Bijective Collection Encodings and Boolean Operations with Hereditarily Binary Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
`http://www.cse.unt.edu/~tarau`

Abstract

Our tree-based *hereditarily binary numbers* apply recursively a run-length compression mechanism. They enable performing arithmetic computations symbolically and lift tractability of computations to be limited by the representation size of their operands rather than by their bitsizes.

We apply them to derive compact representations for "structurally simple" (sparse or dense) lists, sets and multisets, as well as their hereditarily finite counterparts. This enables the use of hereditarily binary numbers to define bijective size-proportionate Gödel numberings for several data types, that we "virtualize" through a generic data type transformation framework. As an application, a size-proportionate Gödel numbering scheme for term algebras is derived.

After extending the arithmetic operations on hereditarily binary numbers with boolean operations, we use them to perform computations with bitvectors and sets as well as a 3-valued logic interpretation for bijective base-2 bitvectors.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Data types and structures

General Terms Algorithms, Languages, Theory

Keywords hereditary numbering systems, compressed number representations, compact bijective encodings of sparse data structures, symbolic arithmetic, computations with giant numbers, tree-based numbering systems.

1. Introduction

This paper is a sequel to [14] where we have introduced a tree based canonical number representation, called *hereditarily binary numbers*, that uses *run-length encoding of bijective base-2 numbers*, recursively.

Bijective base-2 numbers are a canonical representation of natural numbers, where each combination of the 2 digits corresponds to a unique number. When run-length compression is applied recursively a tree-based number representation is obtained which is also canonical.

We have described in [14] specialized algorithms for basic arithmetic operations, that favor *numbers with relatively few blocks of contiguous* `0` *and* `1` *digits*, for which dramatic complexity reductions result even when operating on very large, "towers-of-exponents" numbers. In addition, we have shown that worst case and average case complexity of arithmetic operations is within constant factor of their bitstring counterparts.

The main focus of this paper is on applications of *hereditarily binary numbers* that go beyond arithmetic operations.

Of particular interest are *bijective encodings* of lists, multisets and sets of natural numbers, that result in exponential blow-up when represented with the usual binary notation. On the other hand, we will show that bijections of hereditarily finite sets to $\mathbb{N}$ result in *size-proportionate encodings* when computed with hereditarily binary numbers.

As an application, we obtain a size-proportionate Gödel numbering scheme for term algebras.

As another application, we design boolean operations taking advantage of sparse/dense bitvector representations expressed efficiently with hereditarily finite binary numbers.

These results prove the usefulness of hereditarily binary numbers as a general-purpose canonical number representation.

The paper is organized as follows. Section 2 overviews basic definitions for hereditarily binary numbers and summarizes some of their properties, following [14]. Section 3 describes compact encodings of sparse and dense sets, multisets and lists using hereditarily binary numbers and connects our data types through isomorphisms that allow transferring operations between them. Section 4 extends these to encodings of hereditarily finite lists, sets and multisets. Section 6 describes size-proportionate Gödel numberings of term algebras using hereditarily binary numbers. Section 7 introduces bitvector operations using hereditarily binary numbers and their corresponding set equivalents. Section 8 defines approximations of lower structural complexity. Section 9 discusses related work and section 10 concludes the paper.

We have adopted a *literate programming* style, i.e., the code contained in the paper (restricted to a minimalist subset of Haskell) forms a one file module (tested with ghc 7.6.3) and it is available at `http://www.cse.unt.edu/~tarau/research/2014/HBS.hs`. It imports the code from the literate program [14], also available at `http://www.cse.unt.edu/~tarau/research/2014/HBin.hs`.

2. Hereditarily Binary Numbers

We will summarize, following [14], the basic concepts behind *hereditarily binary numbers*. Through the paper, we denote $\mathbb{N}$ the

PPDP '14, September 8–10, 2014, Canterbury, England, U.K..

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

set of natural numbers and $\mathbb{N}^+$ the set of strictly positive natural numbers.

2.1 Bijective base-2 numbers

Natural numbers can be seen as represented by iterated applications of the functions $o(x) = 2x + 1$ and $i(x) = 2x + 2$ corresponding to the so called *bijective base-2* representation (defined for an arbitrary base in [9] pp. 90-92 as "m-adic" numbers). 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 0 that evaluates to n.*

With this representation, one obtains , $1 = o(0), 2 = i(0), 3 = o(o(0)), 4 = i(o(0)), 5 = o(i(0))$ etc. Clearly:

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

2.2 Efficient arithmetic with iterated functions o^n and i^n

Several arithmetic identities are proven in [14] and used to express efficient "one block of o^n or i^n operations at a time" algorithms for various arithmetic operations. Among them, we recall the following two, showing the connection of our iterated function applications with "left shift/multiplication by a power of 2" operations.

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

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

In particular

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

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

2.3 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 in [14] by the Haskell declaration:*

```
data T = E | V T [T]  | W T [T]
  deriving (Eq,Read,Show)
```

corresponding to the recursive data type equation $\mathbb{T} = 1 + \mathbb{T} \times \mathbb{T}^* + \mathbb{T} \times \mathbb{T}^*$.

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

- The term `E` (empty leaf) corresponds to zero
- the term `V x xs` counts the number `x+1` of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications `xs`
- the term `W x xs` counts the number `x+1` of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications `xs`

In [14] the bijection between $\mathbb{N}$ and $\mathbb{T}$ is provided by the function $n : \mathbb{T} \to \mathbb{N}$ and its inverse $t : \mathbb{N} \to \mathbb{T}$.

DEFINITION 3. *The function $n : \mathbb{T} \to \mathbb{N}$ shown in equation (**6**) defines the unique natural number associated to a term of type $\mathbb{T}$.*

This bijection ensures that hereditarily binary numbers provide a canonical representation of natural numbers and the equality relation on type $\mathbb{T}$ can be derived by structural induction.

The following examples show the workings of the bijection n and illustrate that "structural complexity", defined in [14] as the *size of the tree representation without the root*, is bounded by the bitsize of a number and favors numbers in the neighborhood of towers of exponents of 2.

$2^{2^{16}} - 1 \to$ `V (V (V (V (V E[])[])[])[])[]`
$\to 2^{2^{2^{2^{2^{0+1}-1+1}-1+1}-1+1}-1+1} - 1$
$20 \to$ `W E [E,E,E]`
$\to (((2^{0+1} - 1 + 2)2^{0+1} - 2 + 1)2^{0+1} - 1 + 2)2^{0+1} - 2$

In [14] basic arithmetic operations are introduced with complexity parameterized by the size of the tree representation of their operands rather than their bitsize.

After defining constant average time *successor* and *predecessor* functions `s` and `s'`, constant average time definitions of `o` and `i` are given in [14], as well as for the corresponding inverse operations `o'` and `i'`, that can be seen as "un-applying" a single instance of `o` or `i`, and "recognizers" `e_` (corresponding to `E`), `o_` (corresponding to *odd* numbers) and `i_` (corresponding to *even* numbers).

We refer to the **Appendix** for a list of functions imported from[14] used in this paper.

2.4 Bijective base-2 vs. traditional binary numbers

One might wonder why we would prefer bijective number representations to the traditional binary numbers. Note that compositions of $o(x) = 2x + 1$ and $i(x) = 2x + 2$ applied to 0 correspond uniquely to natural numbers resulting in a *canonical representation*. On the other hand, $\lambda x.2x$ applied to 0 is still 0, giving to 0 an infinite number of equivalent representations. Note also that besides unique decoding, canonical representations allow testing for *syntactic equality*.

3. Representing sets, multisets and lists

We will start by describing bijective mappings between *collection* types as well as a Gödel numbering scheme putting them in bijection with natural numbers. *Our goal is to show that natural number encodings for sparse instances of these collections will have space-efficient representations as natural numbers of type $\mathbb{T}$, in contrast to bitstring-based representations.*

3.1 Bijections between collections and natural numbers

We will first convert between natural numbers and lists, by using the bijection $f(x, y) = 2^x(2y + 1)$, corresponding to the function `cons`.

```
cons :: (T,T)→T
cons (E,y) = o y
cons (x,y) = s (f (s' (o y))) where
  f E = V (s' x) []
  f (W y xs) = V (s' x) (y:xs)
```

We refer to [14] for the definitions of *successor* and *predecessor* functions `s` and `s'`. The function `decons` inverts `cons` to a Haskell ordered pair.

```
decons :: T → (T,T)
decons z | o_ z = (E, o' z)
decons z | i_ z = (s x, g xs) where
  V x xs = s' z

  g [] = E
  g (y:ys) = s (i' (W y ys))
```

PROPOSITION 1. *The operations* `cons` *and* `decons` *are constant time on the average and $O(log^*(bitsize))$ in the worst case, where log^* is the iterated logarithm function, counting how many times log can be applied before reaching 0.*

$$n(t) = \begin{cases} 0 & \text{if } t = \texttt{E}, \\ 2^{n(\texttt{x})+1} - 1 & \text{if } t = \texttt{V x []}, \\ (n(u)+1)2^{n(\texttt{x})+1} - 1 & \text{if } t = \texttt{V x (y:xs)} \text{ and } u = \texttt{W y xs}, \\ 2^{n(\texttt{x})+2} - 2 & \text{if } t = \texttt{W x []}, \\ (n(u)+2)2^{n(\texttt{x})+1} - 2 & \text{if } t = \texttt{W x (y:xs)} \text{ and } u = \texttt{V y xs}. \end{cases} \tag{6}$$

Proof It is proven in [14] that `o, o', i, i'` have the same worst case and average complexity as `s` and `s'`, i.e., constant average and $O(log^*(bitsize))$ worst case and that `o_` and `i_` are constant time operations. Observe that a constant number of them is used in each branch of `cons` and `decons`, therefore the worst case and average complexity of `cons` and `decons` are also the same as that of `s` and `s'`.

The bijection between natural numbers and lists of natural numbers `to_list` and its inverse `from_list` apply repeatedly `decons` and `cons`.

```
to_list :: T → [T]
to_list z | e_ z = []
to_list z = x : to_list y where (x,y) = decons z
```

```
from_list :: [T] → T
from_list [] = E
from_list (x:xs) = cons (x,from_list xs)
```

3.2 Bijections between sequences, sets and multisets

Non-decreasing sequences provide a canonical representation for multisets of natural numbers. While finite multisets and finite lists of elements of $\mathbb{N}$ share a common representation `[N]`, multisets are subject to the implicit constraint that their ordering is immaterial. This suggest that a multiset like $[4, 4, 1, 3, 3, 3]$ could be represented canonically as sequence by first ordering it as $[1, 3, 3, 3, 4, 4]$ and then computing the differences between consecutive elements i.e. $[x_0, x_1 \ldots x_i, x_{i+1} \ldots] \rightarrow [x_0, x_1 - x_0, \ldots x_{i+1} - x_i \ldots]$. This gives $[1, 2, 0, 0, 1, 0]$, with the first element 1 followed by the increments $[2, 0, 0, 1, 0]$

Therefore, *incremental sums*, computed with Haskell's `scanl`, are used to transform arbitrary lists to multisets of natural numbers, inverted by *pairwise differences* computed using `zipWith`.

```
list2mset, mset2list :: [T] → [T]

list2mset [] = []
list2mset (n:ns) = scanl add n ns

mset2list [] = []
mset2list (m:ms) = m : zipWith sub ms (m:ms)
```

Sets of natural numbers are canonically represented as *increasing* sequences. The bijection between sequences and sets is obtained by slightly modifying the bijection to multisets, by mapping non-decreasing to increasing sequences.

```
list2set, set2list :: [T] → [T]

list2set = (map s') . list2mset . (map s)

set2list = (map s') . mset2list . (map s)
```

By composing with natural number-to-list bijections, we obtain bijections to multisets and sets of natural numbers.

```
to_mset :: T → [T]
to_mset = list2mset . to_list

from_mset :: [T] → T
from_mset = from_list . mset2list
```

```
to_set :: T → [T]
to_set = list2set . to_list

from_set :: [T] → T
from_set = from_list . set2list
```

As the following example shows, trees of type $\mathbb{T}$ offer a significantly more compact representation of sparse sets than conventional binary numbers.

```
> n (bitsize (from_set (map t [42,1234,6789])))
6789
> n (tsize (from_set (map t [42,1234,6789])))
32
```

Note that a similar compression occurs for sets of natural numbers with only a few elements missing (that we call *dense sets*), as they have the same representation size with type $\mathbb{T}$ as the `dual` of their sparse counterpart.

```
> n (tsize (from_set (map t ([1,3,5]++[6..220]))))
12
> n (bitsize (from_set (map t ([1,3,5]++[6..220]))))
220
```

The following holds:

PROPOSITION 2. *These encodings/decodings of lists,sets and multisets as hereditarily binary numbers are size-proportionate i.e., their representation sizes are within constant factors.*

3.3 Bijective Data Type Transformations

Along the lines of [11] we can define bijective transformations between data types as follows:

```
data Iso a b = Iso (a→b) (b→a)

from (Iso f _) = f
to (Iso _ f') = f'
```

Morphing between data types is provided by the combinator `as`:

```
as :: Iso a b → Iso c b → c → a
as that this x = to that (from this x)
```

We can now define our "virtual types" as bijections to a common representation. Our tree-based natural numbers will form the "hub" `nat` to/from which other types are transformed.

```
nat = Iso id id
```

The collection types for lists, sets and multisets follow:

```
list, mset, set :: Iso [T] T

list = Iso from_list to_list

mset = Iso from_mset to_mset

set = Iso from_set to_set
```

This results in a small "embedded language" that morphs between our "virtual types" as illustrated by the following example.

```
> as set nat (t 123)
[E,V E [],V (V E []) [],W E [E],V E [E],W (V E []) []]
> map n it
```

```
[0,1,3,4,5,6]
> map t it
[E,V E [],V (V E []) [],W E [E],V E [E],W (V E []) []]
> n (as nat set it)
123
```

We can define combinators that borrow operations from another virtual type, as follows:

```
borrow1 :: Iso b c → (b → b) → Iso a c → a → a
borrow1 lender f borrower =
  as borrower lender . f . as lender borrower
```

```
borrow2 :: Iso c b → (c→c→c) → Iso a b → (a→a→a)
borrow2 lender op borrower x y =
  as borrower lender (op x' y') where
    x'= as lender borrower x
    y'= as lender borrower y
```

We can also encapsulate the bijection between binary bitstring-represented natural numbers and tree-represented natural numbers as the virtual type `bitnat`.

```
type N = Integer
bitnat :: Iso N T
bitnat = Iso t n
```

The following examples illustrate these operations:

```
> as list bitnat 20
[W E [],V E []]
> as bitnat list [V E [],W E [E], E]
194
> as list bitnat it
[V E [],W E [E],E]
> borrow2 bitnat (*) nat (W E []) (V E [E])
W (V E []) [E]
```

3.4 Another compact representation of lists

As with the constructs in subsection 3.1, we start with the functions `decons'` and `cons'` that provide bijections between $\mathbb{N}^+$ and $\mathbb{N} \times \mathbb{N}$. They can be used as an alternative mechanism for building bijections between natural numbers and lists, multisets and sets of natural numbers, based on *separating blocks of* `o` *and* `i` *applications* that build up a natural number represented in bijective base 2.

Implementing `decons'` and `cons'` amounts to extracting/inserting the count of applications of `o` and `i` and encoding/decoding the type of constructor (`V` or `V`) as a "parity bit" added to the first component of the pair.

```
decons' :: T→(T,T)
decons' (V x []) = (s' (o x),E)
decons' (V x (y:ys)) = (x,W y ys)
decons' (W x []) = (o x,E)
decons' (W x (y:ys)) = (x,V y ys)
```

```
cons' :: (T,T)→T
cons' (E,E) = V E []
cons' (x,E) | o_ x =  W (o' x) []
cons' (x,E) | i_ x = V (o' (s x)) []
cons' (x,V y ys) = W x (y:ys)
cons' (x,W y ys) = V x (y:ys)
```

PROPOSITION 3. *The operations* `cons'` *and* `decons'` *are constant time on the average and* $O(log^*(bitsize))$ *in the worst case, where* log^* *is the iterated logarithm function.*

Proof Observe that, as proven in [14] `o`, `o'`, `i`, `i'`,`o_`,`i_` are average constant time and a constant number of them are used in each branch of `cons'` and `decons'`.

An alternative bijection between natural numbers and lists of natural numbers, `to_list'` and its inverse `from_list'` is obtained by applying repeatedly the average constant time operations `cons'` and respectively `decons'`.

```
to_list' :: T → [T]
to_list' x | e_ x = []
to_list' x = hd : (to_list' tl) where (hd,tl)=decons' x
```

```
from_list' :: [T] → T
from_list' [] = E
from_list' (x:xs) = cons' (x,from_list' xs)
```

By composing with list to set and multiset bijections we obtain:

```
to_mset' = list2mset . to_list'
from_mset' = from_list' . mset2list
```

```
to_set' = list2set . to_list'
from_set' = from_list' . set2list
```

PROPOSITION 4. *These encodings/decodings of lists,sets and multisets as hereditarily binary numbers are size-proportionate.*

The following example illustrates these encodings:

```
> map (map n.to_list'.t) [0..15]
[[],[0],[1],[2],[0,0],[0,1],[3],[4],[0,2],[0,0,0],[1,0],
 [1,1],[0,0,1],[0,3],[5],[6],[0,4],[0,0,2]]
> map (n.from_list'.map t) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
```

Note the shorter lists, created close to powers of 2, coming from the longer blocks of consecutive `o` and `i` operations in that region.

The corresponding "virtual types are":

```
list', mset', set' :: Iso [T] T

list' = Iso from_list' to_list'

mset' = Iso from_mset' to_mset'

set' = Iso from_set' to_set'
```

The following examples illustrates their use:

```
> as set nat (t 42)
[V E [],V (V E []) [],V E [E]]
> as set' set it
[V E [],W E [],V (V E []) [],W E [E]]
> as nat set' it
W (V E []) [E,E,E]
> n it
42
```

A comparison is due at this point with our simpler `cons` and `decons`-based encoding described in subsection 3.1. First, the encoding described here is better, as it performs well on both *sparse* and *dense* sets. However the first one is of "historical" importance as it emulates the well known Ackerman's bijection from hereditarily finite sets to $\mathbb{N}$. We will explore this correspondence in more detail in section 4. Note also that while both encodings blow up as a tower of exponents with the usual binary representation, they are size-proportionate with our tree-based encoding.

4. Hereditarily Finite Lists, Sets and Multisets

We will use the data type `H` to support our hereditarily finite collection types similar to those described in [12].

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

The function `t2h` lifts the a transformer `f`, defined from type $\mathbb{T}$ to a collection type, to its hereditarily finite correspondent.

```
t2h :: (T → [T]) → T → H
t2h f E = H []
t2h f n = H (map (t2h f) (f n))
```

Similarly, the function `h2t` lifts the a transformer `f`, defined from a collection type to type $\mathbb{T}$, to its hereditarily finite correspondent.

```
h2t :: ([T] → T) → H → T
h2t g (H []) = E
h2t g (H hs) = g (map (h2t g) hs)
```

clearly, if `f` and `g` are inverses, then so are `t2h` and `h2t`.

Our virtual data types for hereditarily finite lists, multisets and sets, `hfl`, `hfm` and `hfs` are defined in terms of `h2t` and `t2h`:

```
hfl, hfm, hfs :: Iso H T
hfl = Iso  (h2t from_list) (t2h to_list)
hfm = Iso  (h2t from_mset) (t2h to_mset)
hfs = Iso (h2t from_set) (t2h to_set)
```

After defining *Ackermann's bijection* from hereditarily finite sets to our tree-represented natural numbers $\mathbb{T}$

```
ackermann :: H → T
ackermann (H xs) = foldr add E (map (exp2 . ackermann) xs)
```

one can notice that it is identical to the inverse of the bijection "`as hfs bitnat`":

```
> as hfs bitnat 42
H [H [H []],H [H [],H [H []]],H [H [],H [H [H []]]]]
> ackermann it
W (V E []) [E,E,E]
> n it
42
> as bitnat hfs (as hfs bitnat 42)
42
```

Similarly, using our alternative transformers, we define `hfl'`, `hfm'` and `hfs'` as follows:

```
hfl', hfm', hfs' :: Iso H T
hfl' = Iso  (h2t from_list') (t2h to_list')
hfm' = Iso  (h2t from_mset') (t2h to_mset')
hfs' = Iso (h2t from_set') (t2h to_set')
```

Note the small tree size of the `hfs'` representation - matching that of type $\mathbb{T}$, in contrast to the bitsize of the corresponding natural number.

```
> as hfs' nat
    (sub (exp2 (exp2 (exp2 (exp2 (t 2))))) (t 5))
H [H [H []],H [H [H []],H [H [],H [H [H [H []]]]]]]
> n (bitsize (as nat hfs' it))
65535
```

The following holds:

PROPOSITION 5. *These encodings/decodings of hereditarily finite lists, sets and multisets as hereditarily binary numbers are size-proportionate.*

The key intuition behind this fact is that *when numbers are represented as trees, it is relatively easy to ensure that a "natural" bijection between them and various tree-represented data types is size proportionate*.

5. Permutations of $\mathbb{N}$ from permutations of $\mathbb{T}$

A simple bijection between $\mathbb{T}$ and itself (permutation of $\mathbb{T}$) is provided by the `dual` operation that flips toplevel constructors `V` and `W`, and as a result reinterprets all o operations as ı operations and vice-versa. It is therefore its own inverse (an *involution*). It can be "borrowed" by the type $\mathbb{N}$ of ordinary natural numbers as follows:

```
> map (borrow1 nat dual bitnat) [0..15]
[0,2,1,6,5,4,3,14,13,12,11,10,9,8,7,30]
> map (borrow1 nat dual bitnat) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

A more interesting permutation of $\mathbb{T}$ is provided by working on the hereditarily finite list equivalent of an object in $\mathbb{T}$, a member of the Catalan family [10].

```
hdual :: H → H
hdual (H []) = H []
hdual (H (x:xs)) = H (hdual (H xs): ys) where
  H ys = hdual x
```

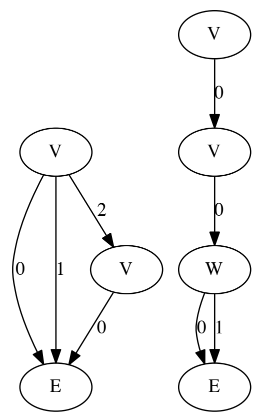
The function `hdual` looks at the multiway trees of type `H` as if they were binary trees (another member of the Catalan family) and *flips their left and right branches*, recursively. We use the bijections provided by our "virtual types" to morph this into `tdual` by borrowing it from the the hereditarily finite list type `hfl'`

```
tdual :: T → T
tdual = borrow1 hfl' hdual nat
```

The resulting bijection is also an *involution*, as illustrated by the following example (borrowed to work on $\mathbb{N}$, for readability):

```
> map (borrow1 nat tdual bitnat) [0..17]
[0,1,4,9,2,7,20,5,62,3,10,94,30,2047,16,41,14,4294967295]
> map (borrow1 nat tdual bitnat) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
```

Note the very large value `4294967295` corresponding to `17` as illustrated by the taller dual in Fig. 1



(a) t 17 (b) tdual of (t 17)

Figure 1: Duals, with trees folded into DAGs. Arcs marked with order of children.

```
> t 17
V E [E,V E []]
> tdual (t 17)
V (V (W E [E]) []) []
```

This asymmetry is even more dramatic for 28:

```
> t 28
W E [E,V E []]
> tdual (t 28)
W (V (V E [E,E]) []) []
> n (bitsize it)
1024
```

Note the dramatic increase in bitsize from `4` to `1024`. It is shown in [14] that the average bitsize of a block is only 2 bits, and therefore the trees of type $\mathbb{T}$ associated to random natural numbers are much wider than tall. The interest of the involution `tdual` is its ability to flip between relatively small random numbers (with high Kolmogorov complexity [6]) and giant numbers with a regular structure corresponding to "tall trees" that can be seen as built from combinations of iterated exponentials.

6. Bijective size-proportionate encodings of term algebras

Term algebras can be seen as the underlying data type shared by a large number of widely used concepts ranging from terms used in logic programming system and proof assistants to syntax trees and XML files.

Devising a Gödel numbering scheme for term algebras that is both size-proportionate and bijective is a difficult task as shown in [13] , when the traditional polynomial number representation is used. It involves a fairly sophisticated ranking/unranking algorithm for Catalan families [10], in combination with a generalization of Cantor's pairing function to tuples [3, 7], the only known polynomial bijection between $\mathbb{N}$ and sequences of natural numbers.

The solution to the same problem using hereditarily binary numbers is strikingly simple. The basic intuition is that we avoid exponential blow-up as we are mapping trees to trees rather than to strings of bits.

We start by defining the data type `Term` hosting a generic term algebra.

```
data Term a = Var a | Const a | Fun a [Term a]
  deriving (Eq,Show,Read)
```

The bijection from $\mathbb{T}$ to terms is defined as follows.

```
toTerm :: T → Term T
toTerm E = Var E
toTerm (V x []) = Var (s x)
toTerm (W x []) = Const x
toTerm (V x xs) = Fun (o x) (map toTerm xs)
toTerm (W x xs) = Fun (db x) (map toTerm xs)
```

Its inverse is also quite straightforward.

```
fromTerm :: Term T → T
fromTerm (Var E) = E
fromTerm (Var y) = V (s' y) []
fromTerm (Const x) = W x []
fromTerm (Fun k xs) | o_ k =
  V (o' k) (map fromTerm xs)
fromTerm (Fun k xs) =
  W (hf k) (map fromTerm xs)
```

The following examples illustrate that this bijection is indeed size-proportionate.

```
> fromTerm (Fun E [Fun E [Fun E [Const E]]])
W E [W E [W E [W E []]]]
>  n (bitsize it)
262146
> fromTerm (Fun E [Fun E [Fun E [Fun E [Const E]]]])
W E [W E [W E [W E [W E []]]]]
>  n (bitsize (bitsize it))
262146
```

While the first term corresponds to a large (262146 bits) number computed as
$(2^{(2^{(2^{2^{0+2}}-2+1}-1+2)2^{0+1}-2+1}-1+2)2^{0+1}-2+1} - 1 + 2)2^{0+1} - 2$
the second is already a giant 2^{262146} bit number.

Note also that we have used trees of type `Term T` rather than the more obvious type `Term N` to ensure that the encoding is size proportionate both ways. Otherwise, if using the type `Term N`, numbering variables, constants and function symbols could explode when converted from an tree of type `T` that, for instance, contains as subterm a tower exponents.

We can encapsulate this transformation in the form of the virtual data type `term`.

```
term :: Iso (Term T) T
term = Iso  fromTerm toTerm
```

The following example shows the conversion from/to a conventionally represented natural number, of virtual type `bitnat`.

```
> as term bitnat 12345
Fun (V E [])
    [Var E,Var E,Const E,Fun (V E []) [Var E],Var E]
> as bitnat term it
12345
```

7. Bitwise operations and their applications

We implement bitvector operations (also seen as efficient bitset operations) to work "one block of o^n or i^n applications at a time", to facilitate their use on large but sparse boolean formulas involving a large numbers of variables. One will be able to evaluate such formulas "all value-combinations at a time" when represented as bitvectors of size 2^{2^n}. Note that such operations will be tractable with our trees, provided that they have a relatively small structural complexity, despite their large bitsize.

7.1 Boolean operations on tree-represented bitvectors

The function `bitwiseOr` implements the bitwise disjunction operations on our tree numbers seen as bitvectors.

```
bitwiseOr :: T → T → T
bitwiseOr E y = y
bitwiseOr x E = x
bitwiseOr x y = s (bwOr (s' x) (s' y))
```

The actual work is delegated to the function `bwOr`. Note that we are mapping a bijective base-2 number to its corresponding bitwise representation by applying the predecessor `s'` and mapping back the result by applying the successor `s`, except for the case when an argument is `E`, which is handled directly (see 7.4 for details of this correspondence). The base cases of `bwOr` are:

```
bwOr :: T → T → T
bwOr E y | o_ y  = s y
bwOr x E | o_ x  = s x
bwOr E y = y
bwOr x E = x
```

Next, in a way similar to the `add` operation in [14], we proceed by case analysis. When both arguments are odd, we extract the blocks of applications of o^a and o^b from each argument with `osplit`, defined in [14], where definitions of `otimes` and `itimes` are also given. We remind that `otimes` and `itimes`, defined in [14], are used for merging blocks of applications of o or i. The function `osplit` returns also the "leftover" even numbers as "`as`" and "`bs`".

After comparing `a` and `b` with `cmp`, defined in [14], the local function `f` is used to process the remaining blocks.

```
bwOr x y | o_ x && o_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = osplit y
  f EQ = orApplyO (s a) as bs
  f GT = orApplyO (s b) (otimes (sub a b) as) bs
  f LT = orApplyO (s a) as (otimes (sub b a) bs)
```

Note that it calls the function `orApplyO` that merges the applications of o^k with the result of calling `bwOr` recursively.

The case when the first number is odd and the second even is similar, except that `isplit` is used instead of `osplit` and the helper function `orApplyI` is called, which merges the applications of i^k with the result of calling `bwOr` recursively.

```
bwOr x y |o_ x && i_ y = f (cmp a b) where
  (a,as) = osplit x
  (b,bs) = isplit y
```

```
  f EQ = orApplyI (s a) as bs
  f GT = orApplyI (s b) (otimes (sub a b) as) bs
  f LT = orApplyI (s a) as (itimes (sub b a) bs)
```

The case when the second number is odd and the first is even also uses `orApplyI` as required for the result of the disjunction operation.

```
bwOr x y |i_ x && o_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = osplit y
  f EQ = orApplyI (s a) as bs
  f GT = orApplyI (s b) (itimes (sub a b) as) bs
  f LT = orApplyI (s a) as (otimes (sub b a) bs)
```

The case when both arguments are even also uses `orApplyI` for the same reason.

```
bwOr x y |i_ x && i_ y = f (cmp a b) where
  (a,as) = isplit x
  (b,bs) = isplit y
  f EQ = orApplyI (s a) as bs
  f GT = orApplyI (s b) (itimes (sub a b) as) bs
  f LT = orApplyI (s a) as (itimes (sub b a) bs)
```

Finally the two helper functions are:

```
orApplyO k x y =  otimes k (bwOr x y)

orApplyI k x y =  itimes k (bwOr x y)
```

Note that they use `otimes` (defined in [14]), applying an o^k-block and `itimes` applying an i^k-block.

Bitwise negation (requiring the additional parameter `k` to specify the intended bitlength of the operand) corresponds to the complement w.r.t. the "universal set" of all natural numbers up to 2^k-1. It is defined as usual, by subtracting from the "bitmask" corresponding to $2^k - 1$:

```
bitwiseNot :: T → T → T
bitwiseNot k x = sub y x where y = s' (exp2 k)
```

The function `bitwiseAndNot` combines `bitwiseOr`, `bitwiseNot` the usual way, except that it uses the helper function `bitsOf` to ensure enough mask bits are made available when negation is applied.

```
bitwiseAndNot :: T → T → T
bitwiseAndNot x y = bitwiseNot l d  where
  l = max2 (bitsOf x) (bitsOf y)
  d = bitwiseOr (bitwiseNot l x) y
```

The function `max2` is defined in terms of comparison operation `cmp` as follows:

```
max2 :: T → T → T
max2 x y = if LT==cmp x y then y else x
```

The function `bitsOf` adapts the integer base-2 logarithm `ilog2`, (defined in [14]), to compute the number of bits of a bitvector.

```
bitsOf :: T → T
bitsOf E = s E
bitsOf x = s (ilog2 x)
```

Bitwise conjunction `bitwiseAnd` is similar, relying also on `bitsOf`:

```
bitwiseAnd :: T → T → T
bitwiseAnd x y = bitwiseNot l d where
  l = max2 (bitsOf x) (bitsOf y)
  d = bitwiseOr (bitwiseNot l x) (bitwiseNot l y)
```

Finally, `bitwiseXor` combines two `bitwiseAndNot` operations with a bitwise disjunction:

```
bitwiseXor :: T → T → T
bitwiseXor x y =
  bitwiseOr (bitwiseAndNot x y) (bitwiseAndNot y x)
```

The following example illustrates that our bitwise operations can be efficiently applied to giant numbers:

```
> bitwiseXor (s (exp2 (exp2 (t 12345))))
     (s' (exp2 (exp2 (t 6789))))
W (V (W E [E,E,V (V E []) [],E,E,E,E,E]) [])
  [V (W E [E,E,V (V E []) [], E,E,E,E,E])
   [W (V E []) [V E [],V E [],E,V E [],E,E,E]]]
> n (tsize it)
39
```

Note that the operation `tsize` (see [14]) computes the structural complexity of a term, defined as the size of its tree representation.

7.2 Set operations

With help from the data transformation operation `borrow2` we can use bitvectors for set operations:

```
setIntersection :: [T]→[T]→[T]
setIntersection = borrow2 nat bitwiseAnd set
```

```
setUnion :: [T]→[T]→[T]
setUnion = borrow2 nat bitwiseOr set
```

The following example illustrates these operations:

```
> map n (setUnion (map t [1,2,3,4])(map t [2,3,6,7]))
[1,2,3,4,6,7]
```

Note that sparse or dense sets containing very large sparse or dense elements benefit significantly from this encoding, given that, despite possibly very large bitsizes involved, it would result in representations of small structural complexity.

7.3 Boolean formula evaluation

Besides definitions for the boolean functions, we also need projection variable $var(n,k)$ corresponding to column k of a truth table for a function with n variables. A compact formula for them, as given in [5] or [15], is

$$var(n,k) = (2^{2^n} - 1)\ /\ (2^{2^{n-k-1}} + 1) \tag{7}$$

However, instead of doing the division, one can compute them as a concatenation of alternating blocks of 1 and 0 bits to take advantage of our efficient block operations.

```
var :: T → T → T
var n k = repeatBlocks nbBlocks blockSize mask where
  k' = s k
  nbBlocks = exp2 k'
  blockSize = exp2 (sub n k')
  mask = s' (exp2 blockSize)
```

The alternating blocks are put together by the function `repeatBlocks` that shifts to the left by the size of a block, at each step, and adds the `mask` made of 2^{n-k} ones, at each even step.

```
  repeatBlocks E _ _ = E
  repeatBlocks k l mask =
   if o_ k then r else add mask r where
    r = leftshiftBy l (repeatBlocks (s' k) l mask)
```

The following example illustrates the evaluation of a boolean formula in conjunctive normal form (CNF). The mechanism is usable as a simple satisfiability or tautology tester, for formulas resulting in possibly large but sparse or dense, low structural complexity bitvectors.

```
cnf = andN (map orN cls) where
  cls = [[v0',v1',v2],[v0,v1',v2],
         [v0',v1,v2'],[v0',v1',v2'],[v0,v1,v2]]

  v0 = var (t 3) (t 0)
```

```
  v1 = var (t 3) (t 1)
  v2 = var (t 3) (t 2)

  v0' = bitwiseNot (exp2 (t 3)) v0
  v1' = bitwiseNot (exp2 (t 3)) v1
  v2' = bitwiseNot (exp2 (t 3)) v2

  orN (x:xs) = foldr bitwiseOr x xs
  andN (x:xs) = foldr bitwiseAnd x xs
```

The execution of function `cnf` evaluates the formula, the result corresponding to bitvector `88 = [0,0,0,1,1,0,1,0]`.

```
> cnf
W E [V E [],V E [],E]
> n it
88
```

This rises the question: *could we use hereditarily binary numbers as a representation of boolean formulas with potential application to circuits?* As they can be seen as a compact representation of the *truth tables* of sparse or or dense boolean formulas one will need only to find the bit corresponding to an input described by a row in the truth table.

A quick look at the bijective base-2 representation of the first few natural numbers shows that they can be seen as successive listings of the columns in truth tables for $0, 1, \ldots, n$-argument boolean functions.

```
(0, [])
(1, [0])
(2, [1])
(3, [0,0])
(4, [1,0])
(5, [0,1])
(6, [1,1])
(7, [0,0,0])
(8, [1,0,0])
(9, [0,1,0])
(10,[1,1,0])
(11,[0,0,1])
(12,[1,0,1])
(13,[0,1,1])
(14,[1,1,1])
```

The following holds:

PROPOSITION 6. *The bijective base-2 representation of natural numbers in* $[2^n - 1 \mathinner{..} 2^{n+1} - 2]$ *describes the inputs in the truth table of a* n*-argument boolean function.*

This suggests defining the value of a boolean function represented as a the (hereditarily binary) natural number x in the range $[0 \mathinner{..} 2^{2^n} - 1]$ as computed by the function `bitval`:

```
bitval :: T → T → T → N
bitval numberOfVars k x | cmp k numberOfRows == LT &&
                cmp x (s truthTableStart) == LT =
  nthBit k (add truthTableStart x) where
    numberOfRows = exp2 numberOfVars
    truthTableStart = s' (exp2 numberOfRows)
```

Note that the variable `numberOfVars` counts the number of input variables and the variable `truthTableStart` marks the starting point in the sequence of bijective base-2 representations, corresponding to the inputs rows in the truth table.

The actual work is performed by `nthBit` in time proportional to the number of blocks, by finding the block to which the bit belongs and then return `0` if it is a `V` block and `1` if it is a `W` block.

```
nthBit k (V x (y:xs)) | cmp k x == GT =
  nthBit (sub k (s x)) (W y xs)
nthBit k (V _ _) = 0
nthBit k (W x (y:xs)) | cmp k x == GT =
  nthBit (sub k (s x)) (V y xs)
nthBit k (W _ _) = 1
```

Note that the functions `sub` and `cmp` defined in [14] are used to progressively subtract block sizes from the index of the bit `k` that we are looking for and, respectively, to check if the right block is reached. As we know for sure that all rows in the corresponding truth table have the same length, validation of inputs `k` and `x` is done only once in function `bitval`.

The following example, using the boolean expression `cnf` known to evaluate to `88 = [0,0,0,1,1,0,1,0]` illustrates the use of hereditarily binary numbers as a representation for boolean formulas.

```
> cnf
W E [V E [],V E [],E]
> bitval (t 3) (t 0) cnf
0
> bitval (t 3) (t 1) cnf
0
> bitval (t 3) (t 2) cnf
0
> bitval (t 3) (t 3) cnf
1
> bitval (t 3) (t 4) cnf
1
> bitval (t 3) (t 5) cnf
0
> bitval (t 3) (t 6) cnf
1
> bitval (t 3) (t 7) cnf
0
```

Note that this suggest a use[1] of hereditarily binary numbers as "boolean circuits" described directly as truth tables, provided that the corresponding natural numbers have a low representation complexity by being made of relatively few contiguous blocks (i.e.; corresponding to sparse or dense binary representations). As in the case of `BDDs` [1], the order of variables can be important and assigning lower k in the $v\ n\ k$ encoding to frequently occurring ones can help with the representation size of the resulting hereditarily binary numbers.

7.4 Transforming between bijective and traditional binary numbers

Hereditarily binary numbers are built as a run-length compressed representation of bijective base-2 numbers while bitvector operations (and their set equivalents) are performed on the traditional binary representation. So it makes sense to look into efficient, "one block at a time" transformations between the two.

We start by defining a block-oriented reversal of bijective base-2 bit representation in terms of operations on hereditarily binary numbers.

The function `rev` proceeds simply by calling `reverse` on the blocks while making sure that the appropriate constructor (`V` or `W`) is assigned to the result, based on the length (computed by `len`) being odd or even.

```
rev :: T → T
rev E = E
rev (V x xs) = r where
  y:ys = reverse (x:xs)
  r = if o_ (len (x:xs)) then V y ys else W y ys
rev (W x xs) = r where
  y:ys = reverse (x:xs)
  r = if o_ (len (x:xs)) then W y ys else V y ys
```

[1] Most likely of theoretical interest only, given that this is unlikely to compete with the current hardware implementations of logic gates-based circuits.

The following holds:

PROPOSITION 7. *The function* `rev` *is an injection from* $\mathbb{T}$ *to* $\mathbb{T}$ *and the composition* `rev . rev` *is the identity application on type* $\mathbb{T}$.

The following examples illustrate its use:

```
> map (n . rev . t) [0..15]
[0,1,2,3,5,4,6,7,11,9,13,8,12,10,14,15]
> map (n . rev . t) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

We will now define a "view" of a bijective base-2 number as a traditional one, provided by the function `as_bin`. When looking at the binary bijective base-2 representation of for instance `12=[0,0,1,1]`, one can notice that it is the same as the bijective base-2 representation of `11=[0,0,1]` with the extra `1` at the end.

```
as_bin E = V E []
as_bin x = (rev . i . rev . s') x
```

We can define the inverse of `as_bin` `as_bbin` as follows:

```
as_bbin (V E []) = E
as_bbin x = (s . rev . i' . rev) x
```

These functions are based on the fact that that a if a 1 digit is added after the highest end of the bijective base-2 representation to the predecessor of a number `x`, then we obtain its traditional binary digits, with the exception of `0` which is handled specially.

The following examples illustrate their use:

```
> map (n.as_bin.t) [0..15]
[1,2,5,6,11,12,13,14,23,24,25,26,27,28,29,30]
> map (n.as_bbin.t) it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

7.5 Bitwise operations, using a 3-valued logic

An interesting question arises at this point: is it possible to use our implicit bijective base-2 representation directly as the basis of a bitvector logic?

The answer is positive, provided that we use a slightly modified version of *Kleene's 3-valued logic* for bit operations. The key intuition is that if "o" stands for "known to be false", "i" stands for "known to be true", then absence of a corresponding value, when one sequence of applications is shorter than the other, will be interpreted as *unknown*. Note that this happens in a stronger sense than in Kleene's logic: conjunction of a value with *unknown* would be interpreted as *unknown*. It is easy to see that this also results in a de Morgan algebra, with the usual double negation and de Morgan's laws verified, and with behavior on classical truth values conserved. If coded in Haskell, the logic would be described by the following truth tables for negation and conjunction

```
negation UnKnown = UnKnown
negation False = True
negation True = False

conjunction False False = False
conjunction False True = False
conjunction True False = False
conjunction True True = True
conjunction False UnKnown = False
conjunction UnKnown False = False
conjunction True UnKnown = True
conjunction UnKnown True = True
conjunction UnKnown UnKnown = UnKnown
```

Negation `neg` can be implemented as the constant time `dual` operation, defined in [14], that flips `V` and `W` with the effect of implicitly flipping all o^n and i^n blocks.

```
neg = dual
```

Note that the *unknown* case corresponds to the sequence of applications ending with `E`.

The *bitwise and* operation `conj` is implemented using classical conjunction for each bit. In this case too, unknown corresponds to one or the other of the sequences ending with `E`.

```
conj :: T → T → T
conj E _ = E
conj _ E = E
conj x y | o_ x && o_ y = o (conj (o' x) (o' y))
conj x y | o_ x && i_ y = o (conj (o' x) (i' y))
conj x y | i_ x && o_ y = o (conj (i' x) (o' y))
conj x y | i_ x && i_ y = i (conj (i' x) (i' y))
```

Similarly, exclusive disjunction `xdisj` is:

```
xdisj :: T → T → T
xdisj E _  = E
xdisj _ E  = E
xdisj x y | o_ x && o_ y = o (xdisj (o' x) (o' y))
xdisj x y | o_ x && i_ y = i (xdisj (o' x) (i' y))
xdisj x y | i_ x && o_ y = i (xdisj (i' x) (o' y))
xdisj x y | i_ x && i_ y = o (xdisj (i' x) (i' y))
```

As classical logic holds for known values, bitwise disjunction `disj` is implemented as a de Morgan equality:

```
disj :: T → T → T
disj x y = neg (conj (neg x) (neg y))
```

Bitwise implication (denoted `geq`) and equality (denoted `eq`) are implemented also like in classical logic:

```
geq, eq :: T → T → T
geq x y = neg (conj (neg x) y)

eq x y = conj (geq x y) (geq y x)
```

A few examples show them at work:

```
> neg E
E
> conj (t 9) (t 12)
V (W E []) []
> n it
7
> > neg (neg (t 1234))
W (V E []) [V E [],E,E,V E [],V E []]
> n it
1234
```

Note that, as for the bitwise operations in section 7, optimized "one block at a time" implementations are possible.

8. Approximating with low structural complexity hereditarily binary numbers

It is a well know fact that humans have a limited ability to precisely distinguish between large numbers relatively close in value. Likewise, computer arithmetic uses floating point numbers to approximate real numbers as needed for practical computations.

We have shown in [14] that hereditarily binary numbers can express compactly very large numbers that are linear combinations of towers of exponents of 2. An interesting question rises: can we approximate natural numbers with their counterparts having a smaller structural complexity?

The functions `inf1` and `sup1` provide such approximations. The function `inf1` approximates from below by turning the least significant block of i^k applications into o^k applications and then collapsing it with the neighboring blocks.

```
inf1 E = E
inf1 (V x []) = V x []
inf1 (V x [y]) = V (s (add x y)) []
```

```
inf1 (V x (y:z:xs)) = V (s (s (add (add x y) z))) xs
inf1 (W x []) = V x []
inf1 (W x (y:xs)) = V (s (add x y)) xs
```

Similarly, `sup1` approximates from above, by turning the least significant block of of o^k applications into i^k applications.

```
sup1 E = E
sup1 (V x []) = W x []
sup1 (V x (y:xs)) = W (s (add x y)) xs
sup1 (W x []) = W x []
sup1 (W x [y]) = W (s (add x y)) []
sup1 (W x (y:z:xs)) = W (s (s (add (add x y) z))) xs
```

The following holds:

PROPOSITION 8. $\forall x$ *inf1* $x \leq x \leq$ *sup1* x.

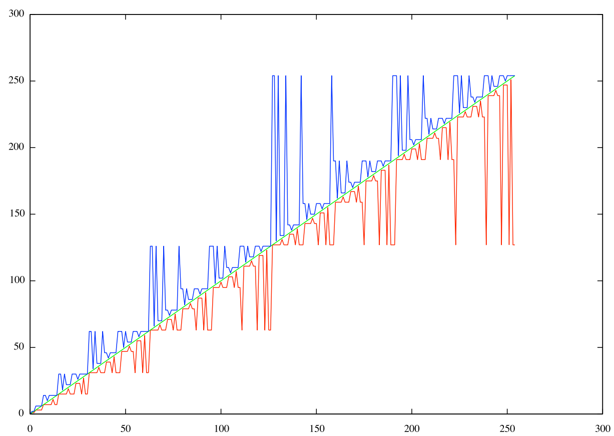


Figure 2: Plots corresponding to *one* application of `inf1` and `sup1`

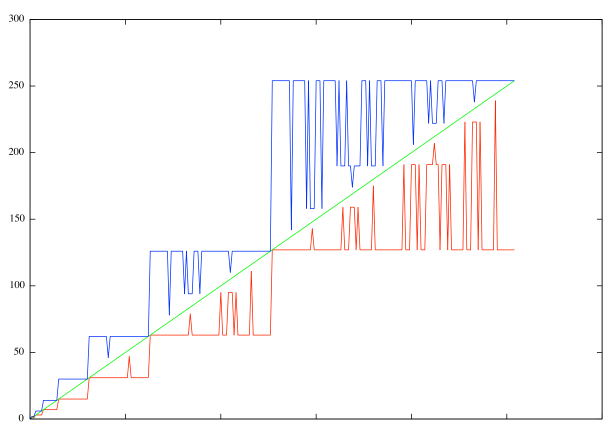


Figure 3: Plots showing *two* applications of `inf1` and `sup1`

By iterating `inf1` and tt sup1 one can further reduce the representation size of the approximations in exchange for accuracy. The following examples and figures 2 and 3 illustrate this for one and two applications of each function.

```
> map (n.inf1.t) [0..15]
[0,1,1,3,3,3,3,7,7,7,7,7,11,7,7,15]
> map (n.sup1.t) [0..15]
[0,2,2,6,6,6,6,14,14,10,14,14,14,14,14,30]
```

As `inf1` is non-increasing and and `sup1` is non-decreasing, we can compute a fixpoint for them as follows:

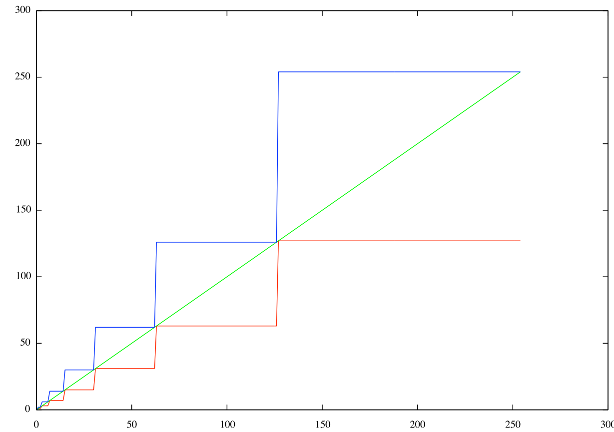


Figure 4: Plots showing fixpoints of `inf1` and `sup1`

```
fixpoint f x = r where
  p (a,b) = a/=b
  xs = iterate f x
  zs = zip xs (tail xs)
  ps = takeWhile p zs
  qs= snd (unzip ps)
  r = if []==qs then x else (last qs)
```

The following examples and Fig. 4 illustrate the stronger upper and lower bounds for these two functions.

```
> map (n.fixpoint inf1.t) [0..15]
[0,1,1,3,3,3,3,7,7,7,7,7,7,7,7,15]
> map (n.fixpoint sup1.t) [0..15]
[0,2,2,6,6,6,6,14,14,14,14,14,14,14,14,30]
```

9. Related work

This paper is a sequel to [14] where hereditary binary numbers are introduced with algorithms working "one block of iterated o and i operations at a time". In contrast to [14], where the focus is on deriving the arithmetic algorithms, this paper is about operations on and encodings of sparse/dense lists, sets and multisets, their hereditarily finite correspondents as well as bitvector boolean logic.

We also make use of the data transformation framework described in [12] that allows morphing bijectively between fundamental data types, except that our target this time is hereditarily binary natural numbers rather than their traditional bitstring based counterparts. This view is also somewhat close to key ideas behind the recent effort on homotopy type theory [16] that sees types that support "homotopy"-like type path transformations as equivalent.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [4] covering operations like *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.

In [17] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. By contrast to these non-canonical representations, each natural number is uniquely represented as a hereditarily binary number with the important consequence that their equality and comparison relations are decided simply by structural induction.

Variants of BDDs [1, 2] like Zero-Suppressed Binary Decision Diagrams [8], have been used for representing sparse sets of sparse bitvectors and their operations. By contrast, our hereditarily binary numbers provide at the same time efficient boolean operations on both sparse and dense sets, as well as the full spectrum of arithmetic operations.

10. Conclusion and Future Work

Together with [14] this paper is a step towards showing that hereditarily binary numbers offer a unique universal representation for both numeric and symbolic data types.

We have focused on natural numbers computations and datatypes as reducing everything else to them is fairly well known. For instance integer and rational number arithmetic can easily be reduced to natural number computations as we have illustrated, for the case of hereditarily binary numbers, with the Scala-based package at `https://code.google.com/p/giant-numbers/`. At the same time, symbolic data structures are easily mapped to natural numbers through the use of symbol tables.

As shown in [14], hereditarily binary numbers provide an interesting performance trade-off: in exchange for a small constant average slow down, they provide a constant-time exponent of two operation. Consequently, they favor by a super-exponential factor, arithmetic operations on numbers in neighborhoods of towers of exponents of two.

More importantly, as shown in this paper, hereditarily binary numbers provide a uniform mechanism for representing lists, multisets and sets of natural numbers through simple and efficiently computable bijections. In contrast to bitstring representations, these bijections are *size proportionate*. This property extends to hereditarily finite sets, multisets and lists, as well as term algebras, therefore providing a unique representation for key mathematical objects mapped to each other through a simple bijective data type transformation framework, defined through Haskell combinators.

Boolean operations specialized to hereditarily binary numbers have their complexity parameterized by their tree representation size that favors functions with uniform (sparse or dense) structure. We have illustrated their application to boolean evaluation as well as to a 3-valued alternative logic. The conditions for lower time and space complexity of hereditarily binary number representations are likely to apply to several other sparse/dense representations ranging from quad-trees to audio/video encoding formats.

In conclusion, *all this surprising versatility comes from the fact that our canonical tree representation, in contrast to the traditional binary representation, supports constant time and space application of exponents of two.*

Future work will focus on extending this framework to cover other important data types, with emphasis on graphs and exploration of various number representation-dependent algorithms that are likely to benefit from efficient operations on hereditarily binary numbers.

Among the applications that we plan to explore, bijective encodings of various objects built recursively from lists, sets and multisets ranging from term various algebras to sparse matrixes, boolean circuits and quad-trees. These data structures are likely to all benefit from sharing of immutable objects, when stored, as well as from the fact that sparse/dense objects, that can be encoded and decoded bijectively, have compact representations.

Acknowledgement

We thank the reviewers of `PPDP'2014` for their careful reading of the manuscript and their thoughtful comments and suggestions.

This research has been supported by NSF research grant 1423324.

References

[1] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C–35(8):677–691, 1986.

[2] R. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pages 236 – 243, 1995.

[3] P. Cegielski and D. Richard. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science*, 222(1-2):55–75, 1999.

[4] D. E. Knuth. Mathematics and Computer Science: Coping with Finiteness. *Science*, 194(4271):1235 –1242, 1976.

[5] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009. ISBN 0321580508, 9780321580504.

[6] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1993. ISBN 0-387-94053-7.

[7] M. Lisi. Some remarks on the Cantor pairing function. *Le Matematiche*, 62(1), 2007. ISSN 2037-5298. URL `http://www.dmi.unict.it/ojs/index.php/lematematiche/article/view/14`.

[8] S.-i. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *30-th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[9] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[10] R. P. Stanley. *Enumerative Combinatorics*. Wadsworth Publ. Co., Belmont, CA, USA, 1986. ISBN 0-534-06546-5.

[11] P. Tarau. A Unified Formal Description of Arithmetic and Set Theoretical Data Types. In S. Auxtier, editor, *Intelligent Computer Mathematics, 17th Symposium, Calculemus 2010, 9th International Conference AISC/Calculemus/MKM 2010*, pages 247–261, Paris, July 2010. Springer, LNAI 6167.

[12] P. Tarau. Declarative Modeling of Finite Mathematics. In *PPDP '10: Proceedings of the 12th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 131–142, New York, NY, USA, 2010. ACM.

[13] P. Tarau. Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) . *Theory and Practice of Logic Programming*, 13(4-5):847–861, 2013.

[14] P. Tarau and B. Buckles. Arithmetic Algorithms for Hereditarily Binary Natural Numbers. In *Proceedings of SAC'14, ACM Symposium on Applied Computing, PL track*, Gyeongju, Korea, Mar. 2014. ACM.

[15] P. Tarau and B. Luderman. Boolean Evaluation with a Pairing and Unpairing Function. In A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. Watt, and D. Zaharie, editors, *Proceedings of SYNASC 2012*, pages 384–392. IEEE, Jan. 2013.

[16] The Univalent Foundations Program. *Homotopy Type Theory*. Institute of Advanced Studies, Princeton, 2013. `http://homotopytypetheory.org/2013/06/20/the-hott-book/`.

[17] J. Vuillemin. Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 7 –14, June 2009.

Appendix

A subset of Haskell as an executable function notation

We mention, for the benefit of the reader unfamiliar with Haskell, that a notation like `f x y` stands for $f(x, y)$, `[t]` represents sequences of type `t` and a type declaration like `f :: s -> t -> u` stands for a function $f : s \times t \to u$ (modulo Haskell's "currying" operation, given the isomorphism between the function spaces $s \times t \to u$ and $s \to t \to u$).

Our Haskell functions are always represented as sequences of recursive equations guided by pattern matching, conditional to constraints (boolean relations *following the "|" symbol and before the "=" symbol*) in an equation.

Locally scoped helper functions are defined in Haskell after the "`where`" keyword, using the same equational style. The composition of functions `f` and `g` is denoted `f . g`. It is customary in Haskell to write $f = g$ instead of $f\ x = g\ x$ ("point-free" notation). We make some use of Haskell's "call-by-need" evaluation that allows us to work with infinite sequences, like the `[0..]` infinite list notation, as well as higher order functions (having other functions as arguments). Note also that the result of the last evaluation is stored in the special Haskell variable `it`.

By restricting ourselves to this *"Haskell - -"* subset, our code can also be easily transliterated into a system of rewriting rules, other pattern-based functional languages as well as deterministic Horn Clauses.

Haskell types for operations on Hereditarily Binary Numbers from [14]

As a convenience to the reader, we include here the Haskell type definitions inferred from the code in the paper [14], a literate program explaining the code in full detail (see draft available from `http://www.cse.unt.edu/~tarau/research/2014/HBin.pdf`).

We have also included the definition of functions like successor `s` and predecessor `s'` that have been used in [14] to prove the constant average time complexity and iterated logarithm worst case complexity of the exponent of two operation `exp2`.

```
-- the type of tree-represented natural numbers
data T = E | V T [T] | W T [T] deriving (Eq,Show,Read)

-- from tree-represented to bit-represented naturals
n :: T -> N

-- from bit-represented to tree-represented naturals
t :: N -> T

-- successor and predecessor
s, s' :: T -> T

s E = V E []
s (V E []) =  W E []
s (V E (x:xs)) =  W (s x) xs
s (V z xs) = W E (s' z : xs)
s (W z [])  = V (s z) []
s (W z [E]) = V z [E]
s (W z (E:y:ys)) = V z (s y:ys)
s (W z (x:xs)) = V z (E:s' x:xs)

s' (V E []) = E
s' (V z []) = W (s' z) []
s' (V z [E]) =  W z [E]
s' (V z (E:x:xs)) =  W z (s x:xs)
s' (V z (x:xs)) =  W z (E:s' x:xs)
s' (W E []) = V E []
s' (W E (x:xs)) = V (s x) xs
s' (W z xs) = V E (s' z:xs)

-- smart constructors adding bijective base 2 "digits"
o, i :: T -> T

o E = V E []
o (V x xs) = V (s x) xs
o (W x xs) = V E (x:xs)

i E = W E []
i (V x xs) = W E (x:xs)
i (W x xs) = W (s x) xs

-- deconstructors removing bijective base 2 "digits"
o',i' :: T -> T

o' (V E []) = E
o' (V E (x:xs)) = W x xs
o' (V x xs) = V (s' x) xs

i' (W E []) = E
i' (W E (x:xs)) = V x xs
i' (W x xs) = W (s' x) xs

-- recognizers for null, odd and positive even naturals
e_,o_,i_ :: T -> Bool

e_ E = True
e_ _ = False

o_ (V _ _ ) = True
o_ _ = False

i_ (W _ _ ) = True
i_ _ = False

-- double, half and exponent of 2
db, hf, exp2 :: T -> T

db = s' . o

hf = o' . s

exp2 E = V E []
exp2 x = s (V (s' x) [])

-- adding a block of o or i applications
otimes, itimes :: T -> T -> T
itimes :: T -> T -> T

-- addition and subtraction
add, sub :: T -> T -> T

-- helpers for addition and subtraction
oplus :: T -> T -> T -> T
oiplus :: T -> T -> T -> T
iplus :: T -> T -> T -> T
ominus :: T -> T -> T -> T
iminus :: T -> T -> T -> T
oiminus :: T -> T -> T -> T
iominus :: T -> T -> T -> T
osplit :: T -> (T, T)
isplit :: T -> (T, T)

-- comparison
cmp :: T -> T -> Ordering

-- dual of a tree-represented natural
dual :: T -> T

-- bitsize of of a tree-represented natural
bitsize :: T -> T

-- integer logarithm in base 2
ilog2 :: T -> T

-- shift operations
leftshiftBy :: T -> T -> T
rightshiftBy :: T -> T -> T

-- tree size, computed as a tree-represented natural
tsize :: T -> T

-- length of a list, as a tree-represented natural
len :: [a]->T
```