Computing with Catalan Families

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

LATA'2014

Motivation

- traditional number representation:
  - binary, decimal, base-N number arithmetics provide an exponential improvement over unary "caveman's" notation
  - quite resilient, staying fundamentally the same for the last 1000 years
  - computations are limited by the size of the operands or results
  - egalitarian: all numbers are treated the same way
  - little effort to take advantage of the structural uniformity of the operands, when present
  - crashes quickly under heavy use of exponentials, e.g, towers of exponents
- $\Rightarrow$ this paper is about how we can we do better if, in an alternative numbering system, based on Catalan families, representation size of the operands can be much smaller than their bitsizes
- we propose an elitist representation: some numbers are treated more favorably, while others "suffer" by a constant factor

"All animals are equal, but some animals are more equal than others."
George Orwell, Animal Farm

Outline

Some context

- the first instance of a *hereditary number system* occurs in the proof of Goodstein's theorem (exponents are expanded recursively) – "hailstone sequences reach 0" – "Hercules and hydra" game
- notations for very large numbers have been invented in the past, all non-canonical (multiple representations for the same number)
  - Knuth's *up-arrow* notation covering operations like the *tetration* (a notation for towers of exponents)
  - Knuth's TCALC program that decomposes $n = 2^a + b$ with $0 \leq b < 2^a$ and then recurses on a and b with the same decomposition
  - Vuillemin uses a similar exponential-based notation called "integer decision diagrams", providing a compressed representation for sparse integers, sets and various other data types
- the question we want answer: are there canonical and hereditary number representations that can represent very large numbers and are closed under arithmetic operations ?

Notations for vs. computations with giant numbers

- *notations* like Knuth's "up-arrow" are useful in describing very large numbers
- but they do not provide the ability to actually *compute* with them – as addition or multiplication results in a number that cannot be expressed with the notation
- the novel contribution of this paper is a a Catalan family-based canonical numbering system that *allows computations* with numbers comparable in size with Knuth's "up-arrow" notation
- these computations have average and worst case complexity that is comparable with the traditional binary numbers
- their best case complexity outperforms binary numbers by an arbitrary tower of exponents factor
- $\Rightarrow$ a *hereditary number system* based on recursively applied *run-length* compression of the usual binary digit notation
- $\Rightarrow$ a concept of *structural complexity* is introduced, that serves as an indicator of the expected performance of our arithmetic operations

A member of the Catalan family: Dyck words

The Catalan family of combinatorial objects spans over a wide diversity of concrete representation ranging from balanced parentheses expressions and rooted plane trees to non-crossing partitions and polygon triangulations

Definition

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 of 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.
$\Rightarrow$ words in $\mathbb{T}$ are self-delimiting (actually also "bifix-free")

We represent the language $\mathbb{T}$ in Haskell as the type T and we will call its members *terms*.

```
data Par = L | R deriving (Eq,Show,Read)
type T = [Par]
```

The "cons-list"-view

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
```

The *ordered rooted tree* view

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)
```

For complexity analysis we can assume that an ordered rooted tree data structure is used for the language $\mathbb{T}$, under which the `from_list` *and* `to_list` *operations are constant time.*

The arithmetic interpretation of Catalan objects

- 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.
- by convention, as the last (and most significant) digit is `1`, the last count on the list `xs` is for `1` digits

Recognizing odd and even

The following simple fact allows inferring parity from the number of subterms of a term.

Proposition

If the length of $\mathtt{xs} = \mathtt{to\_list\ \ x}$ *is odd, then* $\mathtt{x}$ *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 defined true for terms different from `[L,R]` only.

Computing the function $n : \mathbb{T} \to \mathbb{N}$

Definition

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

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

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

For complexity analysis we can assume that length information is stored, and consequently the `odd_` and `even_` operations are constant time.

The bijection between $\mathbb{T}$ and $\mathbb{N}$

Proposition

$n : \mathbb{T} \rightarrow \mathbb{N}$ is a bijection, i.e., each term canonically represents the corresponding natural number.

See explicitly computed inverse $t : \mathbb{T} \rightarrow \mathbb{N}$ in the paper.

```
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]
```

A DAG representation of our numbers

- the DAG is obtained by folding together identical subterms at each level
- we map "L" and "R" to "(" and ")", for readability
- integer labels mark the order of the edges outgoing from a vertex

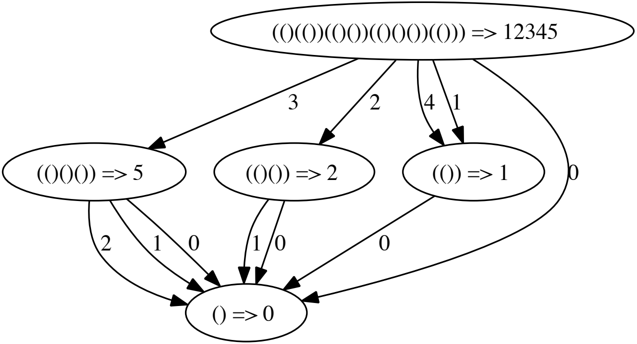

Figure : The DAG illustrating the term associated to 12345

Successor

```
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
```

Note that `e_` recognizes `e=[L,R]`, `u=[L,L,R,R] represents 1`, `u_` recognizes `u` and `s'` is the (mutually recursive) predecessor.

Predecessor

```
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
```

- `s` and `s'` are mutually recursive.
- each call to `s` and `s'` in `s` and `s'` is on a term corresponding to a (much) smaller natural number

s and s′ are inverses

Proposition

Denote $\mathbb{T}^+ = \mathbb{T} - \{\mathrm{e}\}$. *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*.

Complexity of successor and predecessor

- recursive calls to `s, s'` in `s, s'` happen on terms that are logarithmic in the bitsize of their operands ⇒ worst case time complexity of `s` and `s'` is the given by the iterated logarithm (log^*) of their arguments
- average size of a block is 2 bits (see paper for proof) ⇒ average time complexity of `s` is constant
- experimentally: when computing successor on the first $2^{30} = 1073741824$ natural numbers, there are in total 2381889348 calls to `s`, averaging to 2.2183 per successor and predecessor computation

A few low complexity operations

double and half

```
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
```

power of 2

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

Proposition

The costs of `db, hf` *and* `exp2` *are within a constant factor from the cost of* `s, s'` $\Rightarrow$ *log^* worst case and constant on the average.*

What else we can compute with efficiency comparable to binary arithmetic?

- any enumeration on Catalan families can be seen as a Peano algebra, so why is ours special?
- $\Rightarrow$ with constant average time for double and half we can do binary arithmetic efficiently!
- $\Rightarrow$ we can also do better – various arithmetic operations on an equivalent ordered rooted tree representation that work with effort proportional to our Catalan objects' representation size, rather than their bitsize at `http://logic.cse.unt.edu/tarau/Research/2013/rrl.pdf`

Computing representation sizes

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

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

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

$$c(t) = \begin{cases} 0 & \text{if } \mathtt{t} = \mathrm{e}, \\ \sum_{x \in \mathtt{xs}} (1 + c(x)) & \text{if xs = } \mathtt{to\_list}\ \text{t.} \end{cases} \quad (2)$$

Proposition

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

"Structural complexity" as representation size

- for operations like `s, s', db, hf, exp2` worst case effort is proportional to the depth of the tree
- but the depth of the tree is proportional to the height of the corresponding tower of exponents
- for operations like addition, subtraction, comparison, the worst case is proportional with the tree size of the smallest operand (not shown in the paper) but see `http://logic.cse.unt.edu/tarau/Research/2013/rrl.pdf` where these operations are implemented with an ordered rooted binary tree data structure
- so each time when "structural complexity" is $<$ than bitsize we gain,
- but as it is always $\leq$, we never loose
- in the best case, we gain by an arbitrary tower of exponents factor

Best and worst case

best case: bitsize much larger than structural complexity

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

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

worst case: bitsize the same as structural complexity

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

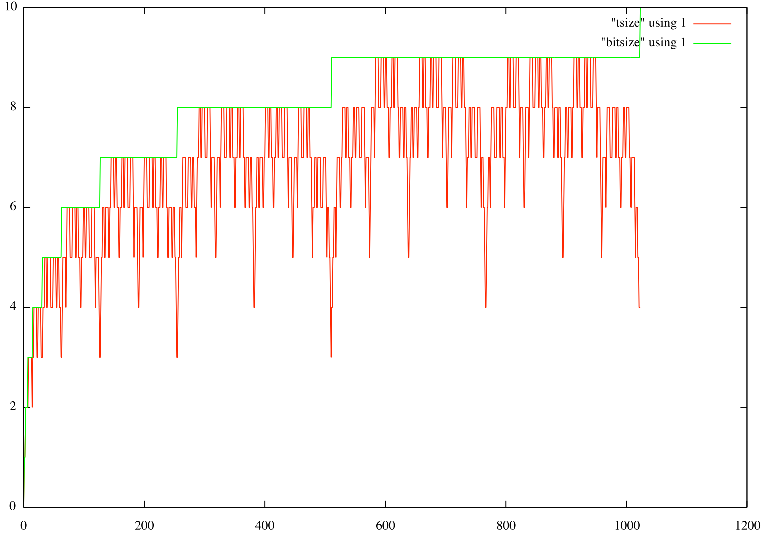

Figure : Structural complexity (red line) vs. bitsize (green line) from 0 to $2^{10} - 1$

Conclusion

- *arithmetic computations* using Catalan families instead of bitstrings can be performed with in constant time or time proportional to their *structural complexity* rather than their *bitsize*
- bidirectional self-delimiting representation ⇒ it makes easier correcting transmission errors
- our structural complexity is a weak approximation of Kolmogorov complexity
- ⇒ random instances are closer to the worst case than the best case
- still, *best cases are important* - humans in the random universe are a good example for that :-)
- Haskell code at `http://logic.cse.unt.edu/tarau/research/2013/catco.hs`
- code with ordered rooted trees, complete set of operations: at: `http://logic.cse.unt.edu/tarau/research/2013/rrl.hs`