A Prolog Specification of Giant Number Arithmetic

Paul Tarau[1]

[1] Department of Computer Science and Engineering
University of North Texas

CICLOPS 2013

Motivation

- binary, decimal, base-N number arithmetics provide an exponential improvement over unary “caveman’s” notation
- they turned out to be quite quite resilient, staying fundamentally the same for the last 1000 years
- still, binary arithmetic makes little effort to take advantage of the “structural uniformity” of the operands, when present
- computations are limited by the size of the operands or results
- $\Rightarrow$ this paper is about how we can we do better if the “structural complexity” of the operands is much smaller than their bitsizes
- the new limit will be closer to the minimal computational effort an omniscient agent would spend on performing the arithmetic operations

Outline

Notations for vs. computations with giant numbers

- *notations* like Knuth's "up-arrow" or tetration 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 numbering system that *allows computations* with numbers comparable in size with Knuth's "arrow-up" notation
- these computations have a 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 a special (bijective) binary digit notation
- $\Rightarrow$ a concept of *structural complexity* is introduced, that serves as an indicator of the expected performance of our arithmetic operations

Bijective base-2 numbers as iterated function applications

- Natural numbers can be seen as represented by iterated applications of the functions $o(x) = 2x + 1$ and $i(x) = 2x + 2$ corresponding the so called *bijective base-2* representation together with the convention that 0 is represented as the empty sequence.
- $0 = \varepsilon$,
- $1 = o(\varepsilon)$,
- $2 = i(\varepsilon)$,
- $3 = o(o(\varepsilon))$,
- $4 = i(o(\varepsilon))$,
- $5 = o(i(\varepsilon))$

Properties of the iterated functions o^n and i^n

Proposition

Let f^n denote application of function f n times. Let $o(x) = 2x+1$ and $i(x) = 2x+2$, $s(x) = x+1$ and $s'(x) = x-1$. Then $k > 0 \Rightarrow s(o^n(s'(k)) = k2^n$ and $k > 1 \Rightarrow s(s(i^n(s'(s'(k)))) = k2^n$. In particular, $s(o^n(0)) = 2^n$ and $s(s(i^n(0))) = 2^{n+1}$.

Proof.

By induction. Observe that for $n = 0, k > 0, s(o^0(s'(k)) = k2^0$ because $s(s'(k))) = k$. Suppose that $P(n) : k > 0 \Rightarrow s(o^n(s'(k))) = k2^n$ holds. Then, assuming $k > 0$, P(n+1) follows, given that $s(o^{n+1}(s'(k))) = s(o^n(o(s'(k)))) = s(o^n(s'(2k))) = 2k2^n = k2^{n+1}$. Similarly, the second part of the proposition also follows by induction on n. $\square$

Some useful identities

$$o^n(k) = 2^n(k+1) - 1 \tag{1}$$

$$i^n(k) = 2^n(k+2) - 2 \tag{2}$$

and in particular

$$o^n(0) = 2^n - 1 \tag{3}$$

$$i^n(0) = 2^{n+1} - 2 \tag{4}$$

Hereditarily binary numbers

Definition

The data type $\mathbb{T}$ of the set of hereditarily binary numbers is defined inductively as the set of Prolog terms such that:
$X \in \mathbb{T}$ if and only if $X = e$ or X is of the form $v(T, Ts)$ or $w(T, Ts)$ where $T \in \mathbb{T}$ and Ts stands for a finite sequence (list) of elements of $\mathbb{T}$.

- The term e (empty leaf) corresponds to zero
- the term $v(T, Ts)$ counts the number $T + 1$ (as counting starts at 0) of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications in Ts
- the term $w(T, Ts)$ counts the number $T + 1$ of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications in Ts
- the same principle is applied recursively for the counters, until the empty sequence is reached

The arithmetic interpretation of hereditarily binary numbers

Definition

The function $n : \mathbb{T} \rightarrow \mathbb{N}$ defines the unique natural number associated to a term of type $\mathbb{T}$.

$$n(T) = \begin{cases} 0 & \text{if } T = \texttt{e}, \\ 2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[])}, \\ (n(U)+1)2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[Y|Xs])} \text{ and } U = \texttt{w(Y,Xs)}, \\ 2^{n(X)+2} - 2 & \text{if } T = \texttt{w(X,[])}, \\ (n(U)+2)2^{n(X)+1} - 2 & \text{if } T = \texttt{w(X,[Y|Xs])} \text{ and } U = \texttt{v(Y,Xs)}. \end{cases} \quad (5)$$

The corresponding Prolog predicate, `n(T,N)` computes `N` as follows:

`?- n(w(v(e, []), [e, e, e]), N)` $\Rightarrow$

`N` = $(((2^{0+1} - 1 + 2)2^{0+1} - 2 + 1)2^{0+1} - 1 + 2)2^{2^{0+1}-1+1} - 2 = 42.$

the first few natural numbers are:

- 0:e,
- 1:v(e,[]),
- 2:w(e,[]),
- 3:v(v(e,[]),[]),
- 4:w(e,[e]),
- 5:v(e,[e])

- a term of the form `v(X,Xs)` represents an odd number $\in \mathbb{N}^+$
- a term of the form `w(X,Xs)` represents an even number $\in \mathbb{N}^+$.

Proposition

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

Successor and predecessor

- we specify successor and predecessor through a *reversible* Prolog predicate `s(Pred,Succ)` holding if `Succ` is the successor of `Pred`
- recursive calls to `s` in `s` happen on terms that are (roughly) logarithmic in the bitsize of their operands
- when computing the successor on the first $2^{30} = 1073741824$ natural numbers (with functional equivalents of `s` and its inverse), there are in total 2381889348 calls to `s`, averaging to 2.2183 per successor and predecessor computation

```
s(e,v(e,[])).
s(v(e,[]),w(e,[])).
s(v(e,[X|Xs]),w(SX,Xs)):-s(X,SX).
s(v(T,Xs),w(e,[P|Xs])):-s(P,T).
s(w(T,[]),v(ST,[])):-s(T,ST).
s(w(Z,[e]),v(Z,[e])).
s(w(Z,[e,Y|Ys]),v(Z,[SY|Ys])):-s(Y,SY).
s(w(Z,[X|Xs]),v(Z, [e,SX|Xs])):-s(SX,X).
```

Emulating the bijective base-2 operations o, *i*

- we emulate single applications of `o` and `i` seen in terms of `s`
- the predicates `o/2` and `i/2` are also *reversible*

```
o(e,v(e,[])).
o(w(X,Xs),v(e,[X|Xs])).
o(v(X,Xs),v(SX,Xs)):-s(X,SX).

i(e,w(e,[])).
i(v(X,Xs),w(e,[X|Xs])).
i(w(X,Xs),w(SX,Xs)):-s(X,SX).
```

- "recognizers" `o_` and `i_` detect `v` and `w` corresponding to `o` (and respectively `i`) being the last operation applied
- `s_` detects that the number is a successor, i.e., not the empty term `e`.

```
s_(v(_,_)).    s_(w(_,_)).

o_(v(_,_)).    i_(w(_,_)).
```

From $\mathbb{N}$ to $\mathbb{T}$

Definition

The function $t : \mathbb{N} \rightarrow \mathbb{T}$ defines the unique tree of type $\mathbb{T}$ associated to a natural number as follows:

$$t(x) = \begin{cases} \mathtt{e} & \textit{if } x = \mathtt{0}, \\ \mathtt{o}(t(\frac{x-1}{2})) & \textit{if } x > 0 \textit{ and } x \textit{ is odd}, \\ \mathtt{i}(t(\frac{x}{2} - 1)) & \textit{if } x > 0 \textit{ and } x \textit{ is even} \end{cases} \tag{6}$$

A few low complexity operations

- `o` is $\lambda x.2x+1$, doubling a number `db` and reversing the `db` operation (`hf`) are

```
db(X,Db):-o(X,OX),s(Db,OX).
hf(Db,X):-s(Db,OX),o(X,OX).
```

- exponent of 2 is:

```
exp2(e,v(e,[])).
exp2(X,R):-s(PX,X),s(v(PX,[]),R).
```

Proposition

The costs of `db, hf` *and* `exp2` *are within a constant factor from the cost of* `s`*.*

Proof.

It follows by observing that at most 2 calls to `s, o` are made in each. □

Arithmetic operations “one block at time”

- efficient addition and subtraction operations similar to the successor / predecessor `s`, that *work on one run-length encoded bloc at a time*, rather than by individual `o` and `i` steps
- key intuition: align / trim / fuse blocks of iterated applications before operating on them

the predicate `otimes` implements $o^n(k)$ and `itimes` implements $i^n(k)$

```
otimes(e,Y,Y).
otimes(N,e,v(PN,[])):-s(PN,N).
otimes(N,v(Y,Ys),v(S,Ys)):-add(N,Y,S).
otimes(N,w(Y,Ys),v(PN,[Y|Ys])):-s(PN,N).
```

```
itimes(e,Y,Y).
itimes(N,e,w(PN,[])):- s(PN,N).
itimes(N,w(Y,Ys),w(S,Ys)):-add(N,Y,S).
itimes(N,v(Y,Ys),w(PN,[Y|Ys])):-s(PN,N).
```

The chain of mutually recursive predicates

- otimes, itimes
- oplus, iplus, oiplus
- ominus, iminus, oiminus, iominus
- osplit, isplit
- add,sub,
- cmp,
- bitsize
- + a few other auxiliary predicates

while apparently intricate, the network of mutually recursive predicates is manageable as they all progress on structurally smaller terms

Addition: the math

We also need a number of arithmetic identities on $\mathbb{N}$ involving iterated applications of o and i.

Proposition

The following hold:

$$o^k(x) + o^k(y) = i^k(x+y) \tag{7}$$

$$o^k(x) + i^k(y) = i^k(x) + o^k(y) = i^k(x+y+1) - 1 \tag{8}$$

$$i^k(x) + i^k(y) = i^k(x+y+2) - 2 \tag{9}$$

Proof.

By (1) and (2), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides. □

Addition: the code

```
add(e,Y,Y).
add(X,e,X):-s_(X).
add(X,Y,R):-o_(X),o_(Y),
  osplit(X,A,As),osplit(Y,B,Bs),
  cmp(A,B,R1),
  auxAdd1(R1,A,As,B,Bs,R).
add(X,Y,R):-o_(X),i_(Y),
  osplit(X,A,As),isplit(Y,B,Bs),
  cmp(A,B,R1),
  auxAdd2(R1,A,As,B,Bs,R).
add(X,Y,R):-i_(X),o_(Y),
  isplit(X,A,As),osplit(Y,B,Bs),
  cmp(A,B,R1),
  auxAdd3(R1,A,As,B,Bs,R).
add(X,Y,R):-i_(X),i_(Y),
  isplit(X,A,As),isplit(Y,B,Bs),
  cmp(A,B,R1),
  auxAdd4(R1,A,As,B,Bs,R).
```

Subtraction: the math

Proposition

$$x > y \;\Rightarrow\; o^k(x) - o^k(y) = o^k(x - y - 1) + 1 \tag{10}$$

$$x > y+1 \;\Rightarrow\; o^k(x) - i^k(y) = o^k(x - y - 2) + 2 \tag{11}$$

$$x \geq y \;\Rightarrow\; i^k(x) - o^k(y) = o^k(x - y) \tag{12}$$

$$x > y \;\Rightarrow\; i^k(x) - i^k(y) = o^k(x - y - 1) + 1 \tag{13}$$

Proof.

By (1) and (2), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides. Note that special cases are handled separately to ensure that subtraction is defined. □

Defining a total order: comparison

```
cmp(e,e,'=').
cmp(e,Y,('<')):-s_(Y).
cmp(X,e,('>')):-s_(X).
cmp(X,Y,R):-s_(X),s_(Y),bitsize(X,X1),bitsize(Y,Y1),
  cmp1(X1,Y1,X,Y,R).

cmp1(X1,Y1,_,_,R):- \+(X1=Y1),cmp(X1,Y1,R).
cmp1(X1,X1,X,Y,R):-
  reversedDual(X,RX),reversedDual(Y,RY),
  compBigFirst(RX,RY,R).
```

- the predicate `compBigFirst` compares two terms known to have the same `bitsize`
- it works on reversed (big digit first) variants, computed by `reversedDual`
- it takes advantage of the block structure, because assuming two terms of the same bitsizes, the one starting with *i* is larger than one starting with *o*

Computing `bitsize`

- The predicate `bitsize` computes the number of applications of the `o` and `i` operations.
- It works by summing up the *counts* of `o` and `i` operations composing a tree-represented natural number of type $\mathbb{T}$.

```
bitsize(e,e).
bitsize(v(X,Xs),R):-tsum([X|Xs],e,R).
bitsize(w(X,Xs),R):-tsum([X|Xs],e,R).
```

```
tsum([],S,S).
tsum([X|Xs],S1,S3):-add(S1,X,S),s(S,S2),tsum(Xs,S2,S3).
```

`bitsize` concludes our chain of *mutually recursive* predicates.

Fast multiplication by an exponent of 2

$$\forall k \geq 0,\ o^n(k) = 2^n(k+1) - 1 \Rightarrow \forall k > 0,\ 2^n k = 2^n(k-1) + 1$$

```
leftShiftBy(_,e,e).
leftShiftBy(N,K,R):-s(PK,K),otimes(N,PK,M),s(M,R).
```

Structural complexity

as a measure of structural complexity we define the predicate `tsize` that counts the nodes of a tree of type $\mathbb{T}$ (except the root). It corresponds to the function $c : \mathbb{T} \rightarrow \mathbb{N}$ defined by equation (14):

$$c(T) = \begin{cases} 0 & \text{if } \mathtt{T} = \mathtt{e}, \\ \sum_{Y \in \mathtt{[X|Xs]}} (1 + c(Y)) & \text{if } \mathtt{T = v(X,Xs)}, \\ \sum_{Y \in \mathtt{[X|Xs]}} (1 + c(Y)) & \text{if } \mathtt{T = w(X,Xs)}. \end{cases} \tag{14}$$

The following holds:

Proposition

For all terms $T \in \mathbb{T}$, $\mathtt{tsize(T)} \leq \mathtt{bitsize(T)}$.

Structural complexity: the code

```
tsize(e,e).
tsize(v(X,Xs),R) :- tsizes([X|Xs],e,R).
tsize(w(X,Xs),R) :- tsizes([X|Xs],e,R).
```

```
tsizes([],S,S).
tsizes([X|Xs],S1,S4):-tsize(X,N),add(S1,N,S2),s(S2,S3),
  tsizes(Xs,S3,S4).
```

- for operations like `s, o, i, 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 `add`, `sub`, `cmp`, worst case is proportional with the tree size of the smallest operand
- 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

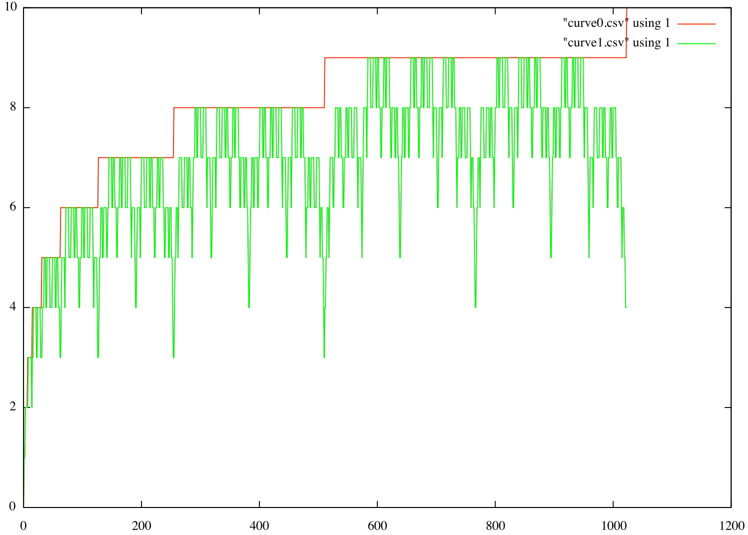

Figure : Structural complexity (yellow line) bounded by bitsize (red line) from 0 to $2^{10} - 1$

Best and worst case

```
?- t(3,T),bestCase(T,Best),n(Best,N).
T = v(v(e, []), []), Best = w(w(w(e, []), []), []),
N = 65534 . % <<<<< BEST

?- t(3,T),worstCase(T,Worst),n(Worst,N).
T = v(v(e, []), []), Worst = w(e, [e, e, e, e, e]),
N = 84 . % <<<<< WORST


?- t(20,X),bestCase(X,A),t(30,Y),bestCase(Y,B),add(A,B,C),
           tsize(C,S),n(S,TSize),write(TSize),nl,fail.
314  % <<<<<< a fairly large tree, but operations tractable
```

- last example: we can compute with towers of exponents 20 and 30 levels tall !
- $\Rightarrow$ this opens the door to a new world where tractability of computations is not limited by the size of the operands but only by their structural complexity

An interesting large number of low structural somplexity

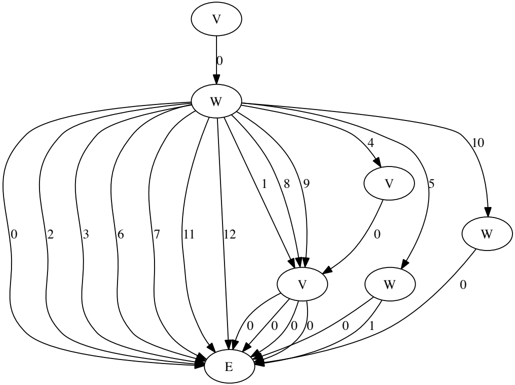

Figure : Largest known prime number: the 48-th Mersenne prime, $2^{57885161} - 1$

Conclusion

- we have shown that *computations* like addition, subtraction, exponent of 2 and bitsize can be performed with giant numbers in quasi-constant time or time proportional to their *structural complexity* rather than their *bitsize*
- our structural 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 :-)
- possible uses for constraint algorithms?
- Prolog code at `http://logic.cse.unt.edu/tarau/research/2013/hbn.pl`
- Scala and Haskell based open source project: at: `http:/code.google.com/p/giant-numbers/`