A Prolog Specification of Giant Number Arithmetic

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
e-mail: tarau@cs.unt.edu

Abstract. The tree based representation described in this paper, *hereditarily binary numbers*, applies recursively a run-length compression mechanism that enables computations limited by the structural complexity of their operands rather than by their bitsizes. While within constant factors from their traditional counterparts for their worse case behavior, our arithmetic operations open the doors for interesting numerical computations, impossible with traditional number representations.
We provide a complete specification of our algorithms in the form of a purely declarative Prolog program.
Keywords: *hereditary numbering systems, compressed number representations, arithmetic computations with giant numbers, tree-based numbering systems, Prolog as a specification language.*

1 Introduction

While *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 tree-based numbering system that *allows computations* with numbers comparable in size with Knuth's "arrow-up" notation. Moreover, these computations have a worse 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 practically constant time and a number of other operations of practical interest like addition, subtraction and comparison benefit from significant complexity reductions. For the curious reader, it is basically a *hereditary number system* similar to [2], based on recursively applied *run-length* compression of a special (bijective) binary digit notation.

A concept of structural complexity is also introduced, based on the size of our tree representations. It provides estimates on worse and best cases for our algorithms and it serves as an indicator of the expected performance of our arithmetic operations.

We have adopted a *literate programming* style, i.e. the code contained in the paper forms a self-contained Prolog program (tested with SWI-Prolog, Lean Prolog and Styla), also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/hbn.pl` . 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 gives some background on representing bijective base-2 numbers as iterated function application and section 3 introduces hereditarily binary numbers. Section 4 describes practically constant time successor and predecessor operations on tree-represented numbers. Section 5 shows an emulation of bijective base-2 with hereditarily binary numbers and section 6 discusses some of their basic arithmetic operations. Section 7 defines a concept of structural complexity studies best and worse cases and comparisons with bitsizes. Section 8 discusses related work. Section 9 concludes the paper and discusses future work. Finally, an optimized general multiplication algorithm is described in the Appendix.

2 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 [3] together with the convention that 0 is represented as the empty sequence. As each $n \in \mathbb{N}$ can be seen as a unique composition of these functions we can make this precise as follows:

Definition 1 *We call bijective base-2 representation of $n \in \mathbb{N}$ the unique sequence of applications of functions o and i to ϵ that evaluates to n.*

With this representation, and denoting the empty sequence ϵ, one obtains $0 = \epsilon, 1 = o\ \epsilon, 2 = i\ \epsilon, 3 = o(o\ \epsilon), 4 = i(o\ \epsilon), 5 = o(i\ \epsilon)$ etc. and the following holds:

$$i(x) = o(x) + 1 \tag{1}$$

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

Proposition 1 *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.

The underlying arithmetic identities are:

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

$$i^n(k) = 2^n(k+2) - 2 \tag{3}$$

and in particular

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

$$i^n(0) = 2^{n+1} - 2 \tag{5}$$

3 Hereditarily binary numbers

3.1 Hereditary Number Systems

Let us observe that conventional number systems, as well as the bijective base-2 numeration system described so far, represent blocks of 0 and 1 digits somewhat naively - one digit for each element of the block. Alternatively, one might think that counting them and representing the resulting counters as *binary numbers* would be also possible. But then, the same principle could be applied recursively. So instead of representing each block of 0 or 1 digits by as many symbols as the size of the block – essentially a *unary* representation – one could also encode the number of elements in such a block using a *binary* representation.

This brings us to the idea of hereditary number systems.

3.2 Hereditarily binary numbers as a data type

First, we define a data type for our tree represented natural numbers, that we call *hereditarily binary numbers* to emphasize that *binary* rather than *unary* encoding is recursively used in their representation.

Definition 2 *The data type* $\mathbb{T}$ *of the set of hereditarily binary numbers is defined inductively as the set of Prolog terms such that:*

$$X \in \mathbb{T} \text{ if and only if } X = e \text{ or } X \text{ is of the form } v(T, T^*) \text{ or } w(T, T^*) \tag{6}$$

where $T \in \mathbb{T}$ and T^* stands for a finite sequence (list) of elements of $\mathbb{T}$.

The intuition behind the set $\mathbb{T}$ is the following:

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

One can see this process as run-length compressed bijective base-2 numbers, represented as trees with either empty leaves or at least one branch, after applying the encoding recursively.

Definition 3 *The function $n : \mathbb{T} \to \mathbb{N}$ shown in equation* **7** *defines the unique natural number associated to a term of type $\mathbb{T}$.*

$$n(T) = \begin{cases} 0 & \text{if } T = \texttt{e}, \\ 2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[])}, \\ (n(U)+1)2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[Y|Xs])} \text{ and } U = \texttt{w(Y,Xs)}, \\ 2^{n(X)+2} - 2 & \text{if } T = \texttt{w(X,[])}, \\ (n(U)+2)2^{n(X)+1} - 1 & \text{if } T = \texttt{w(X,[Y|Xs])} \text{ and } U = \texttt{v(Y,Xs)}. \end{cases} \quad (7)$$

For instance, the computation of `N` in `?- n(w(v(e, []), [e, e, e]),N)` expands to $(((2^{0+1} - 1 + 2)2^{0+1} - 2 + 1)2^{0+1} - 1 + 2)2^{2^{0+1}-1+1} - 2 = 42$. The Prolog equivalent of equation (7) (using bitshifts for exponents of 2) is:

```
n(e,0).
n(v(X,[]),R) :-n(X,Z),R is 1≪(1+Z)-1.
n(v(X,[Y|Xs]),R):-n(X,Z),n(w(Y,Xs),K),R is (K+1)*(1≪(1+Z))-1.
n(w(X,[]),R):-n(X,Z),R is 1≪(2+Z)-2.
n(w(X,[Y|Xs]),R):-n(X,Z),n(v(Y,Xs),K),R is (K+2)*(1≪(1+Z))-2.
```

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

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

Note that a term of the form `v(X,Xs)` represents an odd number $\in \mathbb{N}^+$ and a term of the form `w(X,Xs)` represents an even number $\in \mathbb{N}^+$. The following holds:

Proposition 2 *$n : \mathbb{T} \to \mathbb{N}$ is a bijection, i.e., each term canonically represents the corresponding natural number.*

Proof. It follows from the identities (2), (3) by replacing the power of 2 functions with the corresponding iterated applications of o and i.

4 Successor and predecessor

We will now specify successor and predecessor through a *reversible* Prolog predicate `s(Pred,Succ)` holding if `Succ` is the successor of `Pred`.

```
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).
```

It can be proved by structural induction that Peano's axioms hold and as a result $< \mathbb{T}, e, s >$ is a Peano algebra.

Note also that recursive calls to `s` in `s` happen on terms that are (roughly) logarithmic in the bitsize of their operands. One can therefore assume that their complexity, computed by an *iterated logarithm*, is practically constant[1].

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

To be of any practical interest, we will need to ensure that our data type $\mathbb{T}$ emulates also binary arithmetic. We will first show that it does, and next we will show that on a number of operations like exponent of 2 or multiplication by an exponent of 2, it significantly lowers complexity.

Intuitively, the first step should be easy, as we need to express single applications or "un-applications" of `o` and `i` in terms of their iterates encapsulated in the terms of type $\mathbb{T}$.

First we emulate single applications of `o` and `i` seen in terms of `s`. Note that `o/2` and `i/2` are also *reversible* predicates.

```
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).
```

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

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

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

Note that each of the predicates `o` and `i` calls `s` and on a term that is (roughly) logarithmically smaller. It follows that

[1] Empirically, when computing the successor on the first $2^{30} = 1073741824$ natural numbers, there are in total 1727815601 calls to `s`, averaging to 1.6091 per successor computation. The same average for 100 successor computations on 5000 bit random numbers also oscillates around 1.60.

Proposition 3 *Assuming* `s` *constant time,* `o,i` *are also constant time.*

Definition 4 *The function* $t : \mathbb{N} \to \mathbb{T}$ *defines the unique tree of type* $\mathbb{T}$ *associated to a natural number as follows:*

$$t(x) = \begin{cases} \texttt{e} & \text{if } x = \texttt{0}, \\ \texttt{o}(t(\frac{x-1}{2})) & \text{if } x > 0 \text{ and } x \text{ is odd}, \\ \texttt{i}(t(\frac{x}{2}-1)) & \text{if } x > 0 \text{ and } x \text{ is even} \end{cases} \tag{8}$$

We can now define the corresponding Prolog predicate that converts from terms of type $\mathbb{T}$ to natural numbers. Note that we use bitshifts (`>>`) for division by 2.

```
t(0,e).
t(X,R):-X>0, X mod 2=:=1,Y is (X-1)>>1, t(Y,A),o(A,R).
t(X,R):-X>0, X mod 2=:=0,Y is (X>>1)-1, t(Y,A),i(A,R).
```

The following holds:

Proposition 4 *Let* `id` *denote* $\lambda x.x$ *and "*$\circ$*" function composition. Then, on their respective domains*

$$t \circ n = id, \quad n \circ t = id \tag{9}$$

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

6 Arithmetic operations

6.1 A few low complexity operations

Doubling a number `db` and reversing the `db` operation (`hf`) are quite simple, once one remembers that the arithmetic equivalent of function `o` is $o(x) = 2x + 1$.

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

Note that efficient implementations follow directly from our number theoretic observations in section 2. For instance, as a consequence of proposition 1, the operation `exp2` computing an exponent of 2 , has the following simple definition in terms of `s`.

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

Proposition 5 *Assuming* `s` *constant time,* `db,hf` *and* `exp2` *are also constant time.*

Proof. It follows by observing that only 2 calls to `s,o` are made.

6.2 Reduced complexity addition and subtraction

We now derive 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.

We first define the predicates `otimes` corresponding to $o^n(k)$ and `itimes` corresponding to $i^n(k)$.

```
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).
```

They are part of a chain of *mutually recursive predicates* as they are already referring to the `add` predicate, to be implemented later. Note also that instead of naively iterating, they implement a more efficient "one bloc at a time" algorithm. For instance, when detecting that its argument counts a number of applications of `o`, `otimes` just increments that count. On the other hand, when the last predicate applied was `i`, `otimes` simply inserts a new count for `o` operations. A similar process corresponds to `itimes`. As a result, performance is (roughly) logarithmic rather than linear in terms of the bitsize of argument `N`. We will use this property for implementing a low complexity multiplication by exponent of 2 operation.

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

Proposition 6 *The following hold:*

$$o^k(x) + o^k(y) = i^k(x + y) \tag{10}$$

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

$$i^k(x) + i^k(y) = i^k(x + y + 2) - 2 \tag{12}$$

Proof. By (2) and (3), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides.

The corresponding Prolog code is:

```
oplus(K,X,Y,R):-add(X,Y,S),itimes(K,S,R).

oiplus(K,X,Y,R):-add(X,Y,S),s(S,S1),itimes(K,S1,T),s(R,T).

iplus(K,X,Y,R):-add(X,Y,S),s(S,S1),s(S1,S2),itimes(K,S2,T),s(P,T),s(R,P).
```

Note that the code uses `add` that we will define later and that it is part of a chain of mutually recursive predicate calls, that together will provide an intricate but efficient implementation of the intuitively simple idea: *we want to work on one run-length encoded block at a time.*

The corresponding identities for subtraction are:

Proposition 7

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

$$x > y + 1 \;\Rightarrow\; o^k(x) - i^k(y) = o^k(x - y - 2) + 2 \tag{14}$$

$$x \geq y \;\Rightarrow\; i^k(x) - o^k(y) = o^k(x - y) \tag{15}$$

$$x > y \;\Rightarrow\; i^k(x) - i^k(y) = o^k(x - y - 1) + 1 \tag{16}$$

Proof. By (2) and (3), 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.

The Prolog code, also covering the special cases, is:

```
ominus(_,X,X,e).
ominus(K,X,Y,R):-sub(X,Y,S1),s(S2,S1),otimes(K,S2,S3),s(S3,R).
```

```
iminus(_,X,X,e).
iminus(K,X,Y,R):-sub(X,Y,S1),s(S2,S1),otimes(K,S2,S3),s(S3,R).
```

```
oiminus(_,X,Y,v(e,[])):-s(Y,X).
oiminus(K,X,Y,R):-s(Y,SY),s(SY,X),exp2(K,P),s(P,R).
oiminus(K,X,Y,R):-
  sub(X,Y,S1),s(S2,S1),s(S3,S2),s_(S3), % S3 <> e
  otimes(K,S3,S4),s(S4,S5),s(S5,R).
```

```
iominus(K,X,Y,R):-sub(X,Y,S),otimes(K,S,R).
```

Note the use of `sub`, to be defined later, which is also part of the mutually recursive chain of operations.

The next two predicates extract the iterated applications of o^n and respectively i^n from `v` and `w` terms:

```
osplit(v(X,[]), X,e).
osplit(v(X,[Y|Xs]),X,w(Y,Xs)).
```

```
isplit(w(X,[]), X,e).
isplit(w(X,[Y|Xs]),X,v(Y,Xs)).
```

We are now ready for defining addition. The base cases are:

```
add(e,Y,Y).
add(X,e,X):-s_(X).
```

In the case when both terms represent odd numbers, we apply with `auxAdd1` the identity (10), after extracting the iterated applications of o as `a` and `b` with the predicate `osplit`.

```
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).
```

In the case when the first term is odd and the second even, we apply with `auxAdd2` the identity (11), after extracting the iterated application of o and i as `a` and `b`.

```
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).
```

In the case when the first term is even and the second odd, we apply with `auxAdd3` the identity (11), after extracting the iterated applications of i and o as, respectively, `a` and `b`.

```
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).
```

In the case when both terms represent even numbers, we apply with `auxAdd4` the identity (12), after extracting the iterated application of i as `a` and `b`.

```
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).
```

Note the presence of the comparison operation `cmp`, to be defined later, also part of our chain of mutually recursive operations. Note also that in each case we ensure that a block of the same size is extracted, depending on which of the two operands `a` or `b` is larger. Beside that, the auxiliary predicates `auxAdd1, auxAdd2, auxAdd3` and `auxAdd4` implement the equations of Prop. 6.

```
auxAdd1('=',A,As,_B,Bs,R):- s(A,SA),oplus(SA,As,Bs,R).
auxAdd1('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oplus(SB,R1,Bs,R).
auxAdd1('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),oplus(SA,As,R1,R).
```

```
auxAdd2('=',A,As,_B,Bs,R):- s(A,SA),oiplus(SA,As,Bs,R).
auxAdd2('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oiplus(SB,R1,Bs,R).
auxAdd2('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),oiplus(SA,As,R1,R).
```

```
auxAdd3('=',A,As,_B,Bs,R):- s(A,SA),oiplus(SA,As,Bs,R).
auxAdd3('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),oiplus(SB,R1,Bs,R).
auxAdd3('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),oiplus(SA,As,R1,R).
```

```
auxAdd4('=',A,As,_B,Bs,R):- s(A,SA),iplus(SA,As,Bs,R).
auxAdd4('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),iplus(SB,R1,Bs,R).
auxAdd4('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),iplus(SA,As,R1,R).
```

The code for the subtraction predicate `sub` is similar:

```
sub(X,e,X).
sub(X,Y,R):-o_(X),o_(Y),osplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxSub1(R1,A,As,B,Bs,R).
```

In the case when both terms represent odd numbers, we apply the identity (13), after extracting the iterated applications of o as `a` and `b`. For the other cases, we use, respectively, the identities 14, 15 and 16:

```
sub(X,Y,R):-o_(X),i_(Y),osplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxSub2(R1,A,As,B,Bs,R).
```

```
sub(X,Y,R):-i_(X),o_(Y),isplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxSub3(R1,A,As,B,Bs,R).
```

```
sub(X,Y,R):-i_(X),i_(Y),isplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxSub4(R1,A,As,B,Bs,R).
```

Note also the auxiliary predicates `auxSub1, auxSub2, auxSub3` and `auxSub4` that implement the equations of Prop. 7.

```
auxSub1('=',A,As,_B,Bs,R):- s(A,SA),ominus(SA,As,Bs,R).
auxSub1('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),ominus(SB,R1,Bs,R).
auxSub1('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),ominus(SA,As,R1,R).
```

```
auxSub2('=',A,As,_B,Bs,R):- s(A,SA),oiminus(SA,As,Bs,R).
auxSub2('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oiminus(SB,R1,Bs,R).
auxSub2('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),oiminus(SA,As,R1,R).
```

```
auxSub3('=',A,As,_B,Bs,R):- s(A,SA),ominus(SA,As,Bs,R).
auxSub3('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),ominus(SB,R1,Bs,R).
auxSub3('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),ominus(SA,As,R1,R).
```

```
auxSub4('=',A,As,_B,Bs,R):- s(A,SA),iminus(SA,As,Bs,R).
auxSub4('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),iminus(SB,R1,Bs,R).
auxSub4('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),iminus(SA,As,R1,R).
```

6.3 Defining a total order: comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our type $\mathbb{T}$. It relies on `bitsize` computing the number of applications of o and i that build a term in $\mathbb{T}$, which is also part of our mutually recursive predicates, to be defined later.

We first observe that only terms of the same bitsize need detailed comparison, otherwise the relation between their bitsizes is enough, *recursively*. More precisely, the following holds:

Proposition 8 *Let* `bitsize` *count the number of applications of o or i operations on a bijective base-2 number. Then* $\texttt{bitsize}(x) < \texttt{bitsize}(y) \Rightarrow x < y$.

Proof. Observe that, given their lexicographic ordering in "big digit first" form, the bitsize of bijective base-2 numbers is a non-decreasing function.

```
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):- λ+(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` and it takes advantage of the block structure using the following proposition:

Proposition 9 *Assuming two terms of the same bitsizes, the one starting with i is larger than one starting with o.*

Proof. Observe that "big digit first" numbers are lexicographically ordered with $o < i$.

As a consequence, `cmp` only recurses when *identical* blocks head the sequence of blocks, otherwise it infers the "`<`" or "`>`" relation.

```
compBigFirst(e,e,'=').
compBigFirst(X,Y,R):- o_(X),o_(Y),
  osplit(X,A,C),osplit(Y,B,D),cmp(A,B,R1),
  fcomp1(R1,C,D,R).
compBigFirst(X,Y,R):-i_(X),i_(Y),
  isplit(X,A,C),isplit(Y,B,D),cmp(A,B,R1),
  fcomp2(R1,C,D,R).
compBigFirst(X,Y,('<')):-o_(X),i_(Y).
compBigFirst(X,Y,('>')):-i_(X),o_(Y).
```

```
fcomp1('=',C,D,R):-compBigFirst(C,D,R).
fcomp1('<',_,_,'>').
fcomp1('>',_,_,'<').
```

```
fcomp2('=',C,D,R):-compBigFirst(C,D,R).
fcomp2('<',_,_,'<').
fcomp2('>',_,_,'>').
```

The predicate `reversedDual` reverses the order of application of the o and i operations to a "biggest digit first" order. For this, it only needs to reverse the order of the alternative blocks of o^k and i^k. It uses the predicate `len` to compute with `auxRev1` and `auxRev2` the number of these blocks. Then, it infers that if the number of blocks is odd, the last block is of the same kind as the first; otherwise it is of its alternate kind (`w` for `v` and vice versa).

```
reversedDual(e,e).
reversedDual(v(X,Xs),R):-reverse([X|Xs],[Y|Ys]),len([X|Xs],L),
  auxRev1(L,Y,Ys,R).
reversedDual(w(X,Xs),R):-reverse([X|Xs],[Y|Ys]),len([X|Xs],L),
  auxRev2(L,Y,Ys,R).
```

```
auxRev1(L,Y,Ys,R):-o_(L),R=v(Y,Ys).
auxRev1(L,Y,Ys,R):-i_(L),R=w(Y,Ys).
```

```
auxRev2(L,Y,Ys,R):-o_(L),R=w(Y,Ys).
auxRev2(L,Y,Ys,R):-i_(L),R=v(Y,Ys).
```

```
len([],e).
len([_|Xs],L):- len(Xs,L1),s(L1,L).
```

6.4 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. Note that it also provides an efficient implementation of the integer log_2 operation `ilog2`.

```
ilog2(X,R):-s(PX,X),bitsize(PX,R).
```

6.5 Fast multiplication by an exponent of 2

The predicate `leftshiftBy` uses Prop. 1, i.e., the fact that repeated application of the `o` operation (`otimes`) provides an efficient implementation of multiplication with an exponent of 2.

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

The following holds:

Proposition 10 *Assuming* `s` *constant time,* `leftshiftBy` *is (roughly) logarithmic in the bitsize of its arguments.*

Proof. it follows by observing that at most one addition on data logarithmic in the bitsize of the operands is performed.

7 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).

```
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).
```

It corresponds to the function $c : \mathbb{T} \to \mathbb{N}$ defined by equation (17):

$$c(T) = \begin{cases} 0 & \text{if T} = \text{ e}, \\ \sum_{Y \in \texttt{[X|Xs]}} (1 + c(Y)) & \text{if T} = \texttt{v(X,Xs)}, \\ \sum_{Y \in \texttt{[X|Xs]}} (1 + c(Y)) & \text{if T} = \texttt{w(X,Xs)}. \end{cases} \tag{17}$$

The following holds:

Proposition 11 *For all terms* $T \in \mathbb{T}$, `tsize(T)` $\leq$ `bitsize(T)`.

Proof. By induction on the structure of T, by observing that the two predicates have similar definitions and corresponding calls to `tsize` return terms assumed smaller than those of `bitsize`.

The following example illustrates their use:

```
?- t(123456,T),tsize(T,S1),n(S1,TSize),bitsize(T,S2),n(S2,BSize).
T = w(e, [w(e, [e]), e, v(e, []), e, w(e, []), w(e, [])]),
S1 = w(e, [e, e]), TSize = 12,
S2 = w(e, [w(e, [])]), BSize = 16 .
```

After defining the predicate `iterated`, that applies `K` times the predicate `F`

```
iterated(_,e,X,X).
iterated(F,K,X,R):-s(PK,K),iterated(F,PK,X,R1),call(F,R1,R).
```

we can exhibit a best case, of minimal structural complexity for its size

```
bestCase(K,Best):-iterated(wtree,K,e,Best).

wtree(X,w(X,[])).
```

and a worse case, of maximal structural complexity for its size

```
worseCase(K,Worse):-iterated(io,K,e,Worse).

io(X,Z):-o(X,Y),i(Y,Z).
```

The following examples illustrate these predicates:

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

?- t(3,T),worseCase(T,Worse),n(Worse,N).
T = v(v(e, []), []), Worse = w(e, [e, e, e, e, e]), N = 84 .
```

It follows from identity (5) that the predicate `bestCase` computes the iterated exponent of 2 (tetration) and then applies the predecessor to it twice, i.e., it computes $2^{2^{\cdots^2}} - 2$. A simple closed formula (easy to proof by induction) can also be found for `worseCase`:

Proposition 12 *The predicate* `worseCase k` *computes the value in* $\mathbb{T}$ *corresponding to the value* $\frac{4(4^k-1)}{3} \in \mathbb{N}$.

The average space-complexity of our number representation is related to the average length of the *integer partitions of the bitsize of a number* [4]. Intuitively, the shorter the partition in alternative blocks of o and i applications, the more significant the compression is, but the exact study, given the recursive application of run-length encoding, is likely to be quite intricate.

The following example shows that computations with towers of exponents 20 and 30 levels tall become possible with our number representation.

```
?- 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
```

Note that the structural complexity of the result (that we did not print out) is still quite manageable: `250`. *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.*

8 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 tree-based natural numbers, such notations are not closed under addition and multiplication, and consequently 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 ω allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates.

Arithmetic packages similar to our bijective base-2 view of arithmetic operations are part of libraries of proof assistants like Coq [5].

Arithmetic computations based on recursive data types like the free magma of binary trees (isomorphic to the context-free language of balanced parentheses) are described in [3], where they are seen as Gödel's `System T` types, as well as combinator application trees. In [6] a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite sequences. In [7] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types.

9 Conclusion and future work

We have shown that *computations* like addition, subtraction, exponent of 2 and bitsize can be performed with giant numbers in constant time or time proportional to their structural complexity rather than their bitsize. As *structural complexity* is bounded by bitsize, our computations are within constant time from their traditional counterparts, as also illustrated by our best and worse case complexity cases.

The fundamental theoretical challenge raised at this point is the following: *can more number-theoretically interesting operations expressed succinctly in terms of our tree-based data type? Is it possible to reduce the complexity of some other important operations, besides those found so far?*

The general multiplication algorithm in the `Appendix` shows a first step in that direction.

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. 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
4. Corteel, S., Pittel, B., Savage, C.D., Wilf, H.S.: On the multiplicity of parts in a random partition. Random Struct. Algorithms **14**(2) (1999) 185–197

5. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2012) Version 8.4.
6. 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
7. 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

Appendix

Reduced complexity general multiplication

We can devising a similar optimization as for `add` and `sub` for multiplication:

Proposition 13 *The following holds:*

$$o^n(a)o^m(b) = o^{n+m}(ab + a + b) - o^n(a) - o^m(b) \tag{18}$$

Proof. By 2, we can expand and then reduce as follows: $o^n(a)o^m(b) = (2^n(a + 1) - 1)(2^m(b + 1) - 1) = 2^{n+m}(a + 1)(b + 1) - (2^n(a + 1) + 2^m(b + 1) + 1 = 2^{n+m}(a + 1)(b + 1) - 1 - (2^n(a + 1) - 1 + 2^m(b + 1) - 1 + 2) + 2 = o^{n+m}(ab + a + b + 1) - (o^n(a) + o^m(b)) - 2 + 2 = o^{n+m}(ab + a + b) - o^n(a) - o^m(b)$

The corresponding Prolog code starts with the obvious base cases:

```
mul(_,e,e).
mul(e,Y,e):-s_(Y).
```

When both terms represent odd numbers we apply the identity (18):

```
mul(X,Y,R):-o_(X),o_(Y),osplit(X,N,A),osplit(Y,M,B),
  add(A,B,S),mul(A,B,P),add(S,P,P1),s(N,SN),s(M,SM),
  add(SN,SM,K),otimes(K,P1,P2),sub(P2,X,R1),sub(R1,Y,R).
```

The other cases are reduced to the previous one by the identity $i = s \circ o$.

```
mul(X,Y,R):-o_(X),i_(Y),s(PY,Y),mul(X,PY,Z),add(X,Z,R).
mul(X,Y,R):-i_(X),o_(Y),s(PX,X),mul(PX,Y,Z),add(Y,Z,R).
mul(X,Y,R):-i_(X),i_(Y),
  s(PX,X),s(PY,Y),add(PX,PY,S),mul(PX,PY,P),add(S,P,R1),s(R1,R).
```

Note that when the operands are composed of large blocks of alternating o^n and i^m applications, the algorithm works (roughly) in time proportional to the number of blocks rather than the number of digits. The following example illustrates a multiplication with two "tower of exponent" terms:

```
?- t(30,X),bestCase(X,A),s(A,N),t(40,Y),bestCase(Y,B),s(B,M),
   mul(M,N,P),tsize(P,S),n(S,TSize),write(TSize),nl,fail.
668
```

The structural complexity of the result, `668` is in indicator that such computations are still tractable.