Bijective Collection Encodings and Boolean Operations with Hereditarily Binary Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cse.unt.edu

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 of several data types, that we "virtualize" through a generic data type transformation framework.
After extending the arithmetic operations on hereditarily binary numbers with boolean operations, we use the to perform computations with bitvectors and sets.
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 [1][1] 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. In [1] we describe 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.

At the same time, worst case and average case complexity of arithmetic operations is within constant factor of their bitstring counterparts.

[1] An extended draft version of [1] is available at at the *arxiv* repository [2].

The main focus of this paper is 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 with the usual binary notation. Consequently, bijections of hereditarily finite sets to $\mathbb{N}$ result in size-proportionate encodings when computed with hereditarily binary numbers.

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

The paper is organized as follows. Section 2 overviews basic definitions for *hereditarily binary numbers* and summarizes some of their properties, following [1]. 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 5 introduces bitvector operations using hereditarily binary numbers and their corresponding set equivalents as well as their application to evaluation of boolean formulas. Section 6 discusses related work. Section 7 concludes the paper. The Appendix shows applications to boolean evaluation and an alternative 3-valued logic implementation of bitvector operations.

We have adopted a *literate programming* style, i.e. the code contained in the paper forms a Haskell module (tested with ghc 7.6.3), available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/hbs.hs`. It imports the code from [1], also available at `http://logic.cse.unt.edu/tarau/research/2013/hbin.hs`.

2 Hereditarily Binary Numbers

We will summarize, following [1], the basic concepts behind *hereditarily binary numbers*. Through the paper, we denote $\mathbb{N}$ the 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 [3] 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 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. 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 [1] 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 [1] by the Haskell declaration:*

```
data T = E | V T [T]  | W T [T]
```

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

The intuition behind 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`

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

In [1] 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 [1] 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.

$$2^{2^{16}} - 1 \to \texttt{V (V (V (V (V E[])[])[])[])[]} \to 2^{2^{2^{2^{2^{0+1}-1+1}-1+1}-1+1}-1+1} - 1$$

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

In [1] 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 of `o` and `i` are given in [1], 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).

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. Interestingly, natural number encodings for sparse instances of these collections will have space-efficient representations as natural numbers of type $\mathbb{T}$, in contrast with 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 [1] for the definitions of functions `s, 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.

Proof. It is proven in [1] that `o, o', i, i'`,`o_` and `i_` have the same worst case and average complexity as `s` and `s'`, i.e, constant average and $O(log^*(bitsize))$ worst case. 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 z | e_ z = []
to_list z = x : to_list y where (x,y) = decons z

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

3.2 Bijections between sequences, sets and multisets

Incremental sums are used to transform arbitrary lists to multisets and sets, inverted by pairwise differences.

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

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

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.

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

to_set = list2set . to_list
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 [4, 5] 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 that this x = to that (from this x)
```

We can now define our "virtual types". 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 = 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:

```
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`.

```
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 lists, multisets and sets of natural numbers and natural numbers, based on separating `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`.

```
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 [1] `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' x | e_ x = []
to_list' x = hd : (to_list' tl) where (hd,tl)=decons' x

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
```

The following holds:

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

The following example illustrates their work:

```
> 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' = 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
```

4 Hereditarily Finite Lists, Sets and Multisets

We will use the data type `H` to support our hereditarily finite collection types.

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

The function `nt2f` 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 `nt2f` 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 = 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 $\mathbb{T}$

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

one can notice that it is identical to `as hfs bitnat`:

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

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

```
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 $\mathbb{T}$ by 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 [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.*

5 Bitwise operations and their applications

We implement bitvector operations (also seen as efficient bitset operations) to work "one block of o^n or i^m 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 are tractable with our trees, provided that they have a relatively small structural complexity, despite their large bitsize.

5.1 Boolean operations on tree-represented bitvectors

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

```
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. The base cases of `bwOr` are:

```
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 [1], 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 [1], where definitions of `otimes` and `itimes` are also given. We remind that `otimes` and `itimes`, defined in [1], 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 [1], 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 [1]), 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 k x = sub y x where y = s' (exp2 k)
```

The function bitwiseAndNot, combines bitwiseOr and 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 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 x y = if LT==cmp x y then y else x
```

The function bitsOf adapts the integer base-2 logarithm ilog2, (defined in [1]), to compute the number of bits of a bitvector.

```
bitsOf E = s E
bitsOf x = s (ilog2 x)
```

Bitwise conjunction bitwiseAnd is similar, relying also on bitsOf:

```
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 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 [1]) computes the structural complexity of a term, defined as the size of tree representation.

5.2 Set operations

With help from the data transformation operation lend2 we can use bitvectors for set operation:

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

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.

6 Related work

This paper is a sequel to [1] where hereditary binary numbers are introduced with algorithms working "one block of iterated o and i operations at a time". By contrast to [1], 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.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *up-arrow* notation [6] 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 [7] 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 [8, 9] like Zero-Suppressed Binary Decision Diagrams [10], 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.

7 Conclusion and Future Work

We have provided a uniform mechanism for representing lists, multisets and sets of integers as hereditarily binary numbers through simple and efficiently computable bijections. By contrast to bitstring representations, these bijections turned out to be size proportionate. This property has extended to hereditarily finite sets, multisets and lists, therefore providing a unique representation for key mathematical objects that we have mapped to each other through a simple bijective data type transformation framework, defined through Haskell combinators.

Boolean operations specialized to hereditarily binary numbers have shown that their complexity can be seen as parameterized by their structural complexity (tree representation size) that favors functions with uniform (sparse or dense) structure, not unlikely to occur in practical problems. The same is likely to apply to several other sparse/dense representations ranging from quad-trees to audio/video encoding formats.

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.

References

1. Tarau, P., Buckles, B.: Arithmetic Algorithms for Hereditarily Binary Natural Numbers. In: Proceedings of SAC'14, ACM Symposium on Applied Computing, PL track, Gyeongju, Korea, ACM (March 2014)
2. Tarau, P.: Arithmetic Algorithms for Hereditarily Binary Natural Numbers (June 2013) http://arxiv.org/abs/1306.1128.
3. Salomaa, A.: Formal Languages. Academic Press, New York (1973)
4. Tarau, P.: A Unified Formal Description of Arithmetic and Set Theoretical Data Types. In Auxtier, S., ed.: Intelligent Computer Mathematics, 17th Symposium, Calculemus 2010, 9th International Conference AISC/Calculemus/MKM 2010, Paris, Springer, LNAI 6167 (July 2010) 247–261
5. Tarau, P.: "Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. Complex Systems (18) (2010) 475–493
6. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
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
8. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **C–35**(8) (1986) 677–691
9. Bryant, R.: 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. (1995) 236 – 243
10. Minato, S.i.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. 30-th ACM/IEEE Design Automation Conference (1993) 272–277
11. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional (2009)
12. Tarau, P., Luderman, B.: Boolean Evaluation with a Pairing and Unpairing Function. In Voronkov, A., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S., Zaharie, D., eds.: Proceedings of SYNASC 2012, IEEE (January 2013) 384–392

Appendix

Boolean formula evaluation

Besides definitions for the boolean functions, we also need projection variable $v(n,k)$ corresponding to column k of a truth table for a function with n variables. A compact formula for them, as given in [11] or [12], is

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

but we will compute them here as a concatenation of alternating blocks of 1 and 0 bits to take advantage of our efficient block operations.

```
v 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 = v (t 3) (t 0)
  v1 = v (t 3) (t 1)
  v2 = v (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]`.

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

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. Negation `neg` can be implemented as the constant time `dual` operation, defined in [1], 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 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 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 defined values, bitwise disjunction `disj` is implemented as a de Morgan equality:

```
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 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 5, optimized "one block at a time" implementations are possible.