Computing with Catalan Families, Generically

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract. We describe arithmetic algorithms on a canonical number representation based on the Catalan family of combinatorial objects specified as a Haskell type class.
Our algorithms work on a *generic* representation that we illustrate on instances members of the Catalan family, like ordered binary and multiway trees. We validate the correctness of our algorithms by defining an instance of the same type class based the usual bitstring-based natural numbers.
While their average and worst case complexity is within constant factors of their traditional counterparts, our algorithms provide super-exponential gains for numbers corresponding to Catalan objects of low representation size.
Keywords:
tree-based numbering systems cross-validation with type classes, arithmetic with combinatorial objects, Catalan families, generic functional programming algorithms.

1 Introduction

This paper generalizes the results of [1] and [2], where special instances of the Catalan family of combinatorial objects (the language of balanced parentheses and the set ordered rooted trees with empty leaves, respectively) have been endowed with basic arithmetic operations corresponding to those on bitstring-represented natural numbers.

The main contribution of this paper is a *generic* Catalan family based numbering system that supports computations with numbers comparable in size with Knuth's "arrow-up" notation. 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.

As the Catalan family [3, 4] contains a large number of computationally isomorphic but structurally distinct combinatorial objects, we will describe our arithmetic computations generically, using Haskell's *type classes* [5], of which typical members of the Catalan family, like binary trees and multiway trees 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 cross-validate the correctness of our generically defined arithmetic operations.

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 3.4 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 describes arithmetic operations that favor operands of low representation complexity including computations with giant numbers. Section 6 concludes the paper.

We have adopted a *literate programming* style, i.e. the the code described in the paper forms a self-contained Haskell module (tested with ghc 7.8.3). It is available at `http://www.cse.unt.edu/~tarau/research/2014/GCat.hs`.

2 Related work

The first instance of a hereditary number system occurs in the proof of Goodstein's theorem [6], 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 a bijection. 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.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [8], covering operations like the *tetration* (a notation for towers of exponents). In contrast to 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.

While combinatorial enumeration and combinatorial generation, for which a vast literature exists (see for instance [3], [9] and [4]) can be seen as providing unary Peano arithmetic operations implicitly, we are not aware of any work, with the exception of [1] and [2] enabling arithmetic computations of efficiency comparable to the usual binary numbers (or better) using combinatorial families.

Providing a *generic mechanism* for efficient arithmetic computations with *arbitrary members of the Catalan family* 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 [3].

```
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)
```

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 [3].

```
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]

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 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 $2^i(j+1)-1$.*

[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`.

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 [10]) 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`. Note that $\nu_2(n)$ could also be computed slightly faster, by using Haskell's bit operations, as `n .&. (-n)`.

```
  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.

```
*GCat> c (100,200)
509595541291748219401674688561151
*GCat> c' it
(100,200)
*GCat> map c' [1..10]
[(0,0),(0,1),(1,0),(1,1),(0,2),(0,3),(2,0),(2,1),(0,4),(0,5)]
*GCat> 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 for three consecutive numbers. The order of the edges is marked with `0` and `1`.

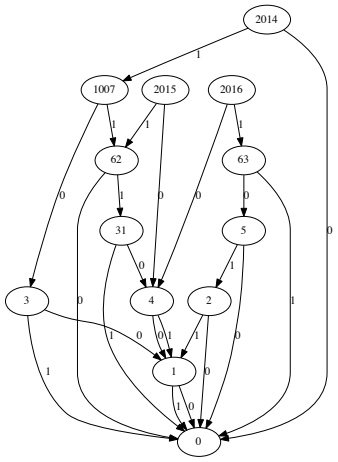

Fig. 1: DAG representing 2014, 2015 and 2016

3.5 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` and `M` as ranges, we will simply provide specialized type declarations for them:

```
n :: Cat a ⇒ a→N
n = view
```

```
t :: Cat a ⇒ a→T
t = view
```

```
m :: Cat a ⇒ a→M
m = view
```

The following examples illustrate the resulting specialized conversion functions:

```
*GCat> t 42
C E (C E (C E (C E (C E (C E E)))))
*GCat> m it
F [F [],F [],F [],F [],F [],F []]
*GCat> 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:

```
*GCat> to_list 2014
[0,3,0,4]
*GCat> 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.

```
*GCat> catShow 0
"()"
*GCat> catShow 1
"(())"
*GCat> catShow 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.

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
```

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 are based on arithmetic observations about the behavior of these blocks when incrementing or decrementing a binary number by `1`, derived from equation (5).

They first decompose their arguments using `c'`. Then, after transforming them as a result of adding or subtracting `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*. The main intuition is that as adding or subtracting `1` changes the parity of a number and as carry-ons propagate over a block of `1`s in the case of addition and over a block of `0`s in the case of subtraction, *blocks* of contiguous `0` and `1` digits will be flipped as a result of applying `s` or `s'`.

```
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`:

```
*GCat> map s [0..15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
*GCat> 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
*GCat> s it
C E (C E E)
*GCat> s it
C (C E E) E
*GCat> s it
C (C E E) (C E E)
*GCat> s it
C E (C E (C E E))
*GCat> s it
C E (C (C E E) E)
```

The function `nums` generates an initial segment of the "natural numbers" defined by an instance of `Cat`.

```
nums :: Cat a ⇒ a → [a]
nums x = f x [] where
  f x xs | e_ x = e:xs
  f x xs = f (s' x) (x:xs)
```

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 where effort 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'`.

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
```

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`:

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

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

```
*GCat> exp2 (exp2 (exp2 (exp2 (exp2 (exp2 (exp2 E))))))
C (C (C (C (C (C E E) E) E) E) E) (C E E)
*GCat> m it
F [F [F [F [F [F [F []]]]]],F []]
*GCat> log2 (log2 (log2 (log2 (log2 (log2 (log2 it))))))
F []
```

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. Another average constant time / worst case *log** algorithm is counting the trailing `0`s of a number (on instance `T`):

```
trailingZeros x | e_ x = e
trailingZeros x | odd_ x = e
trailingZeros x = s (fst (c' x))
```

Note that binary search (with O(log(n) worst case performance) needs to be used to count them with a bitstring representation.

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 by 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 [10] for a 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 0s and 1s

We are now ready to define 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 Conclusion

We have described through a type class mechanism an arithmetic system working on members of the Catalan family of combinatorial objects, that takes advantage of compact representations of some giant numbers and can perform interesting computations intractable with their bitstring-based counterparts.

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

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.

Acknowledgement

This research is supported by NSF research grant 1423324.

References

1. Tarau, P.: Computing with Catalan Families. In Dediu, A.H., Martin-Vide, C., Sierra, J.L., Truthe, B., eds.: Proceedings of Language and Automata Theory and Applications, 8th International Conference, LATA 2014, Madrid, Spain,, Springer, LNCS (March 2014) 564–576
2. Tarau, P.: The Arithmetic of Recursively Run-length Compressed Natural Numbers. In Ciobanu, G., Méry, D., eds.: Proceedings of 8th International Colloquium on Theoretical Aspects of Computing, ICTAC 2014, Bucharest, Romania, Springer, LNCS 8687 (September 2014) 406–423
3. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publ. Co., Belmont, CA, USA (1986)
4. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional (2006)
5. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL. (1989) 60–76
6. Goodstein, R.: On the restricted ordinal theorem. Journal of Symbolic Logic (9) (1944) 33–41
7. Knuth, D.E.: TCALC program (December 1994) `http://www-cs-faculty.stanford.edu/~uno/programs/tcalc.w.gz`.
8. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
9. Kreher, D.L., Stinson, D.: Combinatorial Algorithms: Generation, Enumeration, and Search. The CRC Press Series on Discrete Mathematics and its Applications. CRC PressINC (1999)
10. Tarau, P., Buckles, B.: Arithmetic Algorithms for Hereditarily Binary Natural Numbers. In: Proceedings of SAC'14, ACM Symposium on Applied Computing, PL track, Gyeongju, Korea, ACM (March 2014)