Bijective Collection Encodings and Boolean Operations with Hereditarily Binary Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PPDP'2014

– research supported by NSF grant 1423324 –

Overview

- our tree-based *hereditarily binary numbers* apply recursively a run-length compression mechanism on iterated function applications seen as “digits”
- 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
- in this paper we apply hereditarily binary numbers to derive:
  1. compact representations for “structurally simple” (sparse or dense) lists, sets and multisets, as well as their hereditarily finite counterparts
  2. bijective size-proportionate Gödel numberings for several data types “virtualized” through a generic data type transformation framework
  3. a size-proportionate Gödel numbering scheme for term algebras
  4. boolean operations
  5. operations on bitvectors and sets

Outline

Bijective base-2 numbers as iterated function applications

Natural numbers can be seen as iterated applications of the functions

- $o(x) = 2x + 1$
- $i(x) = 2x + 2$

corresponding the so called *bijective base-2* representation.

- $1 = o(0)$,
- $2 = i(0)$,
- $3 = o(o(0))$,
- $4 = i(o(0))$,
- $5 = o(i(0))$

Iterated applications of o and i: some useful identities

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

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

and in particular

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

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

Hereditarily binary numbers

Hereditarily binary numbers are defined as the Haskell type $\mathbb{T}$:

```
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 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
- 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
- the same principle is applied recursively for the counters, until the empty sequence is reached
- note: `x` counts `x+1` applications, as we start at `0`

The arithmetic interpretation of hereditarily binary numbers

Definition

The bijection $n : \mathbb{T} \to \mathbb{N}$ defines the unique natural number associated to a term of type $\mathbb{T}$. Its inverse is denoted $t : \mathbb{N} \to \mathbb{T}$.

$$n(t) = \begin{cases} 0 & \text{if } t = \mathtt{E}, \\ 2^{n(x)+1} - 1 & \text{if } t = \mathtt{V\ x\ []}, \\ (n(u)+1)2^{n(x)+1} - 1 & \text{if } t = \mathtt{V\ x\ (y:xs)} \text{ and } u = \mathtt{W\ y\ xs}, \\ 2^{n(x)+2} - 2 & \text{if } t = \mathtt{W\ x\ []}, \\ (n(u)+2)2^{n(x)+1} - 2 & \text{if } t = \mathtt{W\ x\ (y:xs)} \text{ and } u = \mathtt{V\ y\ xs}. \end{cases} \tag{5}$$

The computation of $n(\mathtt{W\ (V\ E\ [])\ [E,E,E]})$ expands to $(((2^{0+1} - 1 + 2)2^{0+1} - 2 + 1)2^{0+1} - 1 + 2)2^{2^{0+1}-1+1} - 2 = 42$.

Examples

- each term canonically represents the corresponding natural number
- the first few natural numbers are:

```
0 = n E
1 = n (V E [])
2 = n (W E [])
3 = n (V (V E []) [])
4 = n (W E [E])
5 = n (V E [E])
```

An overview of constant average time and worst case constant or log^* time operations with hereditarily binary numbers

- introduced in our ACM SAC'14 paper:
- mutually recursive successor s and predecessor s'
- defined on top of s and s':
  - $o(x) = 2x + 1$ and $i(x) = 2x + 2$
  - their inverses o' and i'
  - recognizers of odd and even numbers *o_* and *i_*
  - double *db* and its left inverse *hf*
  - power of two *exp*2
- $\Rightarrow$ efficient computations with towers of exponents and numbers in their "neighborhoud"
- $\Rightarrow$ efficient computations with sparse numbers (with a lot of 0s) or dense numbers (with a lot of 1s)

Towers of exponents can grow tall, provideded they are finite :-)

$$2^{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}$$

A bijective encoding of lists of hereditarily binary numbers

- we split a natural number in blocks of *o* and *i* applications
- we also need to encode and remember the parity (at the end of the list)
- we first implement `cons` and `decons` and then we iterate them
- !!! they have the same complexity as successor `s` and predecessor `s'` !!!

```
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)
```

Iterating `cons` and `decons`

- they define a bijection between hereditarily binary numbers and lists of hereditarily binary numbers
- cons and decons are average constant and worst case log_2^*
- $\Rightarrow$ complexity of `to_list` and `from_list` is proportional to the number of blocks of *o* and *i* applications

```
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)
```

Bijections between lists, sets and multisets

- a multiset like $[4,4,1,3,3,3]$ could be represented canonically by first ordering it as $[1,3,3,3,4,4]$ and then computing the differences between consecutive elements:
- $[x_0, x_1 \ldots x_i, x_{i+1} \ldots] \rightarrow [x_0, x_1 - x_0, \ldots x_{i+1} - x_i \ldots]$
- this gives $[1,2,0,0,1,0]$, with the first (1) followed by $[2,0,0,1,0]$

```
list2mset, mset2list, list2set, set2list :: [T] → [T]

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

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

- for sets, we ensure all elements increase by 1, so they end up all different

```
list2set = (map s') . list2mset . (map s)
set2list = (map s') . mset2list . (map s)
```

Virtual types through bijective data transformations

```
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 define “virtual types” as bijections to a “hub"
- our tree-based natural numbers provide the hub `nat`

```
nat = Iso id id
```

- the collection types for lists, sets and multisets are bijections to `nat`

```
list, mset, set :: Iso [T] T
list = Iso from_list to_list
mset = Iso from_mset to_mset
set = Iso from_set to_set
```

Morphing between virtual types

```
> 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
```

combinators that borrow operations from another virtual type

```
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
```

ex: sets will borrow bitwise boolean operations

Hereditarily finite lists, sets and multisets, generically

```
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))
```

- 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)
```

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

Virtual types associated to hereditarily finite collection types

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)
```

```
> 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
```

Proposition

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

A bijective size-proportionate encoding of term algebras

- devising a Gödel numbering scheme for term algebras that is both size-proportionate and bijective is a difficult task
- it involves a fairly sophisticated ranking/unranking algorithm for Catalan families in combination with a generalization of Cantor's pairing function to tuples (see our ICLP'2013 paper)
- 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
- $\Rightarrow$ bijective and size-proportionate Gödel numberings of tree-like structures (in particular term algebras) becomes straightforward

A bijective size-proportionate Gödel numbering of term algebras

```
data Term a = Var a | Const a | Fun a [Term a]

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)

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)
```

Examples

```
> 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
```

- 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.
- 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

Automorphisms of $\mathbb{N}$ from automorphisms of $\mathbb{T}$

- A simple bijection $\mathbb{T} \to \mathbb{T}$ is provided by the `dual` operation that flips toplevel constructors `V` and `W`
- it reinterprets all o operations as $\imath$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:

```
> 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 concept of asymmetric duality

- 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 (see upcoming ICTAC'14 paper on their arithmetic)

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

tdual :: T → T
tdual = borrow1 hfl hdual nat
```

- also an *involution*

```
> 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]
```

Example

- the trees of type $\mathbb{T}$ associated to random natural numbers are much wider than tall.
- the involution `tdual` flips between relatively small random numbers (with high Kolmogorov complexity) and giant numbers with a regular structure

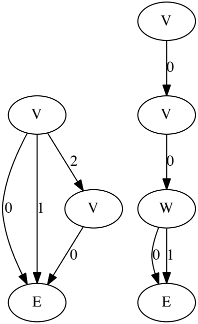

Figure: Duals, with trees folded into DAGs, arcs marked with order of children: *on the left,* `t 17` and *on the right,* `tdual of (t 17) = t 4294967295`

Bitvector operations

- we implement bitvector operations to work “one block of o^n or i^n applications at a time”
- target: large but sparse boolean formulas
- evaluate such formulas “all value-combinations at a time” when represented as bitvectors of size 2^{2^n}
- such operations will be tractable with our trees, provided that they have a relatively small representation size despite their large bitsize
- main idea of the algorithms in the paper:
  1. split and align block fragments of the same size
  2. perform the boolean operation between the blocks as if they were single bits
  3. fuse the resulting blocks into larger blocks when possible

Examples

- our bitwise operations can be efficiently applied to giant numbers
- for instance $(2^{2^{12345}} + 1)$ XOR $(2^{2^{6789}} - 1)$ is computed as:

```
> 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) -- tsize computes the size of the tree
39
```

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
```

- example:

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

- sparse or dense sets containing very large sparse or dense elements benefit significantly from this encoding if despite the large bitsizes involved they are represented as compact trees

Boolean formula evaluation

- *var*(n,k): column k of a truth table for a function with n variables
- Knuth gives a compact formula for them:

$$var(n,k) = (2^{2^n} - 1) \;/\; (2^{2^{n-k-1}} + 1) \tag{6}$$

- instead of doing the division, we compute them as a concatenation of alternating blocks of 1 and 0 bits to take advantage of our efficient block operations
- *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

Hereditarily finite natural numbers as circuits

- bijective base-2 representation of natural numbers can be seen as successive listings of the columns in truth tables for $0, 1, \ldots, n$-argument boolean functions

```
(7, [0,0,0])
(8, [1,0,0])
(9, [0,1,0])
...
(13,[0,1,1])
(14,[1,1,1])
```

- the bijective base-2 representation of natural numbers in $[2^n - 1 \mathrel{..} 2^{n+1} - 2]$ describes the inputs in the truth table of a n-argument boolean function
- $\Rightarrow$ the functions `bitval` and `nthBit` in the paper navigate to a given bit through the tree

Conclusion

- we have shown previously that hereditarily binary numbers favor by a super-exponential factor, arithmetic operations on numbers in neighborhoods of towers of exponents of two
- we show in this paper that hereditarily binary numbers also 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 an application, we derive a size-proportionate bijective Gödel numbering scheme for term algebras
- $\Rightarrow$ hereditarily binary numbers provide a unique representation for key mathematical objects mapped to each other through bijections between “virtual data types” expressed in terms of Haskell combinators

Links

- the paper is a literate program, our Haskell code is at `http://www.cse.unt.edu/~tarau/research/2014/HBS.hs`
- it imports code from the our ACM SAC'14 paper at `http://www.cse.unt.edu/~tarau/research/2014/HBin.hs`
- a draft version of the ACM SAC'14 paper is at `http://www.cse.unt.edu/~tarau/research/2014/HBin.pdf`
- new arithmetic and number theoretical algorithms with HBNs at `http://www.cse.unt.edu/~tarau/research/2014/hbinx.pdf`
- an alternative Scala based implementation of HBNs is at at: `http:/code.google.com/p/giant-numbers/`