A Prolog Specification of Giant Number Arithmetic

Paul Tarau¹

¹Department of Computer Science and Engineering
University of North Texas

CICLOPS 2013
binary, decimal, base-N number arithmetics provide an exponential improvement over unary “caveman’s” notation

eyes turned out to be 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

⇒ this paper is about how we can 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
1. Notations for giant numbers vs. computations with giant numbers
2. Bijective base-2 numbers as iterated function applications
3. Hereditarily binary numbers
4. Successor and predecessor
5. Emulating the bijective base-2 operations $\circ$, $i$
6. Arithmetic operations
7. Structural complexity
8. Conclusion
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 ⇒ a hereditary number system based on recursively applied run-length compression of a special (bijection) binary digit notation ⇒ a concept of structural complexity is introduced, that serves as an indicator of the expected performance of our arithmetic operations
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 $n$. 
Some useful identities

\[ o^n(k) = 2^n(k + 1) - 1 \]  \hspace{1cm} (1)

\[ i^n(k) = 2^n(k + 2) - 2 \]  \hspace{1cm} (2)

and in particular

\[ o^n(0) = 2^n - 1 \]  \hspace{1cm} (3)

\[ i^n(0) = 2^{n+1} - 2 \]  \hspace{1cm} (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 \( \mathbb{T} \).

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

(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**

$n : T \rightarrow \mathbb{N}$ 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

\[
\begin{align*}
s(e, v(e, [])) & . \\
s(v(e, []), w(e, [])) & . \\
s(v(e, [X|Xs]), w(SX, Xs)) & :- s(X, SX) . \\
s(v(T, Xs), w(e, [P|Xs])) & :- s(P, T) . \\
s(w(T, []), v(ST, [])) & :- s(T, ST) . \\
s(w(Z, [e]), v(Z, [e])) & . \\
s(w(Z, [e,Y|Ys]), v(Z, [SY|Ys])) & :- s(Y, SY) . \\
s(w(Z, [X|Xs]), v(Z, [e, SX|Xs])) & :- s(SX, X) .
\end{align*}
\]
Emulating the bijective base-2 operations $\circ$, $i$

- we emulate single applications of $\circ$ and $i$ seen in terms of $s$
- the predicates $\circ/2$ and $i/2$ are also reversible

\begin{verbatim}
\circ(e, v(e, [])).
\circ(w(X, Xs), v(e, [X|Xs])).
\circ(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).
\end{verbatim}

- “recognizers” $\circ_\_ \text{ and } i_\_ \text{ detect } v \text{ and } w \text{ corresponding to } \circ \text{ (and respectively } i \text{) being the last operation applied}
- $s_\_ \text{ detects that the number is a successor, i.e., not the empty term } e$.

\begin{verbatim}
\s_(v(_, _)). \s_(w(_, _)).
\o_(v(_, _)). \i_(w(_, _)).
\end{verbatim}
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:

$$t(x) = \begin{cases} 
  e & \text{if } x = 0, \\
  o(t(x/2 - 1)) & \text{if } x > 0 \text{ and } x \text{ is odd}, \\
  i(t(x/2)) & \text{if } x > 0 \text{ and } x \text{ is even}
\end{cases} \quad (6)$$
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 $\text{otimes}$ implements $o^n(k)$ and $\text{itimes}$ implements $i^n(k)$

\begin{verbatim}
\text{otimes}(e,Y,Y).
\text{otimes}(N,e,v(PN,[])):-s(PN,N).
\text{otimes}(N,v(Y,Ys),v(S,Ys)):-\text{add}(N,Y,S).
\text{otimes}(N,w(Y,Ys),v(PN,[Y|Ys])):-s(PN,N).
\end{verbatim}

\begin{verbatim}
\text{itimes}(e,Y,Y).
\text{itimes}(N,e,w(PN,[])):-\text{s}(PN,N).
\text{itimes}(N,w(Y,Ys),w(S,Ys)):-\text{add}(N,Y,S).
\text{itimes}(N,v(Y,Ys),w(PN,[Y|Ys])):-\text{s}(PN,N).
\end{verbatim}
The chain of mutually recursive predicates

- $\otimes$, $\mathit{itimes}$
- $\oplus$, $\mathit{iplus}$, $\mathit{oiplus}$
- $\ominus$, $\mathit{iminus}$, $\mathit{oiminus}$, $\mathit{iominus}$
- $\mathit{osplit}$, $\mathit{isplit}$
- $\mathit{add,sub}$,
- $\mathit{cmp}$,
- $\mathit{bitsize}$

+ 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:

\[
o^k(x) + o^k(y) = i^k(x + y)
\] (7)

\[
o^k(x) + i^k(y) = i^k(x) + o^k(y) = i^k(x + y + 1) - 1
\] (8)

\[
i^k(x) + i^k(y) = i^k(x + y + 2) - 2
\] (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.
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

\[ x > y \implies o^k(x) - o^k(y) = o^k(x - y - 1) + 1 \]  
\[ x > y + 1 \implies o^k(x) - i^k(y) = o^k(x - y - 2) + 2 \]  
\[ x \geq y \implies i^k(x) - o^k(y) = o^k(x - y) \]  
\[ x > y \implies i^k(x) - i^k(y) = o^k(x - y - 1) + 1 \]

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

\[
\text{cmp}(e, e, '=').
\]
\[
\text{cmp}(e, Y, ('<')):-\text{s}_(Y).
\]
\[
\text{cmp}(X, e, ('>')):-\text{s}_(X).
\]
\[
\text{cmp}(X, Y, R):-\text{s}_(X),\text{s}_(Y),\text{bitsize}(X, X1),\text{bitsize}(Y, Y1),
\]
\[
\quad\text{cmp1}(X1, Y1, X, Y, R).
\]
\[
\text{cmp1}(X1, Y1, __, __, R):- \ \+ (X1=Y1),\text{cmp}(X1, Y1, R).
\]
\[
\text{cmp1}(X1, X1, X, Y, R):-
\]
\[
\quad\text{reversedDual}(X, RX),\text{reversedDual}(Y, RY),
\]
\[
\quad\text{compBigFirst}(RX, RY, R).
\]

- the predicate \text{compBigFirst} compares two terms known to have the same \text{bitsize}
- it works on reversed (big digit first) variants, computed by \text{reversedDual}
- it takes advantage of the block structure, because assuming two terms of the same bitsizes, the one starting with \text{i} is larger than one starting with \text{o}
The predicate \texttt{bitsize} computes the number of applications of the \texttt{o} and \texttt{i} operations.

It works by summing up the \textit{counts} of \texttt{o} and \texttt{i} operations composing a tree-represented natural number of type \texttt{T}.

\begin{verbatim}
bitsize(e,e).
bitsize(v(X,Xs),R):-tsum([X|Xs],e,R).
bitsize(w(X,Xs),R):-tsum([X|Xs],e,R).

\texttt{tsum([],S,S)}.
\texttt{tsum([X|Xs],S1,S3):-add(S1,X,S),s(S,S2),tsum(Xs,S2,S3)}.
\end{verbatim}

\texttt{bitsize} concludes our chain of \textit{mutually recursive} predicates.
\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).
as a measure of structural complexity we define the predicate \( \text{tsize} \) 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):

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

The following holds:

**Proposition**

*For all terms* \( T \in \mathbb{T}, \text{tsize}(T) \leq \text{bitsize}(T). \)
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 `≤`, we never lose.
- 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 !

- 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 complexity

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

⇒ 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](http://logic.cse.unt.edu/tarau/research/2013/hbn.pl)