# Computing with Catalan Families

Paul Tarau

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

**Abstract.** We study novel arithmetic algorithms on a canonical number representation based on the Catalan family of combinatorial objects. For numbers corresponding to Catalan objects of low structural complexity our algorithms provide super-exponential gains while their average case complexity is within constant factors of their traditional counterparts.

**Keywords:** *hereditary numbering systems, arithmetic algorithms for Combinatorial objects, structural complexity of natural numbers, run-length compressed numbers, Catalan families*

## 1 Introduction

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" [1] or tetration are useful in describing very large numbers, they do not provide the ability to actually *compute* with them – as, for instance, addition or multiplication with a natural number results in a number that cannot be expressed with the notation anymore.

The novel 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 complexity that is comparable with the traditional binary numbers, while their best case complexity outperforms binary numbers by an arbitrary tower of exponents factor. Simple operations like successor, multiplication by 2, exponent of 2 are constant time and a number of other operations benefit from significant complexity reductions.

For the curious reader, it is basically a *hereditary number system* [2], based on recursively applied *run-length* compression of the usual binary digit notation. To evaluate best and worst cases, a concept of structural complexity is introduced, based on the size of representations and algorithms favoring large numbers of small structural complexity are designed for arithmetic operations.

We have adopted a *literate programming* style, i.e. the code described in the paper forms a self-contained Haskell module (tested with ghc 7.6.3), also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/catco.hs` . 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 introduces recursively run-length compressed natural numbers seen as a member of the Catalan family of combinatorial objects. Section 3 describes constant time successor and predecessor operations on our numbers. Section 4 describes novel algorithms for arithmetic operations taking advantage of our number representation. Section 5 defines a concept of structural complexity and studies best and worst cases. Section 6 discusses related work. Section 7 concludes the paper and discusses future work.

## 2 Recursively run-length compressed natural numbers as objects of the Catalan family

The Catalan family of combinatorial objects [3] spans over a wide diversity of concrete representation ranging from balances parenthesis expressions and rooted plane trees to non-crossing partitions and polygon triangulations.

### 2.1 The "cons-list"-view of Catalan objects

For simplicity, we will pick here as a representative of the Catalan family a language of balanced parentheses defined as follows.

We fix our set of two parentheses {`L`,`R`} as specified by the Haskell data type `Par`.

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

The set of Dyck words is an important member of the Catalan family of combinatorial objects.

**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.*

Let $\mathbb{T}$ be the language obtained from the set Dick 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 $\mathbb{T}$ in Haskell as the type `T` and we will call its members *terms*.

```
type T = [Par]
```

It is convenient to view $\mathbb{T}$ as the set of *rooted ordered binary trees* through the operations `cons` and `decons` defined as:

```
cons :: (T,T) → T
cons (xs,L:ys) = L:xs++ys
```

```
decons :: T→(T,T)
decons (L:ps) = count_pars 0 ps where
  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
```

$\mathbb{T}$ can also be seen as isomorphic with the *set of ordered rooted trees*, another member of the Catalan family. The forest of subtrees corresponds to the toplevel balanced parentheses composing an element of $\mathbb{T}$ as defined by the bijections `to_list` and `from_list`.

```
to_list :: T → [T]
to_list [L,R] = []
to_list ps = hs:hss where
  (hs,ts) = decons ps
  hss = to_list ts
```

We will call *subterms* the terms extracted by `to_list`.

```
from_list :: [T]→T
from_list [] = [L,R]
from_list (hs:hss) = cons (hs,from_list hss)
```

## 2.2 The Catalan encoding of natural numbers

We are ready for an arithmetic interpretation of the language $\mathbb{T}$, associating a unique natural number to each of its terms $t$:

- The term $t$=`[L,R]` corresponds to zero
- if `xs` is obtained by applying the `to_list` operation to $t$, then each `x` on the list `xs` counts the number of $b \in \{0,1\}$ digits, followed by *alternating* counts of `1-b` and `b` digits, with the conventions that the most significant digit is 1 and the counter `x` represents `x+1` objects.

– the same principle is applied recursively for the counters, until [L,R] is reached.

One can see this process as run-length compressed base-2 numbers, unfolded as an object of the Catalan family, after applying the encoding recursively.

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

**Proposition 1** *If the length of* xs = to_list x *is odd, then* x *encodes an odd number, otherwise it encodes an even number.*

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

This ensures the correctness of the Haskell definitions of the predicates odd_ and even_, the last one defined to be true for terms different from [L,R] only.

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

odd_ :: T→Bool
odd_ x = oddLen (to_list x)

even_ :: T→Bool
even_ x =  f (to_list x) where
  f [] =False
  f (y:ys) = oddLen ys
```

*Note that while these predicates work in time proportional to the length of the list representing a term in* $\mathbb{T}$*,* with a (dynamic) array-based list representation that keeps track of the length or keeps track of the parity bit explicitly, so one can assume that they can be made constant time with an optimal data structure choice*, as we will do in the rest of the paper, while focusing, for simplicity, on the language of balanced parenthesis* $\mathbb{T}$*.*

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

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

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

```
type N = Integer

n :: T→N
n ([L,R]) = 0
n a | even_ a = 2^(n x + 1)*(n xs) where (x,xs) = decons a
n a | odd_ a = 2^(n x + 1)*(n xs+1)-1 where (x,xs) = decons a
```

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

```
0: [L,R]
1: [L,L,R,R]
2: [L,L,R,L,R,R]
3: [L,L,L,R,R,R]
4: [L,L,L,R,R,L,R,R]
5: [L,L,R,L,R,L,R,R]
```

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

```
t :: N→T
t 0 = [L,R]
t k | k>0 = zs where
  (x,y) = if even k then split0 k else split1 k
  ys = t y
  zs = if x==0 then ys else cons (t (x-1),ys)
```

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

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

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

They return a pair `(x,y)` consisting of a count `x` of the number of digits in the block, and the natural number `y` representing the digits left over after extracting the block. Note that `div`, occurring in both functions, is integer division.

The following holds:

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

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

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

Figure 1 shows the DAG obtained by folding together identical subterms at each level for the term corresponding to the natural number `12345`, where we have mapped lists of `L` symbols to strings built of `'('` and `R` to `')'` characters, for readability. Note that integer labels mark the order of the edges outgoing from a vertex.

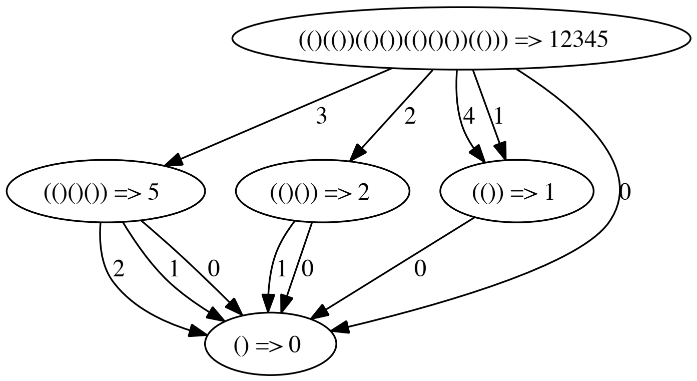

Fig. 1: The DAG illustrating the term associated to `12345`

The constants `e` and `u` correspond to the natural numbers `0` and `1`. The predicates `e_` and `u_` are used to recognize them.

```
e = [L,R]
u = [L,L,R,R]

e_ [L,R] = True
e_ _ = False

u_ [L,L,R,R] = True
u_ _ = False
```

## 3 Successor (`s`) and predecessor (`s'`)

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

```
s x | e_ x = u -- 1
s x | even_ x = from_list (sEven (to_list x)) -- 7
s x | odd_ x = from_list (sOdd (to_list x)) -- 8

sEven (a:x:xs) |e_ a = s x:xs -- 3
sEven (x:xs) = e:s' x:xs -- 4

sOdd [x]= [x,e] -- 2
sOdd (x:a:y:xs) | e_ a = x:s y:xs -- 5
sOdd (x:y:xs) = x:e:(s' y):xs -- 6
```

```
s' x | u_ x = e -- 1
s' x | even_ x = from_list (sEven' (to_list x)) -- 8
s' x | odd_ x = from_list (sOdd' (to_list x)) -- 7

sEven' [x,y] |e_ y = [x] -- 2
sEven' (x:b:y:xs) | e_ b = x:s y:xs -- 6
sEven' (x:y:xs) = x:e:s' y:xs -- 5

sOdd' (b:x:xs) | e_ b = s x:xs -- 4
sOdd' (x:xs) = e:s' x:xs -- 3
```

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

**Proposition 3** *Denote* $\mathbb{T}^+ = \mathbb{T} - \{\mathtt{e}\}$*. The functions* $s : \mathbb{T} \to \mathbb{T}^+$ *and* $s' : \mathbb{T}^+ \to \mathbb{T}$ *are inverses.*

*Proof.* It follows by structural induction after observing that patterns for 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 $< \mathbb{T}, e, s >$ is a *Peano algebra.*

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

**Proposition 4** `s` *and* `s'` *are constant time, on the average.*

*Proof.* Observe that the average size of a contiguous block of `0`s or `1`s in a number of bitsize $n$ has the upper bound 2 as $\sum_{k=0}^{n} \frac{1}{2^k} = 2 - \frac{1}{2^n} < 2$. As on 2-bit numbers we have an average of $\frac{0+0+0+1}{4} = 0.25$ 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 Arithmetic operations

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

### 4.1 A few other constant time operations

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

```
db x | e_ x = e
db xs | odd_ xs = cons (e,xs)
db xxs | even_ xxs = cons (s x,xs) where (x,xs) = decons xxs
```

```
hf x |e_ x = e
hf xxs = if e_ x then xs else cons (s' x,xs) where  (x,xs) = decons xxs
```

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

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

```
exp2 x | e_ x = u
exp2 x = from_list [s' x,e]
```

as it can be derived, for $k = 0$, from the identity

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

**Proposition 5** *The operations* `db,hf` *and* `exp2` *are constant time, on the average.*

*Proof.* As `s,s'` are average constant time, the proposition follows by observing that at most 1 call to `s,s'` is made in each definition.

Due to space constraints we will just mention that algorithms favoring numbers with large contiguous blocks of 0s and 1s in their binary representations can be devised for various arithmetic operations. For instance, addition of odd numbers, would benefit from the use of identity 4.

$$(\lambda x.2x + 1)^k(x) + (\lambda x.2x + 1)^k(y) = (\lambda x.2x + 2)^k(x + y) \qquad (4)$$

## 5 Structural complexity

Arguments similar to those about the average behavior of `s` and `s'` can be carried out to prove that *the average complexity of other arithmetic operations matches their traditional counterparts*, using the fact, shown in the proof of Prop. 3, that the average size of a block of contiguous `0` or `1` bits is at most 2.

To evaluate the best and worst case space requirements of our number representation, after defining the bitsize of a term as

```
bitsize x = sum (map (n.s) (to_list x))
```

we introduce here a measure of *structural complexity*, defined by the function `tsize` that counts the nodes of a term of type $\mathbb{T}$ (except the root).

```
tsize x =foldr add1 0 (map tsize xs) where
  xs = to_list x
  add1 x y = x + y +1
```

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

$$c(t) = \begin{cases} 0 & \text{if } \mathtt{t} = \mathrm{e}, \\ \sum_{x \in \mathrm{xs}} (1 + c(x)) & \text{if } \mathrm{xs} = \mathtt{to\_list}\ \mathrm{t}. \end{cases} \tag{5}$$

The following holds:

**Proposition 6** *For all terms* $t \in \mathbb{T}$, `tsize t` $\leq$ `bitsize t`.

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

The following example illustrates their use:

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

Figure 2 shows the reductions in structural complexity compared with bitsize for an initial interval of $\mathbb{N}$.

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

```
iterated f a x |e_ a = x
iterated f k x = f (iterated f (s' k) x)
```

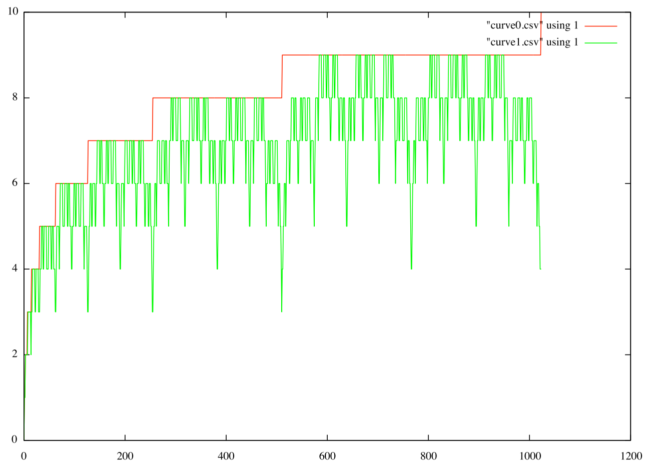

Fig. 2: Structural complexity (yellow line) bounded by bitsize (red line) from 0 to $2^{10} - 1$

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

```
bestCase k = iterated wterm k e where wterm x = cons (x,e)
```

and a worst case

```
worstCase k = iterated (s.db.db) k e
```

The following examples illustrate these functions:

```
*CatCo>  bestCase (t 4)
[L,L,L,L,L,R,R,R,R,R]
*CatCo> n it
65535
*CatCo> bitsize (bestCase (t 4))
16
*CatCo> tsize (bestCase (t 4))
4

*CatCo> worstCase (t 4)
[L,L,R,L,R,L,R,L,R,L,R,L,R,L,R,R]
*CatCo> n it
85
*CatCo> bitsize (worstCase (t 4))
7
*CatCo> tsize (worstCase (t 4))
7
```

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

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

The average space-complexity of the representation is related to the average length of the *integer compositions* of the bitsize of a number. Intuitively, the shorter the composition in alternative blocks of `0` and `1` digits, the more significant the compression is.

## 6 Related work

Several notations for very large numbers have been invented in the past. Examples include Knuth's *arrow-up* notation [1], covering operations like the *tetration* (a notation for towers of exponents). In contrast to our approach, such notations are not closed under arithmetic operations, they cannot be used as a replacement for ordinary binary or decimal numbers.

The first instance of a hereditary number system, at our best knowledge, occurs in the proof of Goodstein's theorem [2], where replacement of finite numbers on a tree's branches by the ordinal $\omega$ allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates. Another hereditary number system is Knuth's TCALC program [4] 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. In [5] a similar (non-canonical) exponential-based notation called "integer decision diagrams" is introduced, providing a compressed representation for sparse integers, sets and various other data types.

This paper is an adaptation of our online draft at the Cornell `arxiv` repository [6], which describes a more complex hereditary number system (based on run-length encoded "bijective base 2" numbers, first introduced in [7] pp. 90-92 as "m-adic" numbers). In contrast to [6], we are using here the familiar binary number system, and we represent our numbers as lists of balanced parentheses rather than the more complex data structure used in [6].

Arithmetic computations based on a member of the Catalan family (ordered rooted of binary trees) are described in [8]. In [9] a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite functions. However likewise [8] and [9], and by contrast to those proposed in this paper, they only compress "sparse" numbers, consisting of relatively few `1` bits in their binary representation.

## 7 Conclusion

We have provided in the form of a literate Haskell program a specification of a number system based on a member of the Catalan family of combinatorial objects.

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

Our novel number representation enables performing arithmetic operations with members of the Catalan family of combinatorial numbers and can deal with numbers significantly larger than traditional bitstring representations.

## Acknowledgement

This research has been supported by NSF research grant 1018172.

## References

1. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
2. Goodstein, R.: On the restricted ordinal theorem. Journal of Symbolic Logic (9) (1944) 33–41
3. Sloane, N.J.A.: A000108, The On-Line Encyclopedia of Integer Sequences. (2006) published electronically at www.research.att.com/∼njas/sequences.
4. Knuth, D.E.: TCALC program (December 1994)
5. Vuillemin, J.: Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates. In: Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on. (june 2009) 7 –14
6. Tarau, P.: Arithmetic Algorithms for Hereditarily Binary Natural Numbers (June 2013) http://arxiv.org/abs/1306.1128.
7. Salomaa, A.: Formal Languages. Academic Press, New York (1973)
8. Tarau, P., Haraburda, D.: On Computing with Types. In: Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track, Riva del Garda (Trento), Italy (March 2012) 1889–1896
9. Tarau, P.: Declarative modeling of finite mathematics. In: PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, New York, NY, USA, ACM (2010) 131–142