Computing with Hereditarily Binary Natural Numbers

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
e-mail: tarau@cse.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?
Our *Hereditarily binary numbers* introduced in this paper provide compact tree-representations of the largest known prime number and its related perfect number. The same also applies to Fermat numbers and interesting complexity reductions happen to important computations like bitsize, while exponentiation of two becomes a constant time operation.
Keywords: *hereditary numbering systems, compressed number representations, arithmetic computations with giant numbers, tree-based numbering systems, compact representation of large Mersenne primes, perfect numbers and Fermat numbers*

1 Introduction

A natural question that may come to one's mind is if it possible to do arithmetic with a radically different underlying data structure, while maintaining the same fundamentals – addition, multiplication, exponentiation, primality, Peano's axioms behaving in the same way?

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. 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 interesting complexity reductions in other computations and section 7 compares the performance of our tree-representation with conventional ones. Section 8 discusses related work. Section 9 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/calc.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 $f : s \times t \rightarrow u$ (modulo Haskell's "currying" operation, given the isomorphism between the function spaces $s \times t \rightarrow u$ and $s \rightarrow t \rightarrow u$). 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.

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 $a_0, a_1, \ldots, a_k$. However, this encoding is 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 type classes

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 the bijective base-2 representation 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.

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 and power operations

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
```

Power is defined as follows:

```
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 = s (fPow o x e)

leftshiftBy :: n→n→n
leftshiftBy _ k |e_ k = e
leftshiftBy n k = s (fPow o n (s' k))

fPow :: (n→n)→n→n→n
fPow _ n x | e_ n = x
fPow f n x  = fPow f (s' n) (f x)
```

Note that such overridings take advantage of the specific encoding, as a result of simple number theoretic observations. Both `exp2` and the more general `leftshiftBy` operation uses the fact that the repeated application of the `o` operation (`fPow o`) provides an efficient implementation of multiplication with an exponent of 2. The correctness of of these operations is a consequence of the following proposition.

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

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

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:

```
*Calc> mul 10 5
50
*Calc> exp2 5
32
*Calc> add (O B) (I (O B))
O (I B)
```

4 Computing with a compressed tree representation

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 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
```

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

```
  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`:

```
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)
```

Note that these overridings take advantage of the efficient encoding of the application of function `o` n times for data type `T`.

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:

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

While the correctness of operations like `exp2` and `leftshiftBy` follows immediately from Prop. 2, we can test their correct behavior by converting their results to ordinary numbers:

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

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`:

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

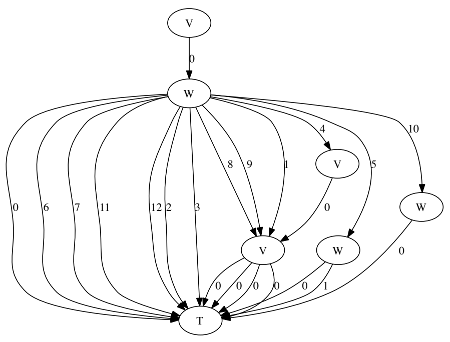

Fig. 1. Largest known prime number discovered in January 2013: the 48-th Mersenne prime, represented as a DAG

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

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

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

While it has a bitsize of 57885161, we have observed that its compressed tree representation using our type `T` is rather small:

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

One the other hand, displaying it 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 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)
```

The DAG representation of the largest known perfect number, derived from Mersenne number 48, involves only 8 nodes.

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

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

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

The largest known Proth prime $19249 * 2^{13018586} + 1$ is computed using the efficient `leftshiftBy` operation on trees of type `T`.

```
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 []]
```

The DAG representation of this Proth prime, the largest non-Mersenne prime known by March 2013, only involves 5 nodes.

The largest known twin primes $3756801695685 * 2^{666669} \pm 1$ computed as a pair of trees of type `T` are:

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

The DAG representation of these twin primes involves only 7 nodes each.

6 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.

6.1 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 N 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:

```
  repsize :: n→n
  repsize = bitsize
```

6.2 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 `bitsize` and `dual` in instance `T`.

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

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

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

7 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`.

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

Fig. 2. Time (in ms.) on a few benchmarks

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 48-th Mersenne prime and 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. 2). 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`.

8 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 [7] 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 [8] 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 [9] and the corresponding regular language has been used as a basis of decidable arithmetic systems like `(W)S2S` [10].

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 [11], 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 [12]. 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.

In [13] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types. However likewise [11] and [1], and by contrast to those proposed in this paper, they do not compress dense sets or numbers.

9 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 more 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?*

To answer some of this questions, future work is planned along the following lines:

– partial evaluation of functional programs with respect to the tree type `T`
– salient number-theoretical observations that relate operations on our tree data type to other identities and number-theoretical algorithms, similar to the results of Proposition 2.

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. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
8. Rigo, M.: Numeration systems on a regular language: arithmetic operations, recognizability and formal power series. Theoretical Computer Science **269**(12) (2001) 469 – 498
9. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.
10. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society **141** (1969) 1–35
11. 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
12. Kiselyov, O., Byrd, W.E., Friedman, D.P., Shan, C.c.: Pure, declarative, and constructive arithmetic relations (declarative pearl). In: FLOPS. (2008) 64–80
13. 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