects constructed with `c`, more precisely that the following hold:

$$\forall x.\ c'(c\ x) = x \wedge \forall y.\ (c\_\ y \Rightarrow c\ (c'\ y) = y) \tag{1}$$

$$\forall x.\ (e\_\ x \vee c\_\ x) \wedge \neg(e\_\ x \wedge c\_\ x) \tag{2}$$

When talking about "objects of type `Cat`" we will actually mean an instance `a` of the polymorphic type `Cat a` that verifies equations (1) and (2).

3.2. The instance `T` *of ordered rooted binary trees*

The operations defined in type class `Cat` correspond naturally to the ordered rooted binary tree view of the Catalan family, materialized as the data type `T`.

```
instance Cat T where
  e = E

  c (x,y) = C x y

  c' (C x y) = (x,y)
```

Note that adding and removing the constructor `C` trivially verifies the assumption that our generic operations `c` and `c'` are inverses[1].

[1] In fact, one can see the functions `e, e_ , c, c', c_` as a generic API abstracting away the essential properties of the constructors `E` and `C`. In a language like Scala [21], that allows arbitrary functions to work as constructors / extractors, one could have defined them directly in terms of the `apply` and `unapply` methods used to define *type cases* [22]. As shown in [13], the same can be achieved using Haskell's *view* construct [23].

3.3. The instance `M` *of ordered rooted multiway trees*

The alternative view of the Catalan family as multiway trees is materialized as the data type `M`.

```
instance Cat M where
  e = F []
  c (x,F xs) = F (x:xs)
  c' (F (x:xs)) = (x,F xs)
```

Note that the assumption that our generic operations `c` and `c'` are inverses is easily verified in this case as well, given the bijection between binary and multiway trees. Moreover, note that operations on types `T` and `M` expressed in terms of their generic type class `Cat` counterparts result in a constant extra effort. Therefore, we will safely ignore it when discussing the complexity of different operations.

3.4. The instance `P` *of balanced parentheses*

Another well known representative of the Catalan family is the language of balanced parentheses. We refer to [1] for developing arithmetic operations specialized to them.

```
instance Cat P where
 e = P [L,R]

 c (P xs,P (L:ys)) = P (L:xs++ys)

 c' (P (L:ps)) = (P xs,P ys) where
  (xs,ys) = count_pars 0 ps

  count_pars 1 (R:ps)  = ([R],L:ps)
  count_pars k (L:ps) = (L:hs,ts) where
     (hs,ts) = count_pars (k+1) ps
  count_pars k (R:ps) = (R:hs,ts) where
     (hs,ts) = count_pars (k-1) ps
```

Note that the assumption that our generic operations `c` and `c'` are inverses is easily verified in this case as well.

As an interesting property, this representation is *self-delimiting*, in fact is a *reversible variable length code* i.e. it is uniquely decodable starting from either its beginning or its end. This property has been noticed as being important for encoding media streams [24], such codes being later adopted in encoding standards like MP4.

The following examples illustrate the generic operations `c` and `c'` on the instances `T`, `M` and `P` of type class `Cat`.

```
*Cats> c (E,c (E,E))
C E (C E E)
*Cats> c (F [F []],F [])
F [F [F []]]
*Cats> c' it
```

```
(F [F []],F [])
*Cats> c (P [L,R],P [L,R])
P [L,L,R,R]
*Cats> c' it
(P [L,R],P [L,R])
```

3.5. An unusual member of the Catalan family: the set of natural numbers $\mathbb{N}$

The (big-endian) binary representation of a natural number can be written as a concatenation of binary digits of the form

$$n = b_0^{k_0} b_1^{k_1} \dots b_i^{k_i} \dots b_m^{k_m} \tag{3}$$

with $b_i \in \{0, 1\}$ and the highest digit $b_m = 1$. The following hold.

Proposition 1. *An even number of the form* $0^i j$ *corresponds to the operation* $2^i j$ *and an odd number of the form* $1^i j$ *corresponds to the operation* $2^i(j+1)-1$.

Proof. It is clearly the case that $0^i j$ corresponds to multiplication by a power of 2. If $f(i) = 2i + 1$ then it is shown by induction (see [9]) that the i-th iterate of f, f^i is computed as in the equation (4)

$$f^i(j) = 2^i(j + 1) - 1 \tag{4}$$

Observe that each block 1^i in n, represented as $1^i j$ in equation (3), corresponds to the iterated application of f, i times, $n = f^i(j)$.

Proposition 2. *A number* n *is even if and only if it contains an even number of blocks of the form* $b_i^{k_i}$ *in equation (3). A number* n *is odd if and only if it contains an odd number of blocks of the form* $b_i^{k_i}$ *in equation (3).*

Proof. It follows from the fact that the highest digit (and therefore the last block in big-endian representation) is 1 and the parity of the blocks alternate.

This suggests defining the `c` operation of type class `Cat` as follows.

$$c(i, j) = \begin{cases} 2^{i+1} j & \text{if } j \text{ is odd,} \\ 2^{i+1}(j + 1) - 1 & \text{if } j \text{ is even.} \end{cases} \tag{5}$$

Note that the exponents are $i + 1$ instead of i as we start counting at 0. Note also that $c(i, j)$ will be even when j is odd and odd when j is even.

Proposition 3. *The equation (5) defines a bijection* $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+ = \mathbb{N} - \{0\}$.

Therefore `c` has an inverse `c'`, that we will constructively define together with `c`. The following Haskell code defines the instance of the Catalan family corresponding to $\mathbb{N}$.

```
type N = Integer
instance Cat Integer where
  e = 0

  c (i,j) | i≥0 && j≥0 = 2^(i+1)*(j+d)-d where
    d = mod (j+1) 2
```

The definition of the inverse `c'` relies on the *dyadic valuation* of a number n, $\nu_2(n)$, defined as the largest exponent of 2 dividing n implemented as the helper function `dyadicVal`.

```
  c' k | k>0 = (max 0 (x-1),ys) where
    b = mod k 2
    (i,j) = dyadicVal (k+b)
    (x,ys) = (i,j-b)

    dyadicVal k | even k = (1+i,j) where
       (i,j) = dyadicVal (div k 2)
    dyadicVal k = (0,k)
```

Note the use of the parity `b` in both definitions, which differentiates between the computations for *even* and *odd* numbers.

The following examples illustrate the use of `c` and `c'` on this instance.

```
*Cats> c (100,200)
5095955412917482194016746885611 51
*Cats> c' it
(100,200)
*Cats> map c' [1..10]
[(0,0),(0,1),(1,0),(1,1),(0,2),(0,3),(2,0),(2,1),(0,4),(0,5)]
*Cats> map c it
[1,2,3,4,5,6,7,8,9,10]
```

Figure 1 illustrates the DAG obtained by applying the operation `c'` repeatedly and merging identical subtrees. The order of the edges is marked with `0` and `1`.

3.6. The transformers: morphing between instances of the Catalan family

As all our instances implement the bijection `c` and its inverse `c'`, a generic transformer from an instance to another is defined by the function `view`:

```
view :: (Cat a, Cat b) ⇒ a → b
view z | e_ z = e
view z | c_ z = c (view x,view y) where (x,y) = c' z
```

To obtain transformers defining bijections with $\mathbb{N}$, `T`, `P` and `M` as ranges, we will simply provide specialized type declarations for them:

```
n :: Cat a ⇒ a→N
n = view
```

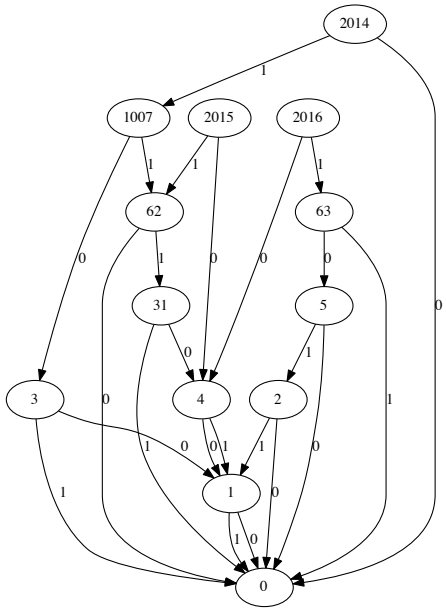


Figure 1: DAG representing 2014

```
t :: Cat a ⇒ a→T
t = view
```

```
m :: Cat a ⇒ a→M
m = view
```

```
p :: Cat a ⇒ a→P
p = view
```

The following examples illustrate the resulting specialized conversion functions:

```
*Cats> t 42
C E (C E (C E (C E (C E (C E (C E E)))))
*Cats> m it
F [F [],F [],F [],F [],F [],F []]
*Cats> p it
P [L,L,R,L,R,L,R,L,R,L,R,L,R,R]
*Cats> n it
42
```

A list view of an instance of type class `Cat` is obtained by iterating the constructor `c` and its inverse `c'`.

```
to_list :: Cat a ⇒ a → [a]
to_list x | e_ x = []
to_list x | c_ x  = h:hs where
    (h,t) = c' x
```

```
    hs = to_list t

from_list :: Cat a ⇒ [a] → a
from_list [] = e
from_list (x:xs) = c (x,from_list xs)
```

They work as follows:

```
*Cats> to_list 2014
[0,3,0,4]
*Cats> from_list it
2014
```

The function `to_list` corresponds to the children of a node in the multiway tree view provided by instance `M`.

The function `catShow` provides a view as a string of balanced parentheses.

```
catShow :: Cat a ⇒ a → [Char]
catShow x | e_ x = "()"
catShow x | c_ x = r where
    xs = to_list x
    r = "(" ++ (concatMap catShow xs) ++ ")"
```

It is illustrated below.

```
*Cats> catShow 0
"()"
*Cats> catShow 1
"(())"
*Cats> catShow 12345
"((()(()))(()())(()(()())(())))"
```

Figure 2 shows the DAG corresponding to a multiway tree view, with merged identical subtrees, and labels showing the result of function `catShow` together with the corresponding natural numbers. The order of edges is marked with consecutive integers starting from `0`.

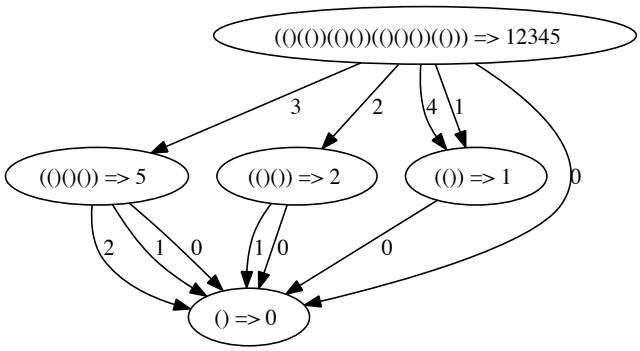


Figure 2: DAG representing 12345

4. Generic arithmetic operations on members of the Catalan family

We will now implement arithmetic operations on Catalan families, generically, in terms of the operations on type class `Cat`.

4.1. Basic Utilities

We start with some simple functions to be used later.

4.1.1. Inferring even and odd

As we know for sure that the instance $\mathbb{N}$, corresponding to natural numbers supports arithmetic operations, we will try to mimic their behavior at the level of the type class `Cat`.

The operations `even_` and `odd_` implement the observation following from of Prop. 2 that parity (staring with 1 at the highest block) alternates with each block of distinct 0 or 1 digits.

```
even_ :: Cat a ⇒ a → Bool
even_ x | e_ x = True
even_ z | c_ z = odd_ y where (_,y)=c' z

odd_ :: Cat a ⇒ a → Bool
odd_ x | e_ x = False
odd_ z | c_ z = even_ y where (_,y)=c' z
```

4.1.2. One

We also provide a constant `u` and a recognizer predicate `u_` for 1.

```
u :: Cat a ⇒ a
u = c (e,e)

u_ :: Cat a ⇒ a→ Bool
u_ z = c_ z && e_ x && e_ y where (x,y) = c' z
```

4.2. Average constant time successor and predecessor

We will now specify successor and predecessor on the family of data types `Cat` through two mutually recursive functions, `s` and `s'`.

They first decompose their arguments using `c'`. Then, after transforming them as a result of adding `1`, they place back the results with the `c` operation. Note that the two functions work *on a block of* `0` *or* `1` *digits at a time*. They are based on arithmetic observations about the behavior of these blocks when incrementing or decrementing a binary number by `1`.

```
s :: Cat a ⇒ a → a
s x | e_ x = u -- 1
s z | c_ z && e_ y = c (x,u) where -- 2
   (x,y) = c' z
```

For the general case, the successor function `s` delegates the transformation of the blocks of 0 and 1 digits to functions `f` and `g` handling `even_` and respectively `odd_` cases.

```
s a | c_ a = if even_ a then f a else g a where
   f k | c_ w && e_ v = c (s x,y) where -- 3
    (v,w) = c' k
    (x,y) = c' w
   f k = c (e, c (s' x,y)) where -- 4
     (x,y) = c' k
   g k | c_ w && c_ n && e_ m = c (x, c (s y,z)) where -- 5
    (x,w) = c' k
    (m,n) = c' w
    (y,z) = c' n
   g k | c_ v = c (x, c (e, c (s' y, z))) where -- 6
    (x,v) = c' k
    (y,z) = c' v
```

The predecessor function `s'` inverts the work of `s` as marked by a comment of the form `k --`, for `k` ranging from `1` to `6`.

```
s' :: Cat a ⇒ a → a
s' k | u_ k = e where -- 1
    (x,y) = c' k
s' k | c_ k && u_ v = c (x,e) where -- 2
    (x,v) = c' k
```

For the general case, the predecessor function `s'` delegates the transformation of the blocks of 0 and 1 digits to functions `g` and `f` handling `even_` and respectively `odd_` cases.

```
s' a | c_ a = if even_ a then g' a else f' a where
     g' k | c_ v && c_ w && e_ r = c (x, c (s y,z)) where -- 6
       (x,v) = c' k
       (r,w) = c' v
       (y,z) = c' w
     g' k  | c_ v = c (x,c (e, c (s' y, z))) where -- 5
       (x,v) = c' k
       (y,z) = c' v
     f' k | c_ v && e_ r = c (s x,z) where -- 4
        (r,v) = c' k
        (x,z) = c' v
     f' k =  c (e, c (s' x,y)) where -- 3
        (x,y) = c' k
```

One can see that their use matches successor and predecessor on instance `N`:

```
*Cats> map s [0..15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
*Cats> map s' it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

The following holds:

Proposition 4. *Denote* $\texttt{Cat}^+ = \texttt{Cat} - \{\texttt{e}\}$. *The functions* $s : \texttt{Cat} \to \texttt{Cat}^+$ *and* $s' : \texttt{Cat}^+ \to \texttt{Cat}$ *are inverses.*

PROOF. For each instance of `Cat`, 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 $< \texttt{Cat}, \texttt{e}, \texttt{s} >$ is a *Peano algebra*. This is expected, as `s` provides a combinatorial enumeration of the infinite stream of Catalan objects, as illustrated below on instance `T`:

```
Cats> s E
C E E
*Cats> s it
C E (C E E)
*Cats> s it
C (C E E) E
*Cats> s it
C (C E E) (C E E)
*Cats> s it
C E (C E (C E E))
*Cats> s it
C E (C (C E E) E)
```

Note 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. We will also assume, that when complexity is discussed, *a representation like the tree data types* `T` *or* `M` *are used, making the operations* `c` *and* `c'` *constant time.* Note also that this is clearly not the case for the instance `N` using the traditional bitstring representation or the instance `P` where linear scanning proportional to the length of the sequence may be involved.

Proposition 5. *The worst case time complexity of the* `s` *and* `s'` *operations on* n *is given by the* iterated logarithm $O(log_2^*(n))$.

PROOF. Note that calls to `s,s'` in `s` or `s'` happen on 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 6. `s` *and* `s'` *are constant time, on the average.*

PROOF. Observe that the average size of a contiguous block of `0`s or `1s` 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.

4.3. A few other average constant time operations

We will derive a few operations that inherit their complexity from `s` and `s'`.

4.3.1. Double and half

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 :: Cat a ⇒ a → a
db x | e_ x  = e
db x | odd_ x = c (e,x)
db z = c (s x,y) where (x,y) = c' z
```

```
hf :: Cat a ⇒ a → a
hf x | e_ x = e
hf z | e_ x = y where (x,y) = c' z
hf z  = c (s' x,y) where (x,y) = c' z
```

4.3.2. Exponent of 2 and its left inverse

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 `c` and `s'` from which it inherits its complexity up to a constant factor.

```
exp2 :: Cat a ⇒ a → a
exp2 x | e_ x = u
exp2 x = c (s' x, u)
```

The same applies to its left inverse `log2`:

```
log2 :: Cat a ⇒ a → a
log2 x | u_ x = e
log2 x | u_ z = s y where (y,z) = c' x
```

Proposition 7. *The operations* `db`, `hf`, `exp2` *and* `log2` *are average constant time and are* log^* *in the worst case.*

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

We illustrate their work on instances `N`:

```
*Cats> map exp2 [0..15]
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768]
*Cats> map log2 it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

More interestingly, a tall tower of exponents that would overflow memory on instance `N`, is easily supported on instances `C, T` and `P` as shown below:

```
*Cats> exp2 (exp2 (exp2 (exp2 (exp2 (exp2 E))))))
C (C (C (C (C (C E E) E) E) E) E) (C E E)
*Cats> t it
C (C (C (C (C (C E E) E) E) E) E) (C E E)
*Cats> exp2 (exp2 (exp2 (exp2 (exp2 (exp2 (exp2 E))))))
C (C (C (C (C (C E E) E) E) E) E) (C E E)
*Cats> p it
P [L,L,L,L,L,L,R,R,R,R,R,R,L,R,R]
*Cats> exp2 (exp2 (exp2 (exp2 (exp2 (exp2 (exp2 E))))))
C (C (C (C (C (C E E) E) E) E) E) (C E E)
*Cats> m it
F [F [F [F [F [F [F []]]]]],F []]
*Cats> p it
P [L,L,L,L,L,L,R,R,R,R,R,R,L,R,R]
*Cats> log2 (log2 (log2 (log2 (log2 (log2 (log2 it))))))
P [L,R]
*Cats> t it
E
```

This example illustrates the main motivation for defining arithmetic computation with the "typical" members of the Catalan family: their ability to deal with giant numbers.

5. Addition, subtraction and their mutually recursive helpers

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

5.1. Multiplication with a power of 2

We start with the functions `leftshiftBy`, `leftshiftBy'` and `leftshiftBy"` corresponding respectively to $2^n k$, $(\lambda x.2x+1)^n(k)$ and $(\lambda x.2x+2)^n(k)$.

The function `leftshiftBy` prefixes an odd number with a block of 1s and extends a block of 0s by incrementing their count.

```
leftshiftBy :: Cat a ⇒ a → a → a
leftshiftBy x y | e_ x = y
leftshiftBy _ y | e_ y = e
leftshiftBy x y | odd_ y = c (s' x, y)
leftshiftBy x v = c (add x y, z) where (y,z) = c' v
```

The function `leftshiftBy'` is based on equation (6).

$$(\lambda x.2x+1)^n(k) = 2^n(k+1) - 1 \tag{6}$$

```
leftshiftBy' :: Cat a ⇒ a → a → a
leftshiftBy' x k = s' (leftshiftBy x (s k))
```

The function `leftshiftBy'` is based on equation (7) (see [9] for an direct proof by induction).

$$(\lambda x.2x+2)^n(k) = 2^n(k+2) - 2 \tag{7}$$

```
leftshiftBy'' :: Cat a ⇒ a → a → a
leftshiftBy'' x k = s' (s' (leftshiftBy x (s (s k))))
```

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`s, `leftshiftBy'` just increments that count. As a result, the algorithm favors numbers with relatively few large blocks of `0` and `1` digits.

5.2. Addition and subtraction, optimized for numbers built from a few large blocks of `0`*s and* `1`*s*

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

```
add :: Cat a ⇒ a → a → a
add x y | e_ x = y
add x y | e_ y = x
```

In the case when both terms represent even numbers, the two blocks add up to an even block of the same size. Note the use of `cmp` and `sub` in helper function `f` to trim off the larger block such that we can operate on two blocks of equal size.

```
add x y |even_ x && even_ y = f (cmp a b) where
  (a,as) = c' x
  (b,bs) = c' y
  f EQ = leftshiftBy (s a) (add as bs)
  f GT = leftshiftBy (s b) (add (leftshiftBy (sub a b) as) bs)
  f LT = leftshiftBy (s a) (add as (leftshiftBy (sub b a) 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 y |even_ x && odd_ y = f (cmp a b) where
  (a,as) = c' x
  (b,bs) = c' y
  f EQ = leftshiftBy' (s a) (add as bs)
  f GT = leftshiftBy' (s b) (add (leftshiftBy (sub a b) as) bs)
  f LT = leftshiftBy' (s a) (add as (leftshiftBy' (sub b a) 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 (8):

$$(\lambda x.2x+1)^k(x) + (\lambda x.2x+1)^k(y) = (\lambda x.2x+2)^k(x+y) \tag{8}$$

```
add x y | odd_ x && odd_ y = f (cmp a b) where
  (a,as) = c' x
  (b,bs) = c' y
  f EQ =  leftshiftBy'' (s a) (add as bs)
  f GT =  leftshiftBy'' (s b) (add (leftshiftBy' (sub a b) as) bs)
  f LT =  leftshiftBy'' (s a) (add as (leftshiftBy' (sub b a) 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 :: Cat a ⇒ a → a → a
sub x y | e_ y = x
sub x y | even_ x && even_ y = f (cmp a b) where
  (a,as) = c' x
  (b,bs) = c' y
  f EQ = leftshiftBy (s a) (sub as bs)
  f GT = leftshiftBy (s b) (sub (leftshiftBy (sub a b) as) bs)
  f LT = leftshiftBy (s a) (sub as (leftshiftBy (sub b a) bs))
```

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

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

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

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

Finally, when the first block is `0` and the second is `1` an identity dual to (8) is used:

```
sub x y | even_ x && odd_ y = f (cmp a b) where
  (a,as) = c' x
  (b,bs) = c' y
  f EQ = s (leftshiftBy (s a) (sub1 as bs))
  f GT = s (leftshiftBy (s b) (sub1 (leftshiftBy (sub a b) as) bs))
  f LT = s (leftshiftBy (s a) (sub1 as (leftshiftBy' (sub b a) 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. 6), so their average complexity is within constant factor of their ordinary counterparts.

On the other hand, as they are limited by the representation size of the operands rather than their bitsize, when compared with their bitstring counterparts, these 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 on them.

The following examples illustrate the agreement with their usual counterparts:

```
*Cats> map (add 10) [0..15]
[10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
*Cats> map (sub 15) [0..15]
[15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0]
```

5.3. Comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our generic type `Cat`. It relies on `bitsize` computing the number of binary digits constructing a term in `Cat`, also 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 8. *Let* `bitsize` *count the number of digits of a base-2 number, with the convention that it is* `0` *for* `0`*. Then* $\texttt{bitsize}(x) < \texttt{bitsize}(y) \Rightarrow x < y$.

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 :: Cat a⇒ a→a→Ordering
cmp x y | e_ x && e_ y = EQ
cmp x _ | e_ x = LT
cmp _ y | e_ y = GT
cmp x y | u_ x && u_ (s' y) = LT
cmp  x y | u_ y && u_ (s' x) = GT
```

For instance, it is easy to see that comparison of `x` and `y` can be reduced to comparison of bitsizes when they are distinct. Note that `bitsize`, to be defined later, is part of the chain of our mutually recursive functions.

```
cmp x y | x' /= y'  = cmp x' y' where
  x' = bitsize x
  y' = bitsize y
```

When bitsizes are equal, a more elaborate comparison needs to be done, delegated to function `compBigFirst`.

```
cmp xs ys = compBigFirst True True (rev xs) (rev ys) where
  rev = from_list . reverse . to_list
```

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 9. *Assuming two terms of the same bitsizes, the one with* 1 *as its first before the highest order digit, is larger than the one with* 0 *as its first before the highest order digit.*

Proof. Observe that little-endian numbers obtained by applying the function `rev` 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.

```
  compBigFirst _ _ x y | e_ x && e_ y = EQ
  compBigFirst False False x y = f (cmp a b) where
    (a,as) = c' x
    (b,bs) = c' y
    f EQ = compBigFirst True True as bs
    f LT = GT
    f GT = LT
  compBigFirst True True x y = f (cmp a b) where
    (a,as) = c' x
    (b,bs) = c' y
    f EQ = compBigFirst False False as bs
    f LT = LT
    f GT = GT
  compBigFirst False True x y = LT
  compBigFirst True False x y = GT
```

The following examples illustrate the agreement of `cmp` with the usual order relation on $\mathbb{N}$.

```
*Cats> cmp 5 10
LT
*Cats> cmp 10 10
EQ
*Cats> cmp 10 5
GT
```

The function `bitsize`, last in our chain of mutually recursive functions, computes the number of digits, except that we define it as `e` for constant function `e` (corresponding to `0`). It works by summing up the counts of `0` and `1` digit blocks composing a tree-represented natural number.

```
bitsize :: Cat a ⇒ a → a
bitsize z | e_ z = z
bitsize  z = s (add x (bitsize y)) where (x,y) = c' z
```

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

```
ilog2 :: Cat a ⇒ a→a
ilog2 = s' . bitsize
```

6. Algorithms for advanced arithmetic operations

6.1. Multiplication, optimized for large blocks of `0`*s and* `1`*s*

Devising a similar optimization as for `add` and `sub` for multiplication (`mul`) is actually easier.

After making sure that the recursion is on its smaller argument, `mul` delegates its work to `mul1`. When the first term represents an even number, `mul1` applies the `leftshiftBy` operation and it reduces the other case to this one.

```
mul :: Cat a ⇒ a → a → a
mul x y = f (cmp x y) where
  f GT = mul1 y x
  f _ = mul1 x y

mul1 :: Cat a ⇒ a → a → a
mul1 x _ | e_ x = e
mul1 a y | even_ a =  leftshiftBy (s x) (mul1 xs y) where (x,xs) = c' a
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 its number of digits. The following example illustrates a blend of arithmetic operations benefiting from complexity reductions on giant tree-represented numbers:

```
*Cats> let term1 = sub (exp2 (exp2 (t 12345))) (exp2 (t 6789))
*Cats> let term2 = add (exp2 (exp2 (t 123))) (exp2 (t 456789))
*Cats> bitsize (bitsize (mul term1 term2))
C E (C E (C E (C (C E (C E E)) (C (C E (C E (C E E))) (C (C E E) E)))))
*Cats> n it
12346
```

6.2. Power

After specializing our multiplication for a squaring operation,

```
square :: Cat a ⇒ a → a
square x = mul1 x x
```

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

```
pow :: Cat a ⇒ a → a → a
pow _ b | e_ b = u
pow a _ | e_ a = e
pow a b | even_ a = c (s' (mul (s x) b),ys) where
  (x,xs) = c' a
  ys = pow xs b
pow a b | even_ b = pow (superSquare y a) ys where
  (y,ys) = c' b
  superSquare k x | e_ k = 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:

```
*Cats> n (bitsize (pow (t 10) (t 100)))
333
*Cats> pow (m 32) (m 10000000)
F [F [F [F [],F [F []]],F [F [F []],F []],F [F [F []]],
  F [],F [],F [],F [F [F []],F []],F [],F [],F []]
  *Cats> catShow it
"((((()(()))((()))(()))((()))(()()(()))(()))()())())"
```

6.3. Division operations

We start by defining an efficient special case.

6.3.1. A special case: division by a power of 2

The function `rightshiftBy` 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 :: Cat a ⇒ a → a → a
rightshiftBy x y | e_ x = y
rightshiftBy _ y | e_ y = e
rightshiftBy x y = f (cmp x a')  where
  (a,b) = c' y
  a' = s a
  f LT = c (sub a x,b)
  f EQ = b
  f GT = rightshiftBy (sub  x a') b
```

6.3.2. General division

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

```
div_and_rem :: Cat a ⇒ a → a → (a, a)
div_and_rem x y | LT == cmp x y = (e,x)
div_and_rem x y | c_ y  = (q,r) where
  (qt,rm) = divstep x y
  (z,r) = div_and_rem rm y
  q = add (exp2 qt) z
```

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

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

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

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

Division and remainder are obtained by specializing `div_and_rem`.

```
divide :: Cat b ⇒ b → b → b
divide n m = fst (div_and_rem n m)

remainder :: Cat b ⇒ b → b → b
remainder n m = snd (div_and_rem n m)
```

The following examples illustrate the agreement with their usual counterparts:

```
*Cats> divide 26 3
8
*Cats> remainder 26 3
2
```

7. Representation complexity

While a precise average complexity analysis of 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 for all our operations, *their average complexity matches their traditional counterparts*, using the fact, shown in the proof of Prop. 6, that the average size of a block of contiguous `0` or `1` bits is at most `2`.

7.1. 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 `catsize` that counts the non-empty nodes of an object of type `Cat`.

```
catsize :: Cat a ⇒ a → a
catsize z | e_ z = z
catsize  z = s (add (catsize x) (catsize y)) where (x,y) = c' z
```

The following holds:

Proposition 10. *For all terms* $t \in$ `Cat`, `catsize t` $\leq$ `bitsize t`.

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

The following example illustrates their use:

```
*Cats> map catsize [0,100,1000,10000]
[0,7,9,13]
*Cats> map catsize [2^16,2^32,2^64,2^256]
[5,6,6,6]
*Cats> map bitsize [2^16,2^32,2^64,2^256]
[17,33,65,257]
```

Figure 3 shows the reductions in representation complexity compared with bitsize for an initial interval of $\mathbb{N}$, from 0 to $2^{10} - 1$.

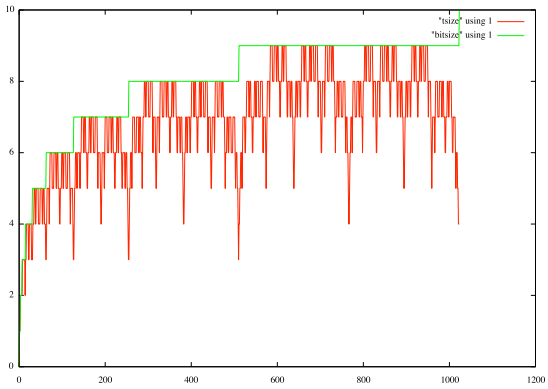


Figure 3: Representation complexity (lower line) bounded by bitsize (upper line)

7.2. Enumerating objects of given representation size

The total number of Catalan objects corresponding to n is given by:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \tag{9}$$

It is shown in [4] that if $L_n = \frac{2^{2n}}{n^{\frac{3}{2}}\sqrt{\pi}}$ then $\lim_{n\to\infty} \frac{C_n}{L_n} = 1$, providing an asymptotic bound for C_n.

The function `cat` describes an efficient computation for of the Catalan number C_n using a direct recursion formula derived from equation (9).

```
cat :: N→N
cat 0 = 1
cat n | n>0 = (2*(2*n-1)*(cat (n-1))) `div` (n+1)
```

The first few members of the sequence are computed below:

```
*Cats> map cat [0..14]
[1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900,2674440]
```

The following holds.

Proposition 11. *Let* `k = catsize x` *where* `x` *is an object of type* `Cat`*. Then* `x` *relates to* `k` *as follows, for instances of* `Cat`*:*

1. `x` *is a binary tree of type* `T` *with* k *internal nodes and* $k+1$ *leaves*
2. `x` *is a multiway tree of type* `M` *with* $k+1$ *nodes*
3. `x` *is a term of the type* `P` *with* $2k+2$ *matching parentheses.*

Proof. Observe that `catsize k` counts the $C_{\mathtt{k}} - 1$ number of `C` constructors in objects of size `k` of type `T`. The rest is a consequence of well-known relations between Catalan numbers and nodes of binary trees, nodes of multiway trees and parentheses in Dyck words as given in [4].

The function `catsized` enumerates for each of our instances, the objects of size `k` corresponding to a given Catalan number.

```
catsized :: Cat a ⇒ a → [a]
catsized a = take k [x | x←iterate s e,catsize x == a] where
  k = fromIntegral (cat (n a))
```

The function extracts exactly `k` elements (with `k` the Catalan number corresponding to size `a`) from the infinite enumeration of Catalan objects of type `Cat` provided by `iterate s e`, as illustrated below:

```
*Cats> catsized (t 2)
[C E (C E E),C (C E E) E]
*Cats> catsized 4
[8,9,10,11,12,13,14,16,30,31,63,127,255,65535]
```

7.3. Best and worst cases

Next we define the higher order function `iterated` that applies `k` times the function `f`, which, contrary to Haskell's `iterate`, returns only the final element rather than building the infinite list of all iterates.

```
iterated :: Cat a ⇒ (t → t) → a → t → t
iterated f k x |e_ k = x
iterated f k x = f (iterated f (s' k) x)
```

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

```
bestCase :: Cat a ⇒ a → a
bestCase k = iterated f k e where f x = c (x,e)
```

and a worst case

```
worstCase :: Cat a ⇒ a → a
worstCase k = iterated f k e where f x = c (e,x)
```

The following examples illustrate these functions:

```
*Cats> bestCase (t 5)
C (C (C (C (C E E) E) E) E) E
*Cats> n (bitsize (bestCase (t 5)))
65536
*Cats> n (catsize (bestCase (t 5)))
5
*Cats> worstCase (t 5)
C E (C E (C E (C E (C E E))))
*Cats> n (bitsize (worstCase (t 5)))
5
*Cats> n (catsize (worstCase (t 5)))
5
```

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

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

For $k = 5$ it corresponds to $2^{65536} - 1$.

Note that our concept of representation complexity is only a weak approximation of Kolmogorov complexity [25]. 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 `catsize`, 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.

7.4. A Concept of duality

As our instances of `Cat` are members of the Catalan family of combinatorial objects, they can be seen as binary trees with empty leaves. Therefore, we can transform the tree representation of our objects by swapping left and right branches under a binary tree view, recursively. The corresponding Haskell code is:

```
dual :: Cat a ⇒ a → a
dual x | e_ x = e
dual z = c (dual y,dual x) where (x,y) = c' z
```

As clearly `dual` is an *involution* (i.e., `dual` ∘ `dual` is the identity of `Cat`), the corresponding permutation of ℕ will bring together huge and small natural numbers sharing representations of the same size, as illustrated by the following example.

```
*Cats> map dual [0..20]
[0,1,3,2,4,15,7,6,12,31,65535,16,8,255,127,5,11,8191,4294967295,32,65536]
*CatsBM> catShow 10
"(()()()())"
*CatsBM> catShow (dual 10)
"((((()))))"
```

For instance, `18` and its dual `4294967295` have 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. 4, with trees folded to DAGs by merging together shared subtrees. Note the significantly different bitsizes that can result for a term and its dual.

```
*Cats> m 18
F [F [],F [],F [F []],F []]
*Cats> dual (m 18)
F [F [F [F [F []],F []]]]
*Cats> n (bitsize (m 18))
5
*Cats> n (bitsize (dual (m 18)))
32
```

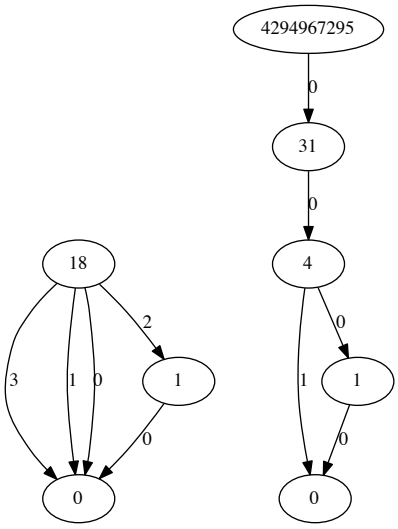
It follows immediately from the definitions of the respective functions, that, as an extreme case, the following holds:

Proposition 12. ∀`x.  dual (bestCase x)` = `worstCase x`.

The following example illustrates it, with a tower of exponents `10000` tall, reached by `bestCase`. Note that we run it on objects of type `T`, as it would dramatically overflow memory on bitstring-represented numbers of type ℕ.

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

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



(a) 18 (b) dual of 18

Figure 4: 18 and its dual, with multiway trees folded to DAGs

```
*Cats> [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]
*Cats> [x|x<-[0..2^5-1],cmp (t x) (dual (t x)) ==  EQ]
[0,1,4,24]
*Cats> [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.

The existence of such a bijection (computed by a program of constant size) between natural numbers of high and low Kolmogorov complexity reveals a somewhat non-intuitive aspect of this concept and its use for the definition of randomness [25].

We will explore next definitions for concepts of depth for our number representation.

7.5. Representation Depth

As we can switch between the binary and multiway view of our Catalan objects, we will define two sets of representation-depth functions. They use the the helper functions minimum `min2` and maximum `max2`.

```
min2, max2 :: Cat a ⇒ a → a → a
min2 x y = if LT==cmp x y then x else y
max2 x y = if LT==cmp x y then y else x
```

Corresponding to the *binary tree view* exemplified by instance `T`, we define `maxTdepth` returning the length of the longest path from the root to a leaf.

```
maxTdepth :: Cat a ⇒ a → a
maxTdepth z | e_ z = z
maxTdepth z = s (max2  (maxTdepth x) (maxTdepth y)) where (x,y) = c' z
```

Corresponding to the *multiway tree view* exemplified by instance `M` we define `maxTdepth` returning the length of the longest path from the root to a leaf.

```
maxMdepth :: Cat a ⇒ a → a
maxMdepth z | e_ z = z
maxMdepth z = s (foldr max2 m ms) where
  (m:ms) = map maxMdepth (to_list z)
```

The following simple facts hold, derived from properties of binary and multiway rooted ordered trees.

Proposition 13. *Let* `x` $\geq$ `y` *stand for* `cmp x y == GT` *and* $=$ *stand for* `cmp x y == EQ`.

1. *For all objects* `x` *of type* `Cat`, `catsize x` $\geq$ `maxTdepth x` $\geq$ `maxMdepth x`.
2. *For all objects* `x` *of type* `Cat`, `catsize x` $=$ `catsize (dual x)`
3. *For all objects* `x` *of type* `Cat` `maxTdepth x` $=$ `maxTdepth (dual x)`.
4. *For all objects* `x` *of type* `Cat`, `maxMdepth (bestCase x)` $=$ `x`.

8. Compact representation and tractable computations with some giant numbers

We will illustrate the representation and computation power of our new numbering system with two case studies. The first shows that several *record holder primes* have compact Catalan representations.

The second shows computation of the hailstone sequence for an equivalent of the *Collatz conjecture* on giant numbers.

8.1. Record holder primes

Interestingly, most record holder giant primes have a fairly simple structure: they are of the form $p = i2^k \pm j$ with $i \in \mathbb{N}$ and $j \in \mathbb{N}$ comparatively small. This is a perfect fit for our representation which favors numbers "in the neighborhood" of linear combinations of (towers of) exponents of two with

Record holder prime	bitsize	catsize
Mersenne prime	57,885,161	25
Generalized Fermat prime	9,167,448	37
Cullen prime	6,679,904	46
Woodall prime	3,752,970	37
Sophie Germain prime	666,712	62
Twin primes 1	666,711	59
Twin primes 2	666,711	60

Figure 5: Bitsizes vs. Catalan representation sizes of record holder primes

comparatively small coefficients, resulting in large contiguous blocks of 0s and 1s when represented as bitstrings.

The largest known primes (as of early 2014) of several types are given by the following Haskell code.

```
mersennePrime f = s' (exp2 (f 57885161))
generizedFermatPrime f = s (leftshiftBy (f 9167433) (f 27653))
cullenPrime f = s (leftshiftBy x x) where x = f 6679881
woodallPrime f = s' (leftshiftBy x x) where x = f 3752948
prothPrime f = s (leftshiftBy (f 13018586) (f 19249))
sophieGermainPrime f = s' (leftshiftBy (f 666667) (f 18543637900515))
twinPrimes f = (s' y,s y) where
  y = leftshiftBy (f 666669) (f 3756801695685)
```

For instance, the largest known prime, having about 17 million decimal digits, (a Mersenne number) has an unusually small Catalan representation as illustrated below:

```
*Cats> catShow (mersennePrime t)
"((((()))()()(((()))(((()))()))()()(()))(((()))()(((()))()(((())))"
*Cats> n (catsize (mersennePrime t))
25
*Cats> n (bitsize (mersennePrime t))
57885161
```

Note the use of parameter `t` indicating that computation proceeds with type `T`, as it would overflow memory with with bitstring-represented natural numbers.

Figure 5 summarizes comparative bitstring and Catalan representation sizes `bitsize` and `catsize` for record holder primes.

8.2. 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 representation complexity. The Collatz conjecture [26] states that the function

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

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

$$collatz'(x) = \begin{cases} \frac{x}{2^{\nu_2(x)}} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{11}$$

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

$$tl(k) = \frac{\frac{k}{2^{\nu_2(k)}} - 1}{2} \tag{12}$$

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

```
hd x = fst (decons x)

tl x = snd (decons x)

decons a | even_ a = (s x,hf (s' xs)) where (x,xs) = c' a
decons a = (e,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

$$syracuse(n) = tl(3n+2) \tag{13}$$

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

```
syracuse :: Cat b ⇒ b → b
syracuse n = tl (add n (db (s n)))
```

Note that all operations except the addition `add` are constant average time.

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

```
nsyr :: Cat t ⇒ t → [t]
nsyr x | e_ x = [e]
nsyr x = x : nsyr (syracuse x)
```

It is easy to see that the Collatz conjecture is true[2] if and only if `nsyr` terminates for all n, as illustrated by the following example:

```
*Cats> nsyr 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]
```

Moreover, in this formulation, the conjecture implies that the the elements of sequence generated by `nsyr` are all different.

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 more transistors than 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*:

```
*Cats> take 100 (map(n . catsize) (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 `catsize` of these numbers, which, obviously, would not fit themselves in a decimal representation.

A slightly longer computation (taking a few minutes) can be also performed on a twin tower of exponents `101` and `103` levels tall like in

```
*Cats> take 2 (map(n.catsize) (nsyr
         (add (bestCase (t 101)) (bestCase (t 103)))))
[10206,10500]
```

where the Catalan representation size at `10500`, proportional to the product of the representation sizes of the operands, slows down computation but still keeps it in a tractable range.

9. Discussion

Our Catalan families based numbering system provides compact representations of giant numbers and can perform interesting computations intractable with their bitstring-based counterparts.

This ability comes from the fact that our canonical tree representation, in contrast to the traditional binary representation supports constant average time and space application of exponentials.

We have not performed a precise time and space complexity analysis (except for the the constant average-time operations), but our experiments indicate that (low) polynomial bounds are likely for addition and subtraction with worst cases

[2] As a side note, it might be interesting to approach the Collatz conjecture by trying to compare the growth in the *Catalan representation size* induced by `3n+2` expressed as `add n (db (s n))` vs. its decrease induced by `tl`.

of size expansion happening with towers of exponents, where results are likely to be proportional to the product of the height of the towers, as illustrated in subsection 8.2.

Our multiplication and division algorithms are derived from relatively simple traditional ones, with some focus on taking advantage of large blocks of 0 and 1 digits. However, it would be interesting to further explore asymptotically better algorithms like Karatsuba multiplication or division based on Newton's method.

As most numbers have high Kolmogorov complexity, it makes sense to extend our type class mechanism to devise a *hybrid* numbering system that switches between representations as needed, to delegate cases where there are no benefits, to the underlying bitstring representation. This is likely to be needed for designing a practical extension with Catalan representations for the arithmetic subsystem used in a programming language like Haskell that benefits from the very fast C-based GMP library.

10. Conclusion

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

We have shown that arithmetic *computations* like addition, subtraction, multiplication, bitsize and 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 operations 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.

The resulting numbering system is *canonical* - each natural number is represented as a unique object. Besides unique decoding, canonical representations allow testing for *syntactic equality*.

It is also *generic* – no commitment is made to a particular member of the Catalan family – our type class provides all the arithmetic operations to several instances, including typical members of the Catalan family together with the usual natural numbers.

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 likely to apply to several practical data representations ranging from quad-trees to audio/video encoding formats.

References

References

[1] P. Tarau, Computing with Catalan Families, in: A.-H. Dediu, C. Martin-Vide, J.-L. Sierra, B. Truthe (Eds.), Proceedings of Language and Automata Theory and Applications, 8th International Conference, LATA 2014, Springer, LNCS, Madrid, Spain,, 2014, pp. 564–576.

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

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

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

[5] N. J. A. Sloane, A000108, The On-Line Encyclopedia of Integer SequencesPublished electronically at http://oeis.org/A000108.

[6] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad-hoc, in: POPL, 1989, pp. 60–76.

[7] D. E. Knuth, TCALC program, `http://www-cs-faculty.stanford.edu/~uno/programs/tcalc.w.gz` (Dec. 1994).

[8] J. H. Conway, On Numbers and Games, 2nd Edition, AK Peters, Ltd., 2000.

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

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

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

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

[13] P. Tarau, Computing with Free Algebras, in: A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. Watt, D. Zaharie (Eds.), Proceedings of SYNASC 2012, IEEE, 2013, pp. 15–22, invited talk.

[14] J. Vuillemin, Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates, in: Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on, 2009, pp. 7 –14.

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

[16] N. Hungerbühler, The isomorphism problem for catalan families, J. Combin. Inform. System Sci 20 (1995) 129–139.

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

[18] D. 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.

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

[20] C. Martinez, X. Molinero, Generic algorithms for the generation of combinatorial objects., in: B. Rovan, P. Vojtas (Eds.), MFCS, Vol. 2747 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2003, pp. 572–581.

[21] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima Inc, 2008.

[22] B. Emir, M. Odersky, J. Williams, Matching objects with patterns, in: E. Ernst (Ed.), ECOOP 2007 - Object-Oriented Programming, Vol. 4609 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 273–298.

[23] P. Wadler, Views: a way for pattern matching to cohabit with data abstraction, in: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '87, ACM, New York, NY, USA, 1987, pp. 307–313.

[24] J. Wen, J. Villasenor, Reversible variable length codes for efficient and robust image and video coding, in: Proceedings Data Compression Conference, 1998, pp. 471–480.

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

[26] J. C. Lagarias, The 3x+1 Problem: An Annotated Bibliography (1963-1999), http://arXiv.org, 0309224v11 (2008).

Appendix

A subset of Haskell as an executable function notation

We have refrained from using any advanced features of Haskell to ensure that our minimalist subset of the language is understandable to the reader familiar only with the usual mathematical notation conventions.

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 lists of type `t` and a type declaration like `f :: a -> b -> c` stands for a function $f : a \times b \to c$. Haskell's ordered pairs of the form `(x,y)` aggregate elements of possibly distinct types `x` and `y`. They are useful also when one wants to return multiple results from a function.

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. Patterns match and deconstruct arguments on the left side of equations and build new terms of their corresponding date type definitions on the right side of equations.

The composition of functions `f` and `g` is denoted `f . g`.

The higher order function `map` is used to apply a function to all elements of a list and return the list of the results.

Haskell's *type classes* can be seen simply as dictionaries that associate to each polymorphic type specific implementations of functions.

As a small detail, occasionally used in our examples, the result of the last evaluation is stored in the special Haskell variable `it`.

Finally, the code in this paper is meant to be 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.