The Arithmetic of Recursively Run-length Compressed Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

ICTAC'2014

Motivation

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

- 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
  - does not take advantage of the structural uniformity of the operands
  - crashes quickly under heavy use of exponentials, e.g, towers of exponents
- $\Rightarrow$ this paper is about how we can we do better when the representation size of the operands is much smaller than their bitsizes
- we propose an elitist representation: some numbers are treated more favorably, while others "suffer" by a constant factor

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 tree-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 *representation complexity* is introduced, that serves as an indicator of the expected performance of our arithmetic operations

Ordered rooted trees with empty leaves: the data type of Recursively Run-length Compressed Natural Numbers

- the paper is a *literate Haskell program*
- a minimal subset, seen as an executable notation for functions

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

- the term `F []` (empty leaf) corresponds to zero
- in the term `F xs`, each `x` $\in$ `xs` counts the number `x+1` of $b \in \{0,1\}$ digits, followed by *alternating* counts of `1-b` and `b` digits
- the same principle is applied recursively for the counters

ex: `123` as the (big-endian) binary number `1101111` is `[1,0,3]`

- run-length compressed base-2 numbers are unfolded as trees with empty leaves, after applying the encoding *recursively*
- note that we count `x+1` as we start at `0`.
- by convention the last count on the list `xs` is for `1` digits

Recognizing odd and even

Can we infer parity from the number of subterms of a term?

Proposition

If the length of `xs` $=$ *in* `Fx x` *is odd, then* `x` *encodes an odd number, otherwise it encodes an even number.*

Proof.

If 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. □

- $\Rightarrow$ correctness of the definitions of the predicates `odd_` and `even_`
- we can assume that *length* information is stored
- $\Rightarrow$ the `odd_` and `even_` operations are constant time

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

A natural number n in base 2, (big-endian):

$$n = b_0^{k_0} b_1^{k_1} \dots b_i^{k_i} \dots b_m^{k_m} \text{ with } b \in \{0,1\} \tag{1}$$

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

Definition

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

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

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

Proposition

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

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

```
0: F []
1: F [F []])
2: F [F [],F []]
3: F [F [F []]]
4: F [F [F []],F []]
5: F [F [],F [],F []]
...
```

A DAG representation of our numbers

- the DAG is obtained by folding together identical subterms at each level
- we remove the $\mathtt{F}$ symbols and map "$\mathtt{[}$" and "$\mathtt{]}$" to "$\mathtt{(}$" and "$\mathtt{)}$" $\Rightarrow$
- our trees are an instance of the *Catalan family of combinatorial objects*
- integer labels mark the order of the edges outgoing from a vertex

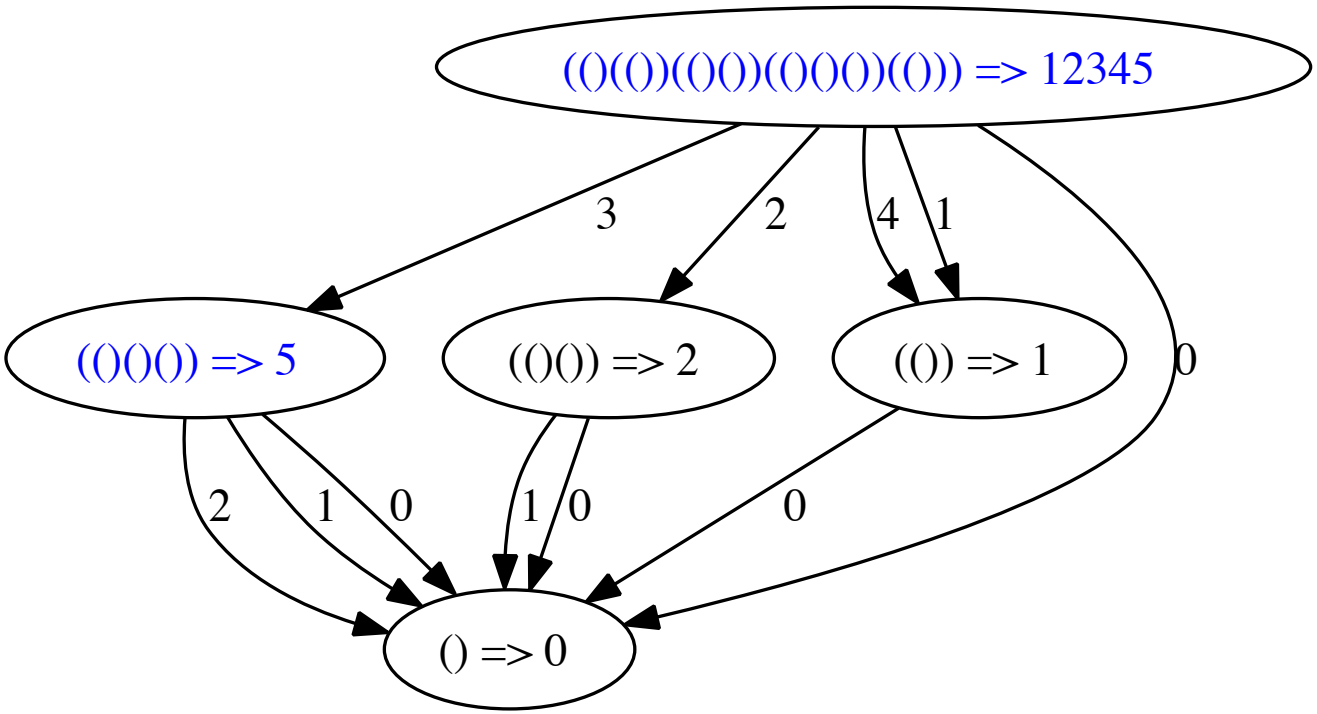

Figure: The DAG illustrating the term associated to 12345

Successor

- derived directly from the definition of $n : \mathbb{T} \rightarrow \mathbb{N}$

```
s :: T → T
s (F []) = F [F []] -- 1
s (F [x]) = F [x,F []] -- 2

s a@(F (F []:x:xs)) | even_ a = F (s x:xs) -- 3
s a@(F (x:xs)) | even_ a = F (F []:s' x:xs) -- 4

s a@(F (x:F []:y:xs)) | odd_ a = F (x:s y:xs) -- 5
s a@(F (x:y:xs)) | odd_ a = F (x:F []:(s' y):xs) -- 6
```

Haskell note: the pattern `a@p` indicates that the parameter `a` has the same value as its expanded version matching the patten `p`

Predecessor

```
s' :: T → T
s' (F [F []]) = F [] -- 1
s' (F [x,F []]) = F [x] -- 2

s' b@(F (x:F []:y:xs)) | even_ b = F (x:s y:xs) -- 6
s' b@(F (x:y:xs)) | even_ b = F (x:F []:s' y:xs) -- 5

s' b@(F (F []:x:xs)) | odd_ b = F (s x:xs) -- 4
s' b@(F (x:xs)) | odd_ b = F (F []:s' x:xs) -- 3
```

- `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

$\mathtt{s}$ and $\mathtt{s'}$ are inverses

Proposition

Denote $e = \mathtt{F}\ \ [\,]$, $\mathbb{T}^+ = \mathbb{T} - \{\mathtt{e}\}$. *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 $-\ \ \mathtt{k}$ in $\mathtt{s}$ correspond one by one to patterns marked by $-\ \ \mathtt{k}$ in $\mathtt{s'}$ 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 $\Rightarrow$ worst case time complexity of `s` and `s'` is the given by the iterated logarithm (log^*) of their arguments
- successor and predecessor are only *log*, ex: $s\ 1111\ldots11 = 10000\ldots0$
- average size of a block is 2 bits (see proof of Prop. 5 in the paper) $\Rightarrow$ 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

Constant average and log^* worst case complexity operations

double and half

```
db (F []) = F []
db a@(F xs) | odd_ a = F (F []:xs)
db a@(F (x:xs)) | even_ a = F (s x:xs)

hf (F []) = F []
hf (F (F []:xs)) = F xs
hf (F (x:xs)) = F (s' x:xs)
```

power of 2

```
exp2 (F []) = F [F []]
exp2 x = F [s' x,F []]
```

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.*

Proof

Can we compute with efficiency comparable to binary arithmetic?

- any enumeration on combinatorial objects (trees in particular) 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!
- the arithmetic operations in the paper show that:
  1. we match binary arithmetic operations by a constant factor
  2. we can do (super)-exponentially better on some interesting giant numbers with small representation size
- a critical step: fast multiplication by an exponent of 2 $\Rightarrow$

Multiplication by an exponent of 2

- the function `leftshiftBy` implements $\lambda x.\lambda k.2^x k$

```
leftshiftBy :: T → T → T
leftshiftBy (F []) k = k
leftshiftBy _ (F []) = F []
leftshiftBy x k@(F xs) | odd_ k = F ((s' x):xs)
leftshiftBy x k@(F (y:xs)) | even_ k = F (add x y:xs)
```

- note the use of addition `add` (on terms of size $log_2(k)$)

```
leftshiftBy' x k = s' (leftshiftBy x (s k))

leftshiftBy'' x k = s' (s' (leftshiftBy x (s (s k))))
```

- the last two are derived from the identities

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

$$(\lambda x.2x+2)^n(k) = 2^n(k+2)-2 \quad (4)$$

The chain of mutually recursive arithmetic functions

- *leftshiftBy* :: $\mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
- *add*, *sub* :: $\mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
- *cmp* :: $\mathbb{T} \rightarrow \mathbb{T} \rightarrow$ `Ord=LT,EQ,GT`
- *bitsize* :: $\mathbb{T} \rightarrow \mathbb{T}$

- details of the code in the paper
- they progress linearly on the number of blocks or on smaller (log_2) terms

- other arithmetic operations - details of code in the paper
  - *mul*, *power*, *divide*, *reminder* :: $\mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
  - *square* :: $\mathbb{T} \rightarrow \mathbb{T}$

Complexity as representation size

```
tsize :: T → T
tsize (F xs) = foldr add1 (F []) (map tsize xs) where
    add1 x y = s (add x y)
```

- it counts the non-leaf nodes of a tree of type $\mathbb{T}$
- `tsize` corresponds to the function $c : \mathbb{T} \to \mathbb{N}$ defined as follows:

$$c(t) = \begin{cases} 0 & \text{if } \mathtt{t} = \mathtt{F}\ [], \\ \sum_{x \in \mathtt{xs}} (1 + c(x)) & \text{if } \mathtt{t} = \mathtt{F}\ \ \mathtt{xs}. \end{cases} \tag{5}$$

Proposition

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

Note: if leaves were added, it would be within a constant factor of `bitsize`.

‘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 term sizes of the operands
- so each time when “representation complexity” is $<$ than bitsize we gain,
- but as it is always $\leq$, we never loose by more than a small constant factor
- 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

```
F [F [F [F [F []]]]]
> n it
65535
> (n (bitsize (bestCase (t 4))),n (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)
F [F [],F [],F [],F []]
> n it
10
> (n (bitsize (worstCase (t 4))),n (tsize (worstCase (t 4))))
(4,4)
```

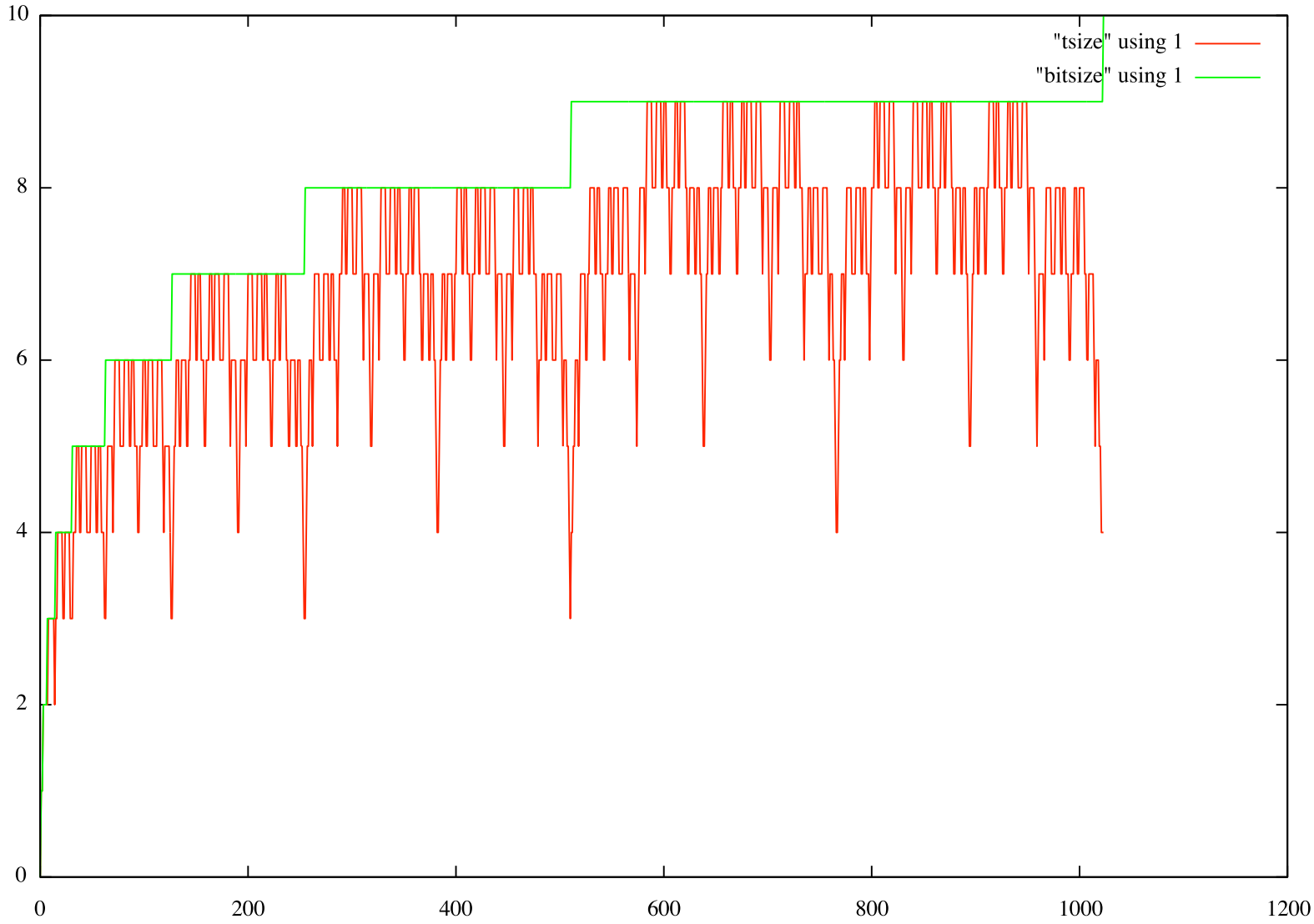

Figure: Structural complexity (red line) vs. bitsize (green line) from 0 to $2^{10} - 1$

A concept of duality

- our ordered rooted *trees* with empty leaves are members of the Catalan family of combinatorial objects
- $\Rightarrow$ they can be seen as ordered rooted *binary trees* with empty leaves, as defined by the bijection `toBinView` and its inverse `fromBinView`

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

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

A concept of duality is defined by *flipping* left and right branches:

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

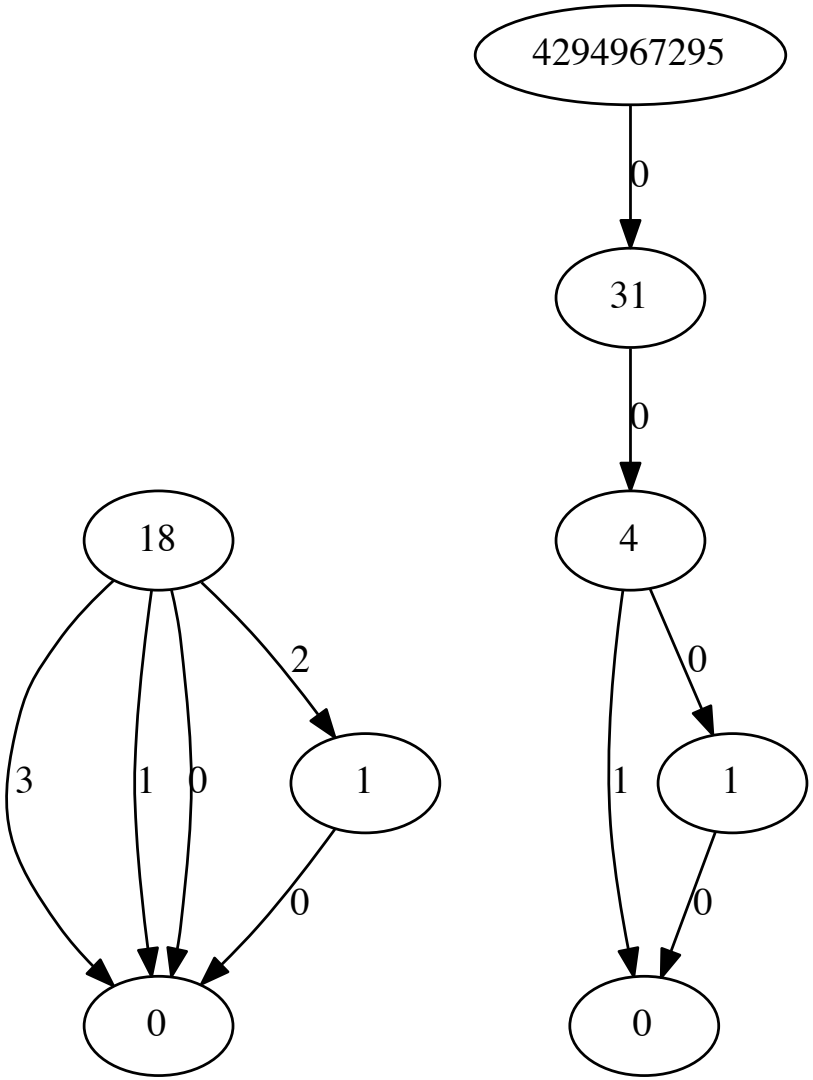

Figure: Duals, with trees folded to DAGs: *t 18* and *dual (t 18)*

`dual`: a bijection between numbers with high and low Kolmogorov complexity

Computing with towers of exponents: the Collatz conjecture

- something one cannot do with traditional arbitrary bitsize integers is to explore the behavior of interesting conjectures in the "new world" of numbers limited not by their sizes but by their representation complexity

The Collatz conjecture states that the function:

$$collatz(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases} \tag{6}$$

reaches 1 after a finite number of iterations.

- variant in the paper: the Syracuse function, starting from a tower of exponents 100 tall:

```
> take 100 (map(n.tsize) (nsyr (bestCase (t 100))))
[100,199,297,298,300,...,440,436,429,434,445,439]
```

Conclusion

- we have specified a new, tree-based number system where trees are built by recursively applying run-length encoding on the usual binary representation until the empty leaves corresponding to `0` are reached
- *arithmetic computations* like addition, subtraction, multiplication, bitsize, exponent of 2, that favor giant numbers with *low representation complexity*, are performed in constant time, or time proportional to their representation complexity
- our representation complexity is a weak approximation of Kolmogorov complexity
- $\Rightarrow$ 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://www.cse.unt.edu/~tarau/research/2014/RRL.hs`

Questions?