Computations and Data Type Encodings with Hereditarily Binary Natural Numbers

Paul Tarau

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

Abstract. Can we do arithmetic in a completely different way, with a radically different data structure? Could this approach provide practical benefits, like operations on giant numbers while having an average performance similar to traditional bitstring representations?
While answering these questions positively, our tree based representation described in this paper, *hereditarily binary numbers*, comes with a few extra benefits: it compresses giant numbers such that, for instance, the largest known prime number as well as its related perfect number are represented as trees of small sizes. The same also applies to Fermat numbers and important computations like exponentiation of two become constant time operations.
At the same time, succinct representations of sparse sets, multisets and sequences become possible through bijections to our tree-represented natural numbers.
Keywords: *hereditary numbering systems, compressed number representations, arithmetic computations with giant numbers, tree-based numbering systems, compact representation of large Mersenne, Cullen, Proth, Woodall, Sophie Germaine primes, perfect numbers and Fermat numbers, compact bijective encodings of sparse data structures*

1 Introduction

If extraterrestrials were to do arithmetic computations, would one assume that their numbering system should look the same? At a first thought, one might be inclined to think so, maybe with the exception of the actual symbols used to denote digits or the base of the system likely to match numbers of fingers or some other anthropocentric criteria. After some thinking about the endless diversity of the universe (or unconventional models of Peano's axioms), one might also consider the possibility that the departure from our well-known number representations could be more significant.

With more terrestrial uses in mind, one would simply ask: is it possible to do arithmetic differently, possibly including giant numbers and a radically different underlying data structure, while maintaining the same fundamentals – addition, multiplication, exponentiation, primality, Peano's axioms behaving in the same way? Moreover, can such exotic arithmetic be as efficient as binary arithmetic, while possibly supporting massively parallel execution?

We have shown in [1] that type classes and polymorphism can be used to share fundamental operations between natural numbers, finite sequences, sets and multisets. As a consequence of this line of research, we have discovered that it is possible to transport the recursion equations describing binary number arithmetics on natural numbers to various tree types.

However, the representation proposed in this paper is *different*. It is arguably simpler, more flexible and likely to support parallel execution of operations. It also allows to easily compute average and worse case complexity bounds. We will describe it in full detail in the next sections, but for the curious reader, it is essentially a *recursively self-similar run-length encoding of bijective base-2 digits*.

The paper is organized as follows. Section 2 describes our type class used to share generic properties of natural numbers. Section 3 describes binary arithmetic operations that are specialized in section 4 to our compressed tree representation. Section 5 describes efficient tree-representations of some important number-theoretical entities like Mersenne, Fermat and perfect numbers. Section 6 shows that sparse sets, multisets and lists have succinct tree representation. Section 7 describes generic isomorphisms between data types, centered around transformations of instances of our type class to their corresponding sets, multisets and lists. Section 8 shows interesting complexity reductions in other computations and section 9 compares the performance of our tree-representation with conventional ones. Section 10 discusses related work. Section 11 concludes the paper and discusses future work.

To provide a concise view of our new number representation and facilitate its comparison with conventional binary numbers, we will use Haskell *type classes* as a precise means to provide an *executable* specification. We have adopted a literate programming style, i.e. the code contained in the paper forms a self-contained Haskell module (tested with ghc 7.4.1), also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/giant.hs` . Alternatively, a Scala package implementing the same tree-based computations is available from `http://code.google.com/p/giant-numbers/`. We hope that this will encourage the reader to experiment interactively and validate the technical correctness of our claims.

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 sets of recursive equations guided by pattern matching, conditional to constraints (simple arithmetic relations following `|` and before the `=` symbol). 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 also customary in Haskell, when defining functions in an equational style (using `=`) to write $f = g$ instead of $f\ x = g\ x$ ("point-free" notation). The use of Haskell's "call-by-need" evaluation allows us to work with infinite sequences, like the `[0..]` infinite list notation, corresponding to the set $\mathbb{N}$ itself.

Our literate Haskell program is organized as the module `Giant` depending on a few packages, as follows:

```
{-# LANGUAGE NoMonomorphismRestriction #-}
-- needed to define toplevel isomorphisms
module Giant where
-- cabal install data-ordlist is required before importing this
import Data.List.Ordered
import Data.List hiding (unionBy)
import System.CPUTime
```

2 A type class for generic arithmetic computations

2.1 Bijective base-2 numbers: computing with $\{0,1\}^*$

Conventional binary representation of a natural number n describes it as the value of a polynomial $P(x) = \sum_{i=0}^{k} a_i x^i$ with digits $a_i \in \{0,1\}$ for x=2, i.e. $n = P(2)$. If one rewrites $P(x)$ using Horner's scheme as $Q(x) = a_0 + x(a_1 + x(a_2 + ...))$, n can be represented as an iterated application of a function $f(a,x,v) = a + xv$ which can be further specialized as a sequence of applications of two functions $f_0(v) = 2v$ and $f_1(v) = 1 + 2v$, corresponding to the encoding of n as the sequence of binary digits $a_0, a_1, \ldots, a_k$. This encoding is clearly not bijective as a function to $\{0,1\}^*$. For instance 1, 10, 100 etc. would all correspond to $n = 1$. As a fix, the functions $o(v) = 1 + 2v$ and $i(v) = 2 + 2v$ can be used instead, resulting in the so called *bijective base-2* representation [2], together with the convention that 0 is represented as the empty sequence. 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. Following [1] we will next develop arithmetic computations using a natural abstraction of this number representation in the form of a Haskell type class.

2.2 Sharing "axiomatizations" – computing generically with a type class –

Haskell's *type classes* [3] are a good approximation of axiom systems as they describe properties and operations generically i.e. in terms of their action on objects of a parametric type. Haskell's type *instances* approximate *interpretations* [4] of such axiomatizations by providing implementations of the primitive operations, with the added benefit of refining and possibly overriding derived operations with more efficient equivalents.

We start by defining a type class that abstracts away properties of binary representations of natural numbers.

The class `N` assumes only a theory of structural equality (as implemented by the class `Eq` in Haskell). It implements a representation-independent abstraction of natural numbers, allowing us to compare our tree representation with "ordinary" natural numbers represented as non-negative arbitrary large `Integers` in Haskell, as well as with a binary representation using bijective base-2 [2].

```
class Eq n ⇒ N n where
```

An instance of this class is required to implement the following 6 primitive operations:

```
  e :: n
  o,o',i,i' :: n→n
  o_ :: n→Bool
```

The constant function `e` can be seen as representing the empty sequence of binary digits. With the usual representation of natural numbers, `e` will be interpreted as `0`. The constructors `o` and `i` can be seen as applying a function that adds a `0` or `1` digit to a binary string on $\{0,1\}^*$. The deconstructors `o'` and `i'` undo these operations by removing the corresponding digit. The recognizer `o_` detects that the constructor `o` is the last one applied, i.e. that the"string ends with the `0` symbol. It will be interpreted on $\mathbb{N}$ as a recognizer of odd numbers.

This type class also endows its instances with generic implementations of the following derived operations:

```
  e_,i_ :: n→Bool
  e_ x = x == e
  i_ x = not (e_ x || o_ x)
```

Note that structural equality is used implicitly in the definition of the recognizer predicate for empty sequences `e_` and the domain is exhausted by the three recognizers in the definition of the recognizer `i_` representing even positive numbers in bijective base 2.

2.3 Successor and predecessor, generically

Successor `s` and predecessor `s'` functions are implemented in terms of these operations as follows:

```
  s,s' :: n→n

  s x | e_ x = o x
  s x | o_ x = i (o' x)
  s x | i_ x = o (s (i' x))

  s' x | x == o e = e
  s' x | i_ x = o (i' x)
  s' x | o_ x = i (s' (o' x))
```

By looking at the code, one might notice that our generic definitions of operations mimic recognizers, constructors and destructors for bijective base-2 numbers, i.e. sequences in the language $\{0,1\}^*$, similar to binary numbers, except that 0 is represented as the empty sequence and left-delimiting by 1 is omitted.

Proposition 1 *Assuming average constant time for recognizers, constructors and destructors* `e_, _o, _i, o, i, o', i'`*, successor and predecessor* `s` *and* `s'` *are also average constant time.*

Proof. Clearly the first two rules are constant time for both `s` and `s'` as they do not make recursive calls. To show that the third rule applies recursion a constant number of times on the average, we observe that the recursion steps are exactly given by the number of 0s or 1s that a (bijective base-2 number) ends with. As only half of them end with a 0 and another half of those end with another 0 etc. one can see that the average number of 0s is bounded by $\frac{1}{2} + \frac{1}{4} + \ldots = 1$. The same reasoning applies to the average number of 1s a number can end with.

The infinite stream of generic natural numbers is generated by iterating over the successor operation `s`:

```
allFrom :: n→[n]
allFrom x = iterate s x
```

3 Efficient arithmetic operations, generically

We will first show that all fundamental arithmetic operations can be described in this abstract, representation-independent framework. This will make possible creating instances that, on top of symbolic tree representations, provide implementations of these operations with asymptotic efficiency comparable to the usual bitstring operations.

3.1 Addition and subtraction

We start with addition (`add`) and subtraction (`sub`):

```
add :: n→n→n
add x y | e_ x = y
add x y | e_ y  = x
add x y | o_ x && o_ y = i (add (o' x) (o' y))
add x y | o_ x && i_ y = o (s (add (o' x) (i' y)))
add x y | i_ x && o_ y = o (s (add (i' x) (o' y)))
add x y | i_ x && i_ y = i (s (add (i' x) (i' y)))

sub :: n→n→n
sub x y | e_ y = x
sub y x | o_ y && o_ x = s' (o (sub (o' y) (o' x)))
sub y x | o_ y && i_ x = s' (s' (o (sub (o' y) (i' x))))
sub y x | i_ y && o_ x = o (sub (i' y) (o' x))
sub y x | i_ y && i_ x = s' (o (sub (i' y) (i' x)))
```

It is easy to see that addition and subtraction are implemented generically, with asymptotic complexity proportional to the size of the operands.

3.2 Multiplication, double and half

Next, we define multiplication:

```
  mul :: n→n→n
  mul x _ | e_ x = e
  mul _ y | e_ y = e
  mul x y = s (m (s' x) (s' y)) where
    m x y | e_ x = y
    m x y | o_ x = o (m (o' x) y)
    m x y | i_ x = s (add y  (o (m (i' x) y)))
```

as well as the double of a number `db` and the half of an even number `hf`, having both simple expressions:

```
  db,hf :: n→n
  db = s' . o
  hf = o' .s
```

3.3 Two obvious instances of our type class

And for the reader curious by now about how this maps to "arithmetic as usual", here is an instance built around the (arbitrary length) `Integer` type, also usable as a witness on the time/space complexity of our operations.

```
instance N Integer where
  e = 0

  o_ x = odd x

  o  x = 2*x+1
  o' x | odd x && x >  0 = (x-1) `div` 2

  i  x = 2*x+2
  i' x | even x && x > 0 = (x-2) `div` 2
```

An instance mapping our abstract operations to actual constructors, follows, in the form of the datatype `B`

```
data B = B | O B | I B deriving (Show, Read, Eq)

instance N B where
  e = B
  o = O
  i = I

  o' (O x) = x
  i' (I x) = x

  o_ (O _) = True
  o_ _ = False
```

One can try out various operations on these instances:

```
*Giant> mul 10 5
```

```
50
*Giant> add (O B) (I (O B))
O (I B)
```

3.4 Representation Converters

It is convenient at this point, as we target a diversity of interpretations materialized as Haskell instances, to provide a polymorphic converter between two different instances of the type class `N` as well as their associated lists, implemented by structural recursion over the representation to convert. The function `view` allows importing a wrapped object of a different instance of `N`, generically.

```
view :: (N a,N b)⇒a→b
view x | e_ x = e
view x | o_ x = o (view (o' x))
view x | i_ x = i (view (i' x))
```

We can specialize `view` to provide conversions to our three data types, each denoted with the corresponding lower case letter, tt `t`, `b` and `n` for the usual natural numbers.

```
t :: (N n) ⇒ n → T
t = view

b :: (N n) ⇒ n → B
b = view

n :: (N n) ⇒ n → Integer
n = view
```

One can try them out as follows:

```
*Giant> t 42
W (V T []) [T,T,T]
*Giant> b it
I (I (O (I (O B))))
*Giant> n it
42
```

3.5 Defining a total order: comparison

Comparison provides the expected total order of $\mathbb{N}$ on our type class:

```
class N n ⇒ OrdN n where
  cmp :: n→n→Ordering
  cmp x y | e_ x && e_ y = EQ
  cmp x _ | e_ x = LT
  cmp _ y | e_ y = GT
  cmp x y | o_ x && o_ y = cmp (o' x) (o' y)
  cmp x y | i_ x && i_ y = cmp (i' x) (i' y)
  cmp x y | o_ x && i_ y = down (cmp (o' x) (i' y)) where
```

```
    down EQ = LT
    down r = r
  cmp x y | i_ x && o_ y = up (cmp (i' x) (o' y)) where
    up EQ = GT
    up r = r
```

And based on it one can define the minimum `min2` and maximum `max2` functions as follows:

```
  min2,max2 :: n→n→n
  min2 x y = if LT==cmp x y then x else y
  max2 x y = if LT==cmp x y then y else x
```

We need to also add instance declarations for our data types:

```
instance OrdN Integer
instance OrdN B
instance OrdN T
```

3.6 Power operations

Power is defined as follows:

```
class OrdN n ⇒ IntegralN n where
  pow :: n→n→n
  pow _ y | e_ y = o e
  pow x y | o_ y = mul x (pow (mul x x) (o' y))
  pow x y | i_ y = mul (mul x x) (pow (mul x x) (i' y))
```

together with more efficient special instances, exponent of 2 (`exp2`) and multiplication by a power of 2 (`leftshiftBy`):

```
  exp2 :: n→n
  exp2 x | e_ x = o e
  exp2 x = db (exp2 (s' x))

  leftshiftBy :: n→n→n
  leftshiftBy x y = mul y (exp2 x)
```

3.7 Division operations

Finally, division and remainder on $\mathbb{N}$ is a bit trickier but can be expressed generically as well:

```
  div_and_rem :: n→n→(n,n)

  div_and_rem x y | LT == cmp x y = (e,x)
  div_and_rem x y | not (e_ y) = (q,r) where
    (qt,rm) = divstep x y
    (z,r) = div_and_rem rm y
    q = add (exp2 qt) z
```

```
  divstep :: IntegralN n ⇒ n→n→(n,n)
  divstep n m = (q, sub n p) where
    q = try_to_double n m e
    p = mul (exp2 q) m

    try_to_double x y k =
      if (LT==cmp x y)
        then s' k
        else try_to_double x (db y) (s k)

divide,remainder :: n→n→n

divide n m = fst (div_and_rem n m)
remainder n m = snd (div_and_rem n m)
```

4 Computing with compressed tree representations

4.1 Hereditary Number Systems

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

This brings us to the idea of hereditary number systems. At our best knowledge the first instance of such a system is used in [5], by iterating the polynomial base-n notation to the exponents used in the notation. We next explore a hereditary number representation that implements the simple idea of representing the number of contiguous 0 or 1 digits in as bijective base-2 numbers, recursively.

4.2 Hereditarily binary numbers as a data type

We will use our type class-based framework to implemented a new, somewhat unusual instance, that results in the ability to do efficient arithmetic computations with trees.

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

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

The intuition behind this *disjoint union* type is the following:

– The type `T` corresponds to an empty sequence

– the type `V x xs` counts the number `x` of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications
– the type `W x xs` counts the number `x` of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications
– the same principle is applied recursively for the counters, until the empty sequence is reached

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

First we define the 6 primitive operations of type class `N` for instance `T`:

```
instance N T where
  e = T

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

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

  o' (V T []) = T
  o' (V T (x:xs)) = W x xs
  o' (V x xs) = V (s' x) xs

  i' (W T []) = T
  i' (W T (x:xs)) = V x xs
  i' (W x xs) = W (s' x) xs

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

4.3 A generic View of Hereditarily Binary Numbers

```
class IntegralN n ⇒ HeredN n where
  v :: (n,[n]) → n
  v (x,[]) = otimes (s x) e
  v (x,y:xs) = otimes (s x) (w (y,xs))

  w :: (n,[n]) → n
  w (x,[]) = itimes (s x) e
  w (x,y:xs) = itimes (s x) (v (y,xs))

  v' :: n→ (n,[n])
  v' z =  (s' (ocount z), f (otrim z)) where
    f y | e_ y = []
```

```
    f y =  (fst x:snd x) where x = w' y

  w' :: n→ (n,[n])
  w' z =  (s' (icount z), f (itrim z)) where
    f y | e_ y = []
    f y = fst x:snd x where x = v' y

  v_ :: n → Bool
  v_ z = e /= ocount z
  w_ :: n → Bool
  w_ z = e /= icount z
```

Implementing `v,w` and `v',w'` requires counting the number of applications of `o` and `i` provided by `ocount` and `icount`, as well trimming the applications of `o` and `i`, performed by `otrim` and `itrim` as well as applying a given times `o` and `i` operation `otimes` and `itimes`.

```
  ocount,icount,otrim,itrim :: n→n

  ocount x | o_ x = s (ocount (o' x))
  ocount _ = e

  icount x | i_ x = s (icount (i' x))
  icount _ = e

  otrim x | o_ x = otrim (o' x)
  otrim x = x

  itrim x | i_ x = itrim (i' x)
  itrim x = x

  otimes,itimes :: n→n→n

  otimes x y | e_ x = y
  otimes x y = otimes (s' x) (o y)

  itimes x y | e_ x = y
  itimes x y = itimes (s' x) (i y)
```

```
instance HeredN Integer
instance HeredN B
instance HeredN T where
  ocount (V x _) = s x
  ocount _ = T

  icount (W x _) = s x
  icount _ = T

  otrim (V _ []) = T
  otrim (V _ (x:xs)) = W x xs
```

```
  otrim x = x

  itrim (W _ []) = T
  itrim (W _ (x:xs)) = V x xs
  itrim x = x

  otimes T y = y
  otimes n T = V (s' n) []
  otimes n (V y ys) = V (add n y) ys
  otimes n (W y ys) = V (s' n) (y:ys)

  itimes T y = y
  itimes n T = W (s' n) []
  itimes n (W y ys) = W (add n y) ys
  itimes n (V y ys) = W (s' n) (y:ys)
```

Next, we override two operations involving exponents of `2` as follows:

```
instance IntegralN Integer
instance IntegralN B
instance IntegralN T where
  exp2 T = V T []
  exp2 x = s (V (s' x) [])

  leftshiftBy _ T = T
  leftshiftBy n k = s (otimes n (s' k))
```

The `leftshiftBy` function uses an efficient implementation, specialized for the type `T`, of the repeated application (`n` times) of constructor `o`, over the second argument of the function `otimes` defined by class `HeredN`:

Note that such overridings take advantage of the specific encoding, as a result of simple number theoretic observations. For instance, the operation `exp2` works in time proportional to `s` and `s'`. The more general `leftshiftBy` operation uses the fact that repeated application of the `o` operation, provides an efficient implementation of multiplication with an exponent of 2.

Proposition 2 *Let f^n denote application of function f k times. Let $o(x) = 2x+1$ and $i(x) = 2x+2$, $s(x) = x+1$ and $s'(x) = x-1$. Then $k > 0 \Rightarrow s(o^n(s'(k)) = k2^n$ and $k > 1 \Rightarrow s(s(i^n(s'(s'(k)))) = k2^n$. In particular, $s(o^n(0)) = 2^n$ and $s(s(i^n(0)) = 2^{n+1}$.*

Proof. By induction. Observe that for $n = 0, k > 0, s(o^0(s'(k)) = k2^0$ because $s(s'k)) = k$. Suppose that $P(n) : k > 0 \Rightarrow s(o^n(s'(k)) = k2^n$ holds. Then, assuming $k > 0$, P(n+1) follows, given that $s(o^{n+1}(s'(k)) = s(o^n(o(s'(k)))) = s(o^n(s'(2k)))) = 2k2^n = k2^{n+1}$. Similarly, the second part of the proposition also follows by induction on n.

Consequently, one could also define the folllowing `itimes`-based alternative to the function `leftshiftBy`:

```
leftshiftBy' _ T = T
```

```
leftshiftBy' T k = k
leftshiftBy' n (V T []) = s (V (s' n) [])
leftshiftBy' n k = s (s (itimes n (s' (s' k))))
```

While the correctness of operations like `exp2` and `leftshiftBy` follows immediately from Prop. 2, we will also illustrate their expected usage as follows:

```
*Giant> t 5
V T [T]
*Giant> exp2 it
W T [V (V T []) []]
*Giant> n it
32
*Giant> t 10
W (V T []) [T]
*Giant> leftshiftBy it (t 1)
W T [W T [V T []]]
*Giant> n it
1024
```

4.4 Implementing Haskell's Number and Order-related Type Classes

We can also make types `B` and `T` instances of Haskell's predefined number and order-related type clases like `Ord` and `Num`, etc., to enable the usual arithmetic operation and comparison syntax. The code for instance `B` follows:

```
instance Ord B where
  compare = cmp
instance Num B where
  (+) = add
  (-) = sub
  (*) = mul
  fromInteger = b
  abs = id
  signum B = B
  signum _ = O B
instance Integral B where
  quot = divide
  div = divide
  rem = remainder
  mod = remainder
  quotRem = div_and_rem
  divMod = div_and_rem
  toInteger = n
instance Real B where
  toRational = toRational . n
instance Enum B where
  fromEnum = fromEnum . n
  toEnum = b . f where
```

```
    f:: Int→Integer
    f = toEnum
  succ = s
  pred = s'
```

Note that as Haskell does not have a built-in natural number class we defined `abs` and `signum` to identity `id` and constant value corresponding to 1 `O B`.

The code for instance `T` is similar:

```
instance Ord T where
  compare = cmp
instance Num T where
  (+) = add
  (-) = sub
  (*) = mul
  fromInteger = t
  abs = id
  signum T = T
  signum _ = V T []
instance Integral T where
  quot = divide
  div = divide
  rem = remainder
  mod = remainder
  quotRem = div_and_rem
  divMod = div_and_rem
  toInteger = n
instance Real T where
  toRational = toRational . n
instance Enum T where
  fromEnum = fromEnum . n
  toEnum = t . f where
    f:: Int→Integer
    f = toEnum
  succ = s
  pred = s'
```

The following example illustrates their use:

```
*Giant> t 3
V (V T []) []
*Giant> t 4
W T [T]
*Giant> (V (V T []) []) + (W T [T])
V (W T []) []
*Giant> n it
7
```

5 Efficient representation of some important number-theoretical entities

Let's first observe that Fermat, Mersenne and perfect numbers have all compact expressions with our tree representation of type `T`.

```
fermat n = s (exp2 (exp2 n))

mersenne p = s' (exp2 p)

perfect p = s (V q [q]) where q = s' (s' p)
```

And one can also observe that this contrasts with both the `Integer` representation and the bijective base-2 numbers `B`:

```
*Giant> mersenne (b 127)
O (O (O (O (O (O (O (O (O (O (O (O (O (O
  ... a few lines of Os and Is
  ))))))))))))))))))))))))))))))))))))))))
  *Giant> mersenne (n 127)
170141183460469231731687303715884105727
*Giant> mersenne (t 127)
V (W (V T [T]) []) []
```

The largest known prime number, found by the GIMP distributed computing project [6] in January 2013 is the 48-th Mersenne prime $= 2^{57885161} - 1$. Its predecessor is $2^{57885161} - 1$. They are defined in Haskell as follows:

```
-- its exponent
prime48 = 57885161 :: Integer
prime45 = 43112609 :: Integer

-- the actual Mersenne primes
mersenne48 = s' (exp2 (t p)) where
  p = prime48::Integer

mersenne45 = s' (exp2 (t p)) where
  p = prime45::Integer
```

While they have bitsizes of 57885161 and 43112609, their compressed tree representation using our type `T` is rather small:

```
*Giant> mersenne48
V (W T [V T [],T,T,V (V T []) [],W T [T],T,T,V T [],V T [],W T [],T,T]) []
*Giant> mersenne45
V (W T [V (V T []) [],T,T,T,W T [],V T [],T,W T [],W T [],T,V T [],T,T]) []
```

One the other hand, displaying them with a decimal or binary representation would take millions of digits.

And by folding replicated subtrees to obtain an equivalent DAG representation, one can save even more memory. Figure 1 shows this representation, involving only 6 nodes.

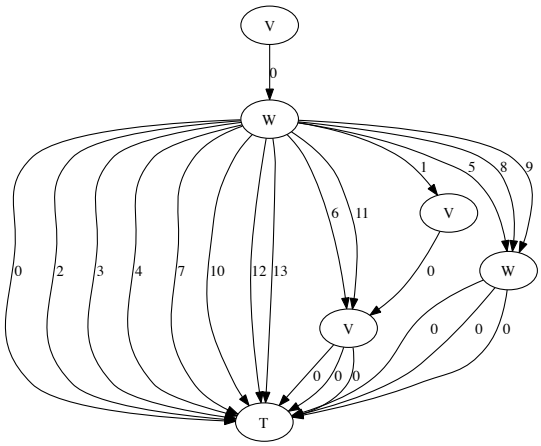

Fig. 1. Largest known prime number until 2013: the 45-th Mersenne prime, represented as a DAG

The equivalent DAG representation of the largest known prime number discovered in January 2013, shown in Figure 2 has only 7 nodes.

It is interesting to note that similar compact representations can also be derived for perfect numbers. For instance, the largest known perfect number, derived from the largest known Mersenne prime as $2^{57885161-1}(2^{57885161}-1)$, is:

```
perfect48 = perfect (t prime48)
perfect45 = perfect (t prime45)
```

Fig. 3 shows the DAG representation of `perfect45` involving only 7 nodes.

Fig. 4 shows the DAG representation of the largest known perfect number, derived from Mersenne number 48, involving only 8 nodes.

Similarly, the largest Fermat number that has been factored so far, F11=$2^{2^{11}}+1$ is compactly represented as

```
*Giant> fermat (t 11)
V T [T,V T [W T [V T []]]]
```

By contrast, its (bijective base-2) binary representation consists of 2048 digits.

Some other very large primes that are not Mersenne numbers also have compact representations.

The generalized Fermat prime $27653 * 2^{9167433} + 1$, (currently the 15-the largest prime number) computed as a V,W tree is:

```
genFermatPrime = s (leftshiftBy n k) where
  n = t (9167433::Integer)
  k = t (27653::Integer)
```

```
*Giant> genFermatPrime
V T [T,W (W T []) [W T [],T,V T [],T,W T [],W T [T],T,T,W T []],
  T,T,T,W (V T []) [],V T [],T,T]
```

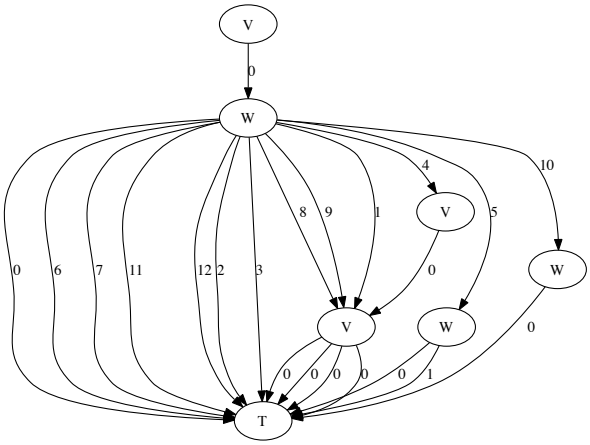

Fig. 2. Largest known prime number discovered in January 2013: the 48-th Mersenne prime, represented as a DAG

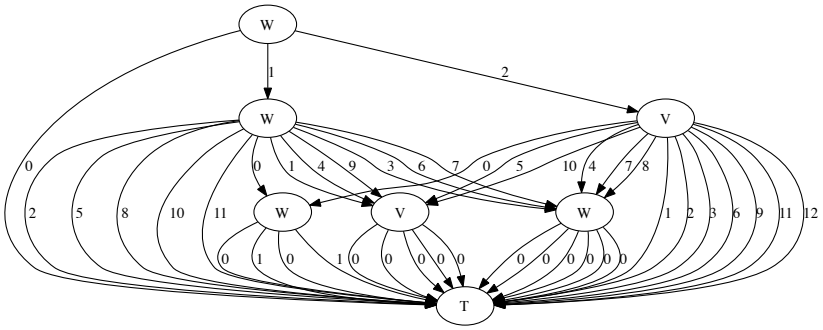

Fig. 3. Largest known perfect number before 2013

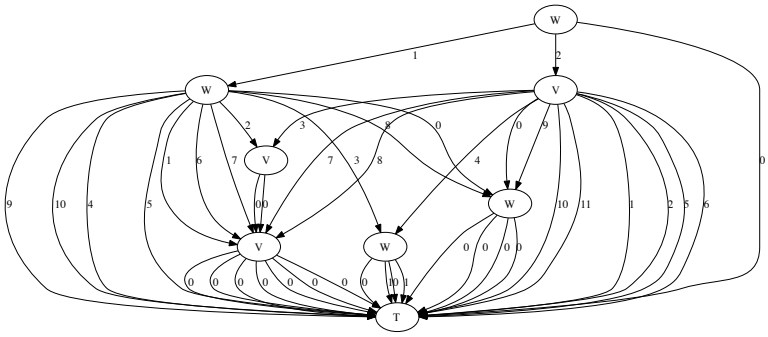

Fig. 4. Largest known perfect number in January 2013

Figure 5 shows the DAG representation of this generalized Fermat prime.

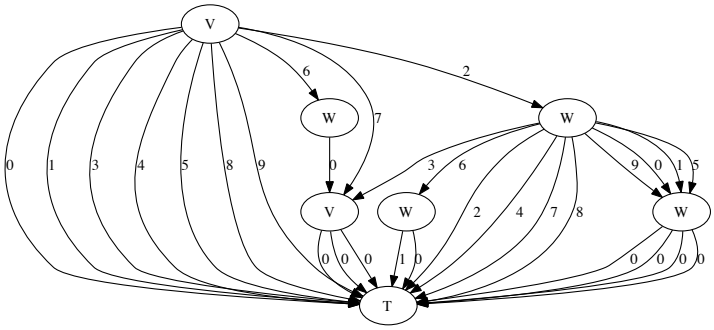

Fig. 5. Generalized Fermat prime

The largest known Cullen prime $6679881 * 2^{6679881} + 1$ computed as V,W-tree is:

```
cullenPrime = s (leftshiftBy n n) where n = t (6679881::Integer)
```

```
*Giant> cullenPrime
V T [T,W (W T []) [W T [],T,T,T,T,V T [],T,
  V (V T []) [],T,T,V T [],T],T,V T [],T,V T [],T,T,T,T,
  V T [],T,V (V T []) [],T,T,V T [],T]
```

Figure 6 shows the DAG representation of this Cullen prime.

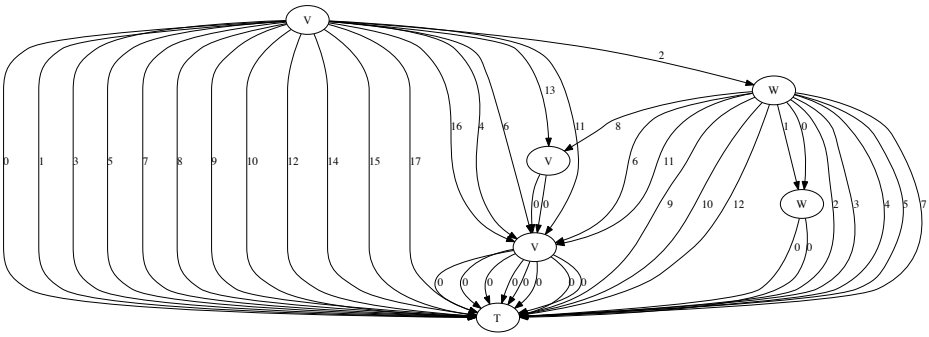

Fig. 6. largest known Cullen prime

The largest known Woodall prime $3752948 * 2^{3752948} - 1$ computed as a V,W tree is:

```
woodallPrime = s' (leftshiftBy n n) where n = t (3752948::Integer)
```

```
*Giant> woodallPrime
V (V T [V T [],T,V T [T],V (V T []) [],T,T,T,V T [],V T []])
  [T,T,V T [T],V (V T []) [],T,T,T,V T [],V T []]
```

Figure 7 shows the DAG representation of this Woodall prime.

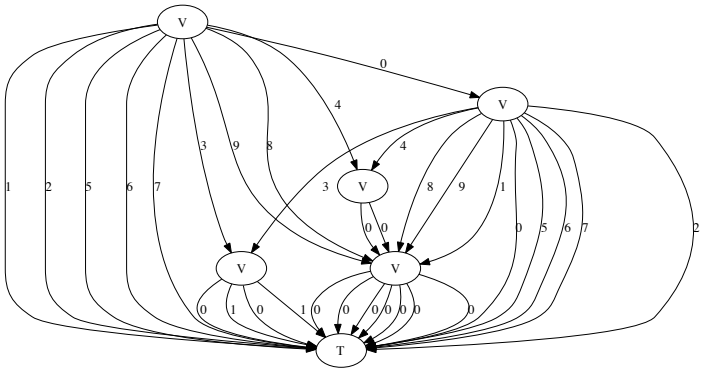

Fig. 7. Largest known Woodall prime prime

The largest known Proth prime $19249 * 2^{13018586} + 1$ computed as a V,W tree is:

```
prothPrime = s (leftshiftBy n k) where
  n = t (13018586::Integer)
  k = t (19249::Integer)
```

```
*Giant> prothPrime
V T [T,V (W T []) [V T [],T,W T [],T,T,V T [],T,T,T,T,V T [],
     W T [],T],T,W T [],V T [],V T [],V T [],T,T,V T []]
```

Figure 8 shows the DAG representation of this Proth prime, the largest non-Mersenne prime known by March 2013.

The largest known Sophie Germaine prime $18543637900515 * 2^{666667} - 1$ computed as a V,W tree is:

```
sophieGermainePrime = s' (leftshiftBy n k) where
  n = t (666667::Integer)
  k = t (18543637900515::Integer)
```

```
*Giant> sophieGermainePrime
V (W (V T []) [T,T,T,T,V (V T []) [],V T [],T,T,W T [],T,T])
  [V T [],W T [],W T [],V T [],V T [],T,T,V T [],V T [],V T [],
  V (V T []) [],T,V T [],V (V T []) [],V T [],T,W T [],T,V T [],
  V (V T []) []]
```

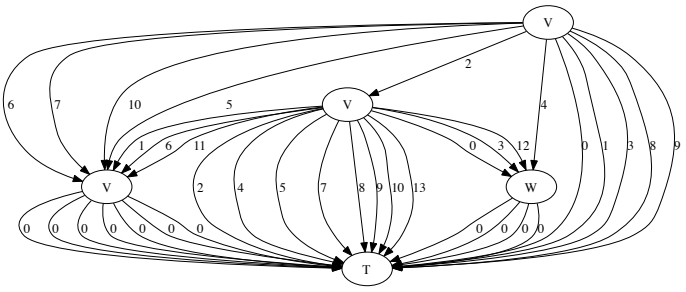

Fig. 8. Largest known Proth prime

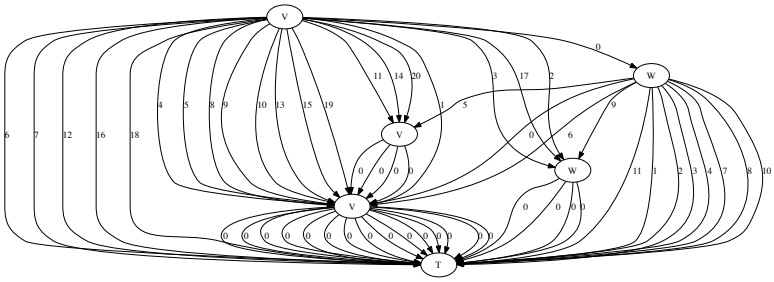

Fig. 9. Largest known sophie Germaine prime

Figure 9 shows the DAG representation of this prime.

The largest known twin primes $3756801695685 * 2^{666669} \pm 1$ computed as a pair of V,W-trees are:

```
twinPrimes = (s' m,s m) where
  n = t (666669::Integer)
  k = t (3756801695685::Integer)
  m = leftshiftBy n k
```

```
*Giant> fst twinPrimes
V (W T [T,V T [],T,T,V (V T []) [],V T [],T,T,W T [],T,T])
   [T,T,T,W T [],W (V T []) [],V T [],T,V T [],T,T,T,T,V T [],T,T,
   V T [],V T [],T,T,T,T,T,T,T,V T [],T,T]
*Giant> snd twinPrimes
V T [T,W (V T []) [T,T,T,T,V (V T []) [],V T [],T,T,W T [],T,T],T,T,T,
  W T [],W (V T []) [],V T [],T,V T [],T,T,T,T,V T [],T,T,V T [],
  V T [],T,T,T,T,T,T,T,V T [],T,T]
```

Figures 10 and 11 show the DAG representation of these twin primes.

One can appreciate the succinctness of our representations, given that all these numbers are have hundreds of thousands or millions of decimal digits.

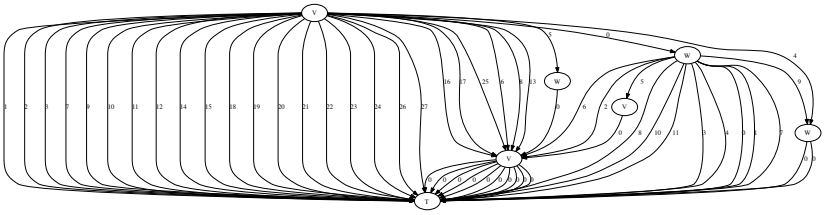

Fig. 10. Largest known twin prime 1

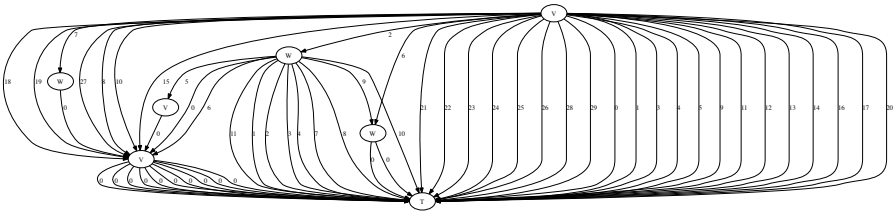

Fig. 11. Largest known twin prime 2

6 Representing sparse sets, multisets and lists

We will now describe 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 `T`, in contrast with bitstring or conventional `Integer`-based representations.

6.1 Bijections between collections and natural numbers

The type class `Collections` will convert between natural numbers and lists, by using the bijection $f(x, y) = 2^x(2y + 1)$, implemented by the function `c` and its first and second projections `c'` and `c''`, inverting it.

```
class HeredN n ⇒ Collections n where
  c :: n→n→n
  c',c'' :: n→n

  c x y = mul (exp2 x) (o y)

  c' x | not (e_ x) = if o_ x then e else s (c'  (hf x))
  c'' x | not (e_ x) = if o_ x then o' x else c'' (hf x)
```

The bijection between natural numbers and lists of natural numbers, `to_list` and its inverse `from_list` apply repeatedly `c` and respectively `c'` and `c''`.

```
  to_list :: n→[n]
  to_list x | e_ x = []
```

```
to_list x = (c' x) : (to_list (c'' x))

from_list:: [n]→n
from_list [] = e
from_list (x:xs) = c x (from_list xs)
```

6.2 Bijections between sequences, sets and multisets

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

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

list2mset ns = tail (scanl add e ns)
mset2list ms =  zipWith sub ms (e: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, to_set :: n→[n]
from_mset, from_set :: [n]→n

to_mset = list2mset . to_list
from_mset = from_list . mset2list

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

6.3 Instantiating the collection transformers

We will add the usual instances to the type class Collections. A simple number-theoretic observation connecting 2^n and n applications of the constructor `i`, implemented by the function `otimes`, allows a shortcut that speeds up the bijection from lists to natural numbers, by overriding the functions `c, c', c''` in instance `T`.

```
instance Collections B
instance Collections Integer
instance Collections T where
  c = cons where
    cons n y = s (otimes n (s' (o y)))

  c' = hd where
    hd z | o_ z = T
    hd z = s x where
      V x _ = s' z
```

```
  c'' = tl where
    tl z | o_ z = o' z
    tl z = f xs where
      V _ xs = s' z

      f [] = T
      f (y:ys) = s (i' (W y ys))
```

As the following example shows, trees of type `T` offer a significantly more compact representation of sparse sets.

```
*Giant> from_set (map t [1,100,123,234])
W (V T []) [V T [T,W T [],T],T,V T [V T [],T],T,V T [W T [],T,T]]
*Giant> from_set (map n [1,100,123,234])
27606985387162255149739023449108112443629669818608757680508075841159170
*Giant> from_set (map b [1,100,123,234])
I (I (O (O (O (O (O (O (O (O (O (O (O (O (O (O (O (O ...
... a few lines ...
..)))))))))))
```

Note that a similar compression occurs for sets of natural numbers with only a few elements missing, as they have the same representation size with type `T` as the `dual` of their sparse counterpart.

```
Giant> from_set ([1,3,5]++[6..220])
336999333339382997433337688587745383420464305281757156013795128113O
*Giant> t it
W (V T []) [T,T,T,W (W T []) [T,T,T,T]]
*Giant> dual it
V (V T []) [T,T,T,W (W T []) [T,T,T,T]]
*Giant> to_set it
[T,V T [],W T [T],W T [T,W T [],T,T]]
*Giant> map n it
[0,1,4,220]
```

6.4 An application: bitwise operations via ordered sets

As an application, we can define bitwise operations on our natural numbers by borrowing the corresponding ordered set operations, provided in Haskell by the package `Data.List.Ordered`.

First we define the type class `BitwiseOperations` and the higher order function `l_op` transporting a binary operation from ordered sets to natural numbers.

```
class Collections n ⇒ BitwiseOperations n where
  l_op :: ([n]→[n]→[n])→n→n→n
  l_op op x y = from_set (op (to_set x) (to_set y))
```

Next we define the bitwise `and`, `or` and `xor` operations:

```
  l_and,l_or,l_xor,l_dif:: n→n→n
  l_and = l_op (isectBy cmp)
```

```
l_or = l_op (unionBy cmp)
l_xor = l_op (xunionBy cmp)
l_dif = l_op (minusBy cmp)
```

More complex operations like the ternary `if-the-else` can be defined as a combination of binary operations:

```
l_ite :: n→n→n→n
l_ite x y z = from_set (ite (to_set x) (to_set y) (to_set z)) where
  ite d a b =  xunionBy cmp e b where
    c = xunionBy cmp a b
    e = isectBy cmp c d
```

Finally, bitwise negation (requiring additional parameter `l` defining the bitlength of the operand) can be defined using set complement with respect to $2^l - 1$, corresponding to the set of all elements up to `l`.

```
l_not :: Integer→n→n
l_not l x |xl≤l = from_set (minusBy cmp ms xs) where
  xs = to_set x
  xl = genericLength xs
  ms = genericTake l (allFrom e)
```

6.5 Alternative bitwise operations, using a 3-valued logic

An interesting question arises at this point: is it possible to use our generic bijective base-2 representation as the basis of a bitvector logic, without first transforming to ordered sets? 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" and absence of a corresponding value, when one sequence of applications is shorter than the other, will be interpreted as *undefined*. Note that this happens in a stronger sense than in Kleene's logic: conjunction of a value with *undefined* would be interpreted as *undefined*. Is it 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 (with `U` denoting *undefined*).

```
data Val = U|O|I

negation U = U
negation O = I
negation I = O

conjunction O O = O
conjunction O I = O
conjunction I O = O
conjunction I I = I
conjunction O U = O
conjunction U O = O
```

```
conjunction I U = I
conjunction U I = I
conjunction U U = U
```

Bitwise negation is implemented as follows:

```
neg :: n→n
neg x | e_ x = e
neg x | o_ x = i (neg (o' x))
neg x | i_ x = o (neg (i' x))
```

Note that the *undefined* 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, undefined corresponds to one or the other of the sequences ending with `e_`.

```
conj :: n→n→n
conj x _ | e_ x = e
conj _ y | e_ y = 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))
```

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

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

Bitwise <= and equality are implemented also like in classical logic:

```
leq :: n→n→n
leq x y = neg (conj x (neg y))

eq :: n→n→n
eq x y = conj (leq x y) (leq y x)
```

After adding the usual instances, one can try out this operation as follows:

```
instance BitwiseOperations B
instance BitwiseOperations Integer
instance BitwiseOperations T

*Giant> neg T
T
*Giant> conj (I (I (O B))) (O (I (I B)))
O (I (O B))
*Giant> disj (I (I (O B))) (O (I (I B)))
I (I (I B))
*Giant> conj 9 12
7
*Giant> leq (I (I (O B))) (O (I (I B)))
O (I (I B))
```

We will next generalize the iso-functor mechanism implemented generically by `l_op` that transports operations back an forth between data types, as a special data type `Iso`, consisting of two bijections inverse to each other.

7 Isomorphisms between data types, generically

Along the lines of [7] we can define isomorphisms 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 as well as "lending" operations from one to another is provided by the combinators `as`, `land1` and `land2`:

```
as that this x = to that (from this x)

lend1 op1 (Iso f f') x = f' (op1 (f x))
lend2 op2 (Iso f f') x y = f' (op2 (f x) (f y))
```

Assuming that the Haskell option `NoMonomorphismRestriction` is set on, we can now define generically "virtual types" centered around the type class `N`:

```
nat = Iso id id
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 various instances of type class `N` and their corresponding list, multiset and set types, as follows:

```
*Giant> as set nat 1234
[1,4,6,7,10]
*Giant> map t it
[V T [],W T [T],W (V T []) [],V (W T []) [],W (V T []) [T]]
*Giant> as nat set it
W (V T []) [V T [],T,T,V T [],V T []]
*Giant> n it
1234
*Giant> lend1 s set [0,2,3]
[1,2,3]
*Giant> lend2 add set [0,2,3] [4,5]
[0,2,3,4,5]
```

8 Complexity reduction in other computations

A number of other, somewhat more common computations also benefit from our data representations. The type class `SpecialComputations` groups them together and provides their bitstring inspired generic implementations.

8.1 Computing `bitsize` and `dual`, generically

The function `dual` flips `o` and `i` operations for a natural number seen as written in bijective base 2.

```
class Collections n ⇒ SpecialComputations n where
  dual :: n→n
  dual x | e_ x = e
  dual x | o_ x = i (dual (o' x))
  dual x | i_ x = o (dual (i' x))
```

The function `bitsize` computes the number of applications of the `o` and `i` operations:

```
  bitsize :: n→n
  bitsize x | e_ x = e
  bitsize x | o_ x = s (bitsize (o' x))
  bitsize x | i_ x = s (bitsize (i' x))
```

The function `repsize` computes the representation size, which defaults to the bitsize in bijective base 2:

```
  -- representation size - defaults to bitsize
  repsize :: n→n
  repsize = bitsize
```

8.2 Encoding lists with `cons` and `decons`

The functions `decons` and `cons` provide bijections between $\mathbb{N}-\{0\}$ and $\mathbb{N}\times\mathbb{N}$ and can be used as an alternative mechanism for building bijections between lists, multisets and sets of natural numbers and natural numbers. They are based on separating `o` and `i` applications that build up a natural number represented in bijective base 2.

```
  decons ::n→(n,n)
  cons :: (n,n)→n

  decons z | o_ z = (x,y) where
    x0 = s' (ocount z)
    y = otrim z
    x = if e_ y then (s'.o) x0 else x0
  decons z | i_ z = (x,y) where
    x0 = s' (icount z)
    y = itrim z
    x = if e_ y then (s'.i) x0 else x0

  cons (x,y) | e_ x && e_ y = s e
  cons (x,y) | o_ x && e_ y = itimes (s (i' (s x))) e
  cons (x,y) | i_ x && e_ y = otimes (s (o' (s x))) e
  cons (x,y) | o_ y = itimes (s x) y
  cons (x,y) | i_ y = otimes (s x) y
```

Note that implementing `decons` and `cons` requires counting the number of applications of `o` and `i` provided by `ocount` and `icount`, as well trimming the applications of `o` and `i`, performed by `otrim` and `itrim` defined in type class `HeredN`.

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

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

from_list' :: [n]→n
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', to_set' :: n→[n]
from_mset', from_set' :: [n]→n

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

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

8.3 Complexity reductions on hereditarily binary numbers

One can observe the significant reduction of asymptotic complexity with respect to the default operations provided by the type class `SpecialComputations` when overriding with `tbitsize` and `tdual` in instance `T`.

```
instance SpecialComputations Integer
instance SpecialComputations B
instance SpecialComputations T where
  bitsize = tbitsize where
    tbitsize T = T
    tbitsize (V x xs) = s (foldr add1 x xs)
    tbitsize (W x xs) = s (foldr add1 x xs)

    add1 x y = s (add x y)

  dual = tdual where
    tdual T = T
    tdual (V x xs) = W x xs
    tdual (W x xs) = V x xs

  repsize = tsize where
    tsize T = T
```

```
   tsize (V x xs) = s (foldr add T (map tsize (x:xs)))
   tsize (W x xs) = s (foldr add T (map tsize (x:xs)))
```

The replacement with special purpose code for the `cons / decons` functions follows from unfolding of the definitions of tt otimes, itiems, ocount, icount, otrim and `itrim` functions, specialized to data type `T`:

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

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

One can also see that their complexity is now proportional to `s` and `s'` given that the `V` and `W` operations perform in constant time the work of `otimes` and `itimes`. The following example illustrates their work:

```
*Giant> map to_list' [0..20]
[[],[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],[1,2],[1,0,0],[0,0,0,0]]
*Giant> map from_list' it
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

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

9 A performance comparison

Our performance measurements (run on a Mac Air with 8GB of memory and an Intel i7 processor) serve two objectives:

1. to show that, on the average, our tree based representations perform on a blend of arithmetic computations within a small constant factor compared with conventional bitstring-based computations
2. to show that on interesting special computations they outperform the conventional ones due to the much lower asymptotic complexity of such operations on data type `T`.

Objective *1* is served by the Ackerman function that exercises the successor and predecessor functions quite heavily, the prime generation and the Lucas-Lehmer primality test for Mersenne numbers that exercise a blend of arithmetic operations.

Objective *2* is served by the other benchmarks that take advantage of the overriding by instance `T` of operations like `exp2` and `bitsize`, as well as the compressed representation of large numbers like the 45-th Mersenne prime and

Benchmark	Integer	binary type `B`	tree type `T`
Ackermann 3 7	9418	7392	12313
exp2 (exp2 14)	23	315	0
sparse set encoding	7560	2979	45
bitsize on Mersenne 48	?	?	0
bitsize on Perfect 48	?	?	2
generating primes	2722	2567	3591
Mersenne prime tests	6925	6431	15037

Fig. 12. Time (in ms.) on a few benchmarks

perfect numbers. In some cases the conventional representations are unable to run these benchmarks within existing computer memory and CPU-power limitations (marked with `?` in the comparison table of Fig. 12). In other cases, like in the sparse set encoding benchmark, data type `T` performs significantly faster than binary representations.

Together they indicate that our tree-based representations are likely to be competitive with existing bitstring-based packages on typical computations and significantly outperform them on some number-theoretically interesting computations. While the code of the benchmarks is omitted due to space constraints, it is part of the companion Haskell file at `http://logic.cse.unt.edu/tarau/Research/2013/giant.hs`.

10 Related work

We will briefly describe here some related work that has inspired and facilitated this line of research and will help to put our past contributions and planned developments in context.

Several notations for very large numbers have been invented in the past. Examples include Knuth's *arrow-up* notation [8] covering operations like the *tetration* (a notation for towers of exponents). In contrast to our tree-based natural numbers, such notations are not closed under addition and multiplication, and consequently they cannot be used as a replacement for ordinary binary or decimal numbers.

The first instance of a hereditary number system, at our best knowledge, occurs in the proof of Goodstein's theorem [5], where replacement of finite numbers on a tree's branches by the ordinal ω allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates.

Numeration systems on regular languages have been studied recently, e.g. in [9] and specific instances of them are also known as bijective base-k numbers [2]. Arithmetic packages similar to our bijective base-2 view of arithmetic operations are part of libraries of proof assistants like Coq [10] and the corresponding regular

languages have been used as a basis of decidable arithmetic systems like `(W)S1S` [11] and `(W)S2S` [12].

Arithmetic computations based on recursive data types like the free magma of binary trees (isomorphic to the context-free language of balanced parentheses) are described in [13] and [14], where they are seen as Gödel's `System T` types, as well as combinator application trees. In [1] a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite functions.

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [15]. Their approach is similar as far as a symbolic representation is used. The key difference with our work is that our operations work on tree structures, and as such, they are not based on previously known algorithms.

Arithmetic computations with types expressed as `C++` templates are described in [16] and in online articles by Oleg Kiselyov using Haskell's type inference mechanism. However, the algorithm described there is basically the same as [15], focusing on Peano and binary arithmetics.

Efficient representation of sparse sets are usually based on a dedicated data structure [17] and they cannot be at the same time used for arithmetic computations as it is the case with our tree-based encoding.

In [18] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. However likewise [13] and [1], and by contrast to those proposed in this paper, they do not compress dense sets or numbers.

Ranking functions (bijections between data types and natural numbers) can be traced back to Gödel numberings [19] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [20, 21].

As a fresh look at the topic, we mention recent work in the context of functional programming on connecting heterogeneous data types through bijective mappings and natural number encodings [22–24].

11 Conclusion and future work

We have seen that the average performance of arithmetic computations with trees of type `T` is comparable, up to small constant factors, to computations performed with the binary data type `B` and it outperforms them by an arbitrarily large margin on the interesting special cases favoring the tree representations.

Still, does that mean that such binary trees can be used as a basis for a practical arbitrary integers package?

Native arbitrary length integer libraries like GMP or BigInteger take advantage of fast arithmetic on 64 bit words.

To match their performance, we plan to switch between bitstring representations for numbers fitting in a machine word and a a tree representation for numbers not fitting in a machine word.

We have shown that some interesting number-theoretical entities like Mersenne, Fermat and perfect numbers have significantly more compact representations with our tree-based numbers. One may observe their common feature: they are all represented in terms of exponents of 2, successor/predecessor and specialized multiplication operations.

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

The methodology to be used relies on two key components, that have been proven successful so far, in discovering succinct representations for Mersenne, Fermat and perfect numbers, as well as low complexity algorithms for operations like `bitsize` and `exp2`:

- partial evaluation of functional programs with respect to known data types and operations on them, as well as the use of other program transformations
- salient number-theoretical observations, provable by induction, that relate operations on our tree data types to known identities and number-theoretical algorithms

Acknowledgement

This research has been supported by NSF research grant 1018172.

References

1. Tarau, P.: Declarative modeling of finite mathematics. In: PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, New York, NY, USA, ACM (2010) 131–142
2. Wikipedia: Bijective numeration — wikipedia, the free encyclopedia (2012) [Online; accessed 2-June-2012].
3. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL. (1989) 60–76
4. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
5. Goodstein, R.: On the restricted ordinal theorem. Journal of Symbolic Logic (9) (1944) 33–41
6. Wikipedia: Great internet mersenne prime search — wikipedia, the free encyclopedia (2012) [Online; accessed 9-December-2012].
7. Tarau, P.: "Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. Complex Systems (18) (2010) 475–493
8. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
9. Rigo, M.: Numeration systems on a regular language: arithmetic operations, recognizability and formal power series. Theoretical Computer Science **269**(12) (2001) 469 – 498

10. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.
11. Büchi, J.R.: On a decision method in restricted second order arithmetic. International Congress on Logic, Method, and Philosophy of Science **141** (1962) 1–12
12. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society **141** (1969) 1–35
13. Tarau, P., Haraburda, D.: On Computing with Types. In: Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track, Riva del Garda (Trento), Italy (March 2012) 1889–1896
14. Tarau, P.: Declarative Specification of Tree-based Symbolic Arithmetic Computations. In Russo, C., Zhou, N.F., eds.: Proceedings of PADL'2012, Practical Aspects of Declarative Languages, Philadelphia, PA, Springer, LNCS7149 (January 2012) 273–289
15. Kiselyov, O., Byrd, W.E., Friedman, D.P., Shan, C.c.: Pure, declarative, and constructive arithmetic relations (declarative pearl). In: FLOPS. (2008) 64–80
16. Kiselyov, O.: Type arithmetics: Computation based on the theory of types. CoRR **cs.CL/0104010** (2001)
17. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems **2** (1993) 59–69
18. 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
19. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
20. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Berlin Heidelberg, Springer (2003) 572–581
21. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
22. Vytiniotis, D., Kennedy, A.: Functional Pearl: Every Bit Counts. ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming (September 2010) ACM Press.
23. Kobayashi, N., Matsuda, K., Shinohara, A.: Functional Programs as Compressed Data. ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (January 2012) ACM Press.
24. 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

Appendix

This appendix contains some additional code, used for testing and benchmarking our functions, grouped in the type class `Benchmarks`. First we define a prime number generator working with all our instances.

```
class SpecialComputations n ⇒ Benchmarks n where
  primes :: [n]
```

```
primes = s (s e) : filter is_prime (odds_from (s (s (s e)))) where
  odds_from x = x : odds_from (s (s x))

  is_prime p = [p]==to_primes p

  to_primes n = to_factors n p ps where
     (p:ps) = primes

  to_factors n p ps | cmp (mul p p) n == GT = [n]
  to_factors n p ps | e_ r =  p : to_factors q p ps where
     (q,r) = div_and_rem n p
  to_factors n p (hd:tl) = to_factors n hd tl
```

Next we define the Lucas-Lehmer fast primality test for Mersenne numbers:

```
lucas_lehmer :: n → Bool
lucas_lehmer p = e_ y where
  p_2 = s' (s' p)
  four = i (o e)
  m  = exp2 p
  m' = s' m
  y = f p_2 four

  f k n | e_ k = n
  f k n = r where
    x = f (s' k) n
    y = s' (s' (mul x x))
    --r = remainder y m'
    r = fastmod y m

  -- fast computation of k mod 2^p-1
  fastmod k m | k == s' m = e
  fastmod k m | LT == cmp k m = k
  fastmod k m = fastmod (add q r) m where
     (q,r) = div_and_rem k m

-- exponents leading to Mersenne primes
mersenne_prime_exps :: [n]
mersenne_prime_exps = filter lucas_lehmer primes

-- actual Mersenne primes
mersenne_primes :: [n]
mersenne_primes = map f mersenne_prime_exps where
  f p = s' (exp2 p)
```

The Ackerman function is a good benchmark for successor and predecessor operations:

```
ack :: n→n→n
ack x n | e_ x = s n
ack m1 x | e_ x = ack (s' m1) (s e)
```

```
  ack m1 n1 = ack (s' m1) (ack m1 (s' n1))
```

Next we define a variant of the 3x+1 problem / Collatz conjecture / Syracuse function (see `http://en.wikipedia.org/wiki/Collatz_conjecture`) that, somewhat surprisingly, can be expressed as a mix of arithmetic operations and reflected list operations, to test the relative performance of some of our instances. It is easy to show that the Collatz conjecture is true iff the function `nsyr`, implementing the n-th iterate of the *Syracuse function*, always terminates:

```
  syracuse  :: n→n
  -- n→c'' (3n+2)
  syracuse n = c'' (add n (i n))

  nsyr :: n→[n]
  nsyr n | e_ n = [e]
  nsyr n = n : nsyr (syracuse n)
```

Finally we close our type class with the usual instance declarations:

```
instance Benchmarks Integer
instance Benchmarks B
instance Benchmarks T
```

The following example illustrates the first 8 sequences of the Syracuse function:

```
*Giant> map nsyr [0..7]
[[0],[1,2,0],[2,0],[3,5,8,6,2,0],[4,3,5,8,6,2,0],
     [5,8,6,2,0],[6,2,0],[7,11,17,26,2,0]]
```

Our generic benchmark function measures the CPU time for running a no argument toplevel function `f` received as a parameter.

```
benchmark mes f = do
  x←getCPUTime
  print f
  y←getCPUTime
  let time=(y-x) `div` 1000000000
  return (mes++" :time="++(show time))
```

The following benchmarks provide the code used in the section 9.

```
bm1t = benchmark "ack 3 7 on t" (ack (t (toInteger 3)) (t (toInteger 7)))
bm1b = benchmark "ack 3 7 on b" (ack (b (toInteger 3)) (b (toInteger 7)))
bm1n = benchmark "ack 3 7 on n" (ack (n (toInteger 3)) (n (toInteger 7)))

bm2t = benchmark "exp2 t" (exp2 (exp2 (t (toInteger 14))))
bm2b = benchmark "exp2 b" (exp2 (exp2 (b (toInteger 14))))
bm2n = benchmark "exp2 n" (exp2 (exp2 (n (toInteger 14))))

bm3 tvar = benchmark "sparse_set on a type" (n (bitsize (from_set ps)))
  where ps = map tvar [101,2002..100000]
```

```
bm4t =benchmark "bitsize of Mersenne 45"  (n (bitsize mersenne45))
bm5t = benchmark "bitsize of Perfect 45" (n (bitsize perfect45))

bm6t = benchmark "large leftshiftBy" (leftshiftBy n n) where
  n = t prime45

bm3' tvar m = benchmark "to/from list on a type"
              (n (bitsize (from_list (to_list (from_list ps)))))
  where ps = map tvar [101,2002..3000+m]

bm3'' tvar m = benchmark "to/from list on a type"
              (n (bitsize (from_list (to_list (from_list ps)))))
  where ps = map (dual.tvar) [101,2002..3000+m]

bm7t = benchmark "primes on t"
  (last (take 100 ps)) where ps = primes :: [T]
bm7b = benchmark "primes on b"
  (last (take 100 ps)) where ps = primes :: [B]
bm7n = benchmark "primes on n"
  (last (take 100 ps)) where ps = primes :: [Integer]

bm8t = benchmark "mersenne on t"
  (last (take 7 ps)) where ps = mersenne_primes :: [T]
bm8b = benchmark "mersenne on b"
  (last (take 7 ps)) where ps = mersenne_primes :: [B]
bm8n = benchmark "mersenne on n"
  (last (take 7 ps)) where ps = mersenne_primes :: [Integer]
```

The following tests the syracuse / Collatz conjecture up to `m`

```
test_syr tvar m = maximum (map length (map (nsyr . tvar) [0..m]))

compress_syr tvar m = r where
  nss = map (nsyr . tvar) [0..m]
  r = maximum (map (n.bitsize) (map from_list nss))

-- overflows for m>2 except for tvar=t
compress_syr_twice tvar m = r  where
  nss = map (nsyr . tvar) [0..m]
  r = (n.bitsize) (from_list (map from_list nss))

bm9 tvar = benchmark "test syracuse" (test_syr tvar 2000)

bm10 tvar = benchmark "compress syracuse" (compress_syr tvar 100)

bm11 tvar = benchmark "compress syracuse_twice" r where
            r = compress_syr_twice tvar 20
```

The following function computes the size of a tree-represented natural number:

```
tsize T = 1
```

```
tsize (V x xs) = 1+ sum (map tsize (x:xs))
tsize (W x xs) = 1+ sum (map tsize (x:xs))
```

The function `kth` computes the k-th iteration of a function application.

```
kth _ k x | e_ k  = x
kth f k x  = f (kth f (s' k) x)
```

The following assertions are used for testing some of our operations:

```
-- relation between iterations of o,i and power of 2
a1 k = pow (i e) k == s (kth o k e)
a2 k = pow (i e) k == s (s (kth i (s'  k) e))

-- relations between power operations and multiplication
a3 n b = (u==v,u,v) where
 m = pow (i e) n
 u = kth o n b
 v = s' (mul m (s b))

a4 x y = (a==b,a,b) where
  a = mul (pow (i e) x) y
  b = s (kth o x (s' y))
```