Computing with Free Algebras and Generalized Constructors

Paul Tarau

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

Abstract

We describe arithmetic computations in terms of operations on some well known free algebras (S1S, S2S and ordered rooted binary trees) while emphasizing the common structure present in all them when seen as isomorphic with the set of natural numbers.

Constructors and deconstructors seen through an initial algebra semantics are generalized to recursively defined functions obeying similar laws. Implementations using Scala's `apply` and `unapply` and GHC's `view` construct are discussed together with an application to a realistic arbitrary size arithmetic package written in Scala, based on the free algebra of rooted ordered binary trees.

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

General Terms Algorithms, Languages, Theory

Keywords arithmetic computations with free algebras, generalized constructors, declarative modeling of computational phenomena, bijective Gödel numberings and algebraic datatypes

1. Introduction

The presocratic philosopher Anaximandros believed that *"everything is generated from apeiron and then it is destroyed there according to necessity"*. While "everything" meant back then mostly concrete observables like water, air, fire and earth, the ghost of a universal generator for the concepts we care about at a given time and in a given cultural context, has stayed with us ever since.

For instance, one can view the intrinsic information content [16] of objects represented in a computer's memory as a generic entity, that, by analogy with the presocratic *apeiron*, can take the *shape* of various data types. Under this assumption, if we want to be able to shift views while keeping the informational content invariant, the *morphisms* providing the transformations become *isomorphisms*. The natural framework to organize such "shapeshifting isomorphisms" is a *groupoid* i.e. a category [9] where every morphism is an isomorphism, with objects provided by the data types and morphisms provided by their bijective transformations [18].

A standard measure of this invariant information content is provided by the bitsize of its natural number representation.

When choosing the set of natural numbers $\mathbb{N}$ as the *hub* through which various data types are isomorphically connected, one rediscovers that these are just Gödel numberings - with the important addition that they are *bijective*, rather than injective only, as originally introduced in the proof of the incompleteness theorems [5].

This opens the door for doing with arbitrary data types everything one can do with natural numbers, and see their intrinsic information content as an invariant, always preserved by bijective transformations.

It is interesting to note, that a similar emphasis is apparent in recent work on mutual interpretability of Peano Arithmetic and finite ZF Set Theory's axiom systems [7, 12] and on connecting heterogeneous data types through bijective mappings [8, 17, 22].

Classical mathematics frequently uses functions defined on equivalence classes (e.g. modular arithmetic, factor objects in algebraic structures) provided that it can prove that the choice of a representative in the class is irrelevant.

On the other hand, when working with proof assistants based on type theory and its computationally refined extensions like the Calculus of Construction [3] (e.g. Coq [10]), or in type inference-enabled functional programming languages like Haskell, one cannot avoid noticing the prevalence of free objects on top of which everything else is built in the form of canonical representations.

Category-theory based descriptions of Peano arithmetic fit naturally in the general view that data types are *initial algebras* - in this case the initial algebra generated by the successor function, as a provider of the canonical representation of natural numbers. Of course, a critical element in choosing such free algebras is computational efficiency of the operations one wants to perform on them, in terms of low time and space complexity. For instance Coq formalizations of natural numbers typically use binary representations while keeping the Peano arithmetic view when more convenient in proofs [10]. Note also that free algebras corresponding to one and two successor arithmetic (S1S and S2S) have been used as a basis for decidable weak arithmetic systems like [2] and [13]. It has been shown recently in [20, 21] that the initial algebra of ordered rooted binary trees corresponding to the language of Gödel's System **T** types [6] can be used as a the language of arithmetic representations, with hyper-exponential gains when handling numbers built from "towers of exponents" like $2^{2^{\cdots^2}}$. Independently, this view is confirmed by the suggestion to use λ-terms as a form of universal data compression tool [8] and by deriving bijective encodings of data types using a game-based mechanism [22].

These results suggest a *free algebra based reconstruction of fundamental data types* that are relevant as building blocks for finite mathematics and computer science. We will sketch in this paper an (elementary, not involving category theory) foundation for arithmetic computations with free algebras, in which construction of sets, sequences, graphs, etc. can be further carried out along the lines of [18, 19, 21].

We define in section 2 isomorphisms between the free algebras of signatures consisting of one constant and respectively, of one

PPDP'12, Leuven, Belgium

successor (`S`), two successors (`O` and `I`) and a free magma constructor (`C`).

Such isomorphisms provide efficient and/or simple algorithms enabling a full range of arithmetic computations with any of our free algebras (sections 3, 4 and 5).

As the one successor free algebra models Peano natural numbers, these isomorphisms can be used to derive Gödel numberings of fundamental data types and programming language constructs, including sets, multisets, sequences as shown in section 6.

As an application to computations with the objects of the free algebras, we discuss the use of generalized constructors / destructors derived from these free algebras using the *apply / unapply* and *view* constructs available in Scala and Haskell (section 7).

Sections 8 and 9 discuss related work and our conclusions.

2. Free Algebras and Data Types

DEFINITION 1. *Let σ be a signature consisting of an alphabet of constants (called generators) and an alphabet of function symbols (called* constructors*) with various arities. We define the free algebra A_σ of signature σ inductively as the smallest set such that:*

1. *if c is a constant of σ then $c \in A_\sigma$*
2. *if f is an n-argument function symbol of σ, then $\forall i, 0 \leq i < n, t_i \in A_\sigma \Rightarrow f(t_0, \ldots, t_i, \ldots, t_{n-1}) \in A_\sigma$.*

We will write `c/0` for constants and `f/n` for function symbols with `n` arguments belonging to a given a signature.

More general definitions, e.g. as *initial objects* in the category of algebraic structures, are also used in the literature and a close relation exists with term algebras distinguishing between function constructors (generating the *Herbrand Universe*) and predicate constructors (generating the *Herbrand Base*).

Recursive data types in programming languages like Haskell, ML, Scala can be seen as a notation for free algebras. We refer to [24] for a clear and convincing description of this connection.

For instance, the Haskell declarations

```
data AlgU = U | S AlgU deriving (Eq,Read,Show)
data AlgB = B | O AlgB | I AlgB deriving (Eq,Read,Show)
data AlgT = T | C AlgT AlgT deriving (Eq,Read,Show)
```

correspond, respectively to

- the free algebra `AlgU` with a single generator `U` and unary constructor S (that can be seen as part of the language of Peano or Robinson arithmetic, or the decidable (W)S1S system, [2])
- the free algebra `AlgB` with single generator `B` and two unary constructors `O` and `I` (corresponding to the language of the decidable system (W)S2S [13]), as well as "bijective base-2" number notation [25]
- the free algebra `AlgT` with single generator `T` and one binary constructor `C` (essentially the same thing as the *free magma* generated by T).

The set-theoretical construction corresponding to the "`|`" operation is *disjoint union* and the data types correspond to infinite sets generated by applying the respective constructors repeatedly. The set-theoretical interpretation of "self-reference" in such data type definitions can be seen as fixpoint operation on sets of natural numbers as shown in the $P\omega$ construction used by Dana Scott in defining the denotational semantics for various λ-calculus constructs [15].

We will next "instantiate" some general results to make the underlying mathematics as *elementary* and self-contained as possible. While category theory is frequently used as the mathematical backing for data-types, we will provide here a simple set theory-based formalism, along the lines of [1].

We will start with the elementary mathematics behind the `AlgT` data type and follow with an outline for a similar treatment of `AlgU` and `AlgB`.

2.1 The free magma of ordered rooted binary trees with empty leaves

DEFINITION 2. *A set M with a (total) binary operation $*$ is called a* magma*.*

DEFINITION 3. *A morphism between two magmas M and M' is a function $f : M \to M'$ such that $f(x * y) = f(x) * f(y)$.*

Let X be a set. We define the sets $M_n(X)$ inductively as follows: $M_1(X) = X$ and for $n > 1$, $M_n(X)$ is the disjoint union of the sets $M_k(X) \times M_{n-k}(X)$ for $0 < k < n$. Let $M(X)$ be the disjoint union of the family of sets $M_n(X)$ for $n > 0$. We identify each set M_n with its canonical image in $M(X)$. Then for $w \in M_n(X)$, we call n the *length* of w and denote it $l(w)$. Let $w, w' \in M(X)$ and let $p = l(w)$ and $q = l(w')$. The image of $(w, w') \in M_p \times M_q$ under the canonical injection in $M(X)$ is called the composition of w and w' and is denoted $w * w'$.

When $X = \{T\}$ where T is interpreted as the "empty" leaf of ordered rooted binary trees, the elements of M_n can be seen as ordered rooted binary trees with n leaves while the composition operation "$*$" represents joining two trees at their roots to form a new tree.

DEFINITION 4. *The set $M(X)$ with the composition operation $(w, w') \to w * w'$ is called the* free magma *generated by X.*

PROPOSITION 1. *Let M be a magma. Then every mapping $u : X \to M$ can be extended in a unique way to a morphism of $M(X)$ into M.*

Proof We define inductively the mappings $f_n : M_m(X) \to M$ as follows: For $n = 1, f_1 = f$. For $n > 1, \forall p \in \{1, .., n-1\}, f_n(w * w') = f_p(w) * f_{n-p}(w')$. Let $g : M(X) \to M$ such that $\forall n > 0, \forall x \in M_n(X), g(x) = f_n(x)$. Then g is the unique morphism of $M(X)$ into M which extends f.

Note that this property corresponds of the *initial algebra* [24] view of the corresponding (ordered, rooted) *binary tree* data type.

DEFINITION 5. *If $u : X \to Y$, we denote $M(u) : M(X) \to M(Y)$ the unique morphism of magmas defined by the construction in* **Proposition 1**.

If $v : Y \to Z$ then the morphism $M(v) \circ M(u)$ extends $v \circ u : X \to Z$ and therefore $M(v) \circ M(u) = M(v \circ u)$.

PROPOSITION 2. *If $u : X \to Y$ is respectively injective, surjective, bijective then so is $M(u)$.*

It follows that

PROPOSITION 3. *If $X = \{x\}$ and $Y = \{y\}$ and $u : X \to Y$ is the bijection such that $f(x) = y$, then $M(u) : M(X) \to M(Y)$ is a bijective morphism (i.e. an* isomorphism*) of free magmas.*

Proof If X is empty so is $M(X)$, hence u is injective. If u is injective, then $\exists u' : Y \to X, u' \circ u = id_{M(X)}$ where $id_{M(X)}$ denotes the identity mapping of $M(X)$. Then $M(u') \circ M(u) = M(u' \circ u) = id_{M(X)}$ and hence $M(u)$ is injective. If u is surjective, then $\exists u' : Y \to X, u \circ u' = id_{M(Y)}$. Then $M(u) \circ M(u') = M(u \circ u') = id_{M(Y)}$ and hence $M(u)$ is surjective. If u is bijective, than it is injective and surjective and so is $M(u)$.

We will identify the data type `AlgT` with the free magma generated by the set $\{$`T`$\}$ and denote its binary operation $x * y$ as `C x y`.

It corresponds to the free algebra (that we will also denote `AlgT`) defined by the signature {`T/0, C/2`}.

We can now instantiate the results described by the previous propositions to `AlgT`:

PROPOSITION 4. *Let X be an algebra defined by a constant t and a binary operation c. Then there's a unique morphism f : `AlgT` $\rightarrow$ X that verifies*

$$f(T) = t \tag{1}$$

$$f(C(x, y)) = c(f(x), f(y)) \tag{2}$$

Moreover, if X is a free algebra then f is an isomorphism.

Proof It follows from Proposition 2 and equation $f(T) = t$, given that f is a bijection between the singleton sets {`T`} and $\{t\}$.

2.2 The One Successor and Two Successors Free Algebras

The *one successor* free algebra (also known as unary natural numbers or Peano algebra, as well as the language of the monoid $\{0\}^*$ and the decidable systems WS1S and S1S) is defined by the signature {`U/0, S/1`}, where `U` is a constant (seen as zero) and `S` is the unary successor function. We will denote `AlgU` this algebra and identify it with its corresponding Haskell data type.

We state an analogue of Proposition 4 for the free algebra `AlgU`.

PROPOSITION 5. *Let X be an algebra defined by a constant u and a unary operation s. Then there's a unique morphism f : `AlgU` $\rightarrow$ X that verifies*

$$f(U) = u \tag{3}$$

$$f(S(x)) = s(f(x)) \tag{4}$$

Moreover, if X is a free algebra then f is an isomorphism.

Note that following the usual identification of data types and initial algebras, `AlgU` corresponds to the initial algebra "**1 +** _" through the operation $g =$<`U,S`> seen as a bijection $g : 1 + \mathbb{N} \rightarrow \mathbb{N}$.

The *two successor* free algebra (also known as bijective base-2 natural numbers or Peano algebra, as well as the language of the monoid $\{0, 1\}^*$ and the decidable systems WS2S and S2S) is defined by the signature {`B/0, O/1, I/1`} where `B` is a constant (seen as zero) and `O, I` are two unary successor functions. We will denote `AlgB` this algebra and identify it with its corresponding Haskell data type.

We can state an analogue of Proposition 4 for the free algebra `AlgB`.

PROPOSITION 6. *Let X be an algebra defined by a constant b and a two unary operations o, i. Then there's a unique morphism f : `AlgB` $\rightarrow$ X that verifies*

$$f(B) = b \tag{5}$$

$$f(O(x)) = o(f(x)) \tag{6}$$

$$f(I(x)) = i(f(x)) \tag{7}$$

Moreover, if X is a free algebra then f is an isomorphism.

These observations suggest that for defining isomorphisms between `AlgU, AlgB` and `AlgT` that enable a complete set of equivalent arithmetic (and later set-theoretic) operations on each of them, we will need a mechanism to prove such equivalences. To this end, it will be enough to prove that such non-constructor operations also form free algebras of matching signatures.

We will call `terms` the elements of our initial algebras.

3. Successor and Predecessor in `AlgB`

We start with a straightforward definition of a successor/predecessor pair on `AlgB`. Note that this is essentially the language of the decidable two successor arithmetic systems `WS2S` and `S2S` which also resurfaces as *bijective base-2 arithmetic*. The intuition for designing these operations is their conventional arithmetic interpretation, as `O` for `B`, $\lambda x.2x + 1$ for `O` and $\lambda x.2x + 2$ for `I`.

```
sB B = O B            -- 1 --
sB (O x) = I x        -- 2 --
sB (I x) = O (sB x)   -- 3 --

sB' (O B) = B         -- 1' --
sB' (O x) = I (sB' x) -- 3' --
sB' (I x) = O x       -- 2' --
```

PROPOSITION 7. *Let $\mathbb{B}$ be the set of terms of the initial algebra* `AlgB` *and $\mathbb{B}^+ = \mathbb{B} - \{B\}$. Then* `sB`: $\mathbb{B} \rightarrow \mathbb{B}^+$ *is a bijection and* `sB'`: $\mathbb{B}^+ \rightarrow \mathbb{B}$ *is its inverse.*

Proof (Sketch). We proceed by structural induction. Clearly the proposition holds for the base case as `sB'(sB B) = sB' (O B) = B` and `sB (sB' (O B)) = sB B = O B`. The result follows from the inductive hypothesis by observing that exactly one rule matches each expression and an application of rule "`-- 2 --`" is undone by "`-- 2' --`" and an application of rule "`-- 3 --`" is undone by rule "`-- 3' --`" and viceversa.

The functor `u2b` defined as

```
u2b ::AlgU → AlgB
u2b U = B
u2b (S x) = sB (u2b x)
```

and its inverse

```
b2u ::AlgB → AlgU
b2u B = U
b2u x = S (b2u (sB' x))
```

define an isomorphism between the two algebras which allows us to see `AlgB` as a model for an axiomatization of arithmetic on $\mathbb{N}$.

We can thus generate the stream enumerating the terms of `algB` as follows:

```
binNats = iterate sB B


*FreeAlg> take 8 binNats
[B,O B,I B,O (O B),I (O B),O (I B),I (I B),O (O (O B))]
```

Other arithmetic operations, can be defined in terms of `sB`, `sB'` and structural recursion. For instance, the addition `addB` operation looks as follows:

```
addB B y = y
addB x B = x
addB(O x) (O y) = I (addB x y)
addB(O x) (I y) = O (sB (addB x y))
addB(I x) (O y) = O (sB (addB x y))
addB(I x) (I y) = I (sB (addB x y))
```

4. Successor and Predecessor in `AlgT`

We proceed with similar successor/predecessor definitions for the algebra `AlgT`. First, we state the induction principle for `AlgT`.

PROPOSITION 8. *Let $P(x)$ be a predicate about the terms of* `AlgT`. *If P holds for the generator $T \in$`AlgT` and from $P(x)$ and $P(y)$ one can conclude $P(C\ x\ y)$, then P holds for all terms of* `AlgT`.

This time, the definitions of successor `s` and predecessor `s'`, together with the helper functions `d` and `d'` are mutually recursive:

```
s T = C T T              -- 1  --
s (C T y) = d (s y)      -- 2  --
s z = C T (d' z)         -- 3  --

s' (C T T) = T           -- 1' --
s' (C T y) = d y         -- 3' --
s' z = C T (s' (d' z))   -- 2' --

d (C a b) = C (s a) b    -- 4  --
d' (C a b) = C (s' a) b  -- 4' --
```

The intuition for designing these operations is their conventional arithmetic interpretation, as `0` for `T`, $\lambda x.\lambda y.2^x(2y+1)$ for `C`, $\lambda x.2x$ for `d` (assuming $x > 0$) and $\lambda x.x/2$ (assuming x even and $x > 0$) for `d'`.

We will give a simple proof of a key property of the mutually recursive successor `s` and predecessor `s'` functions on the initial algebra of ordered rooted binary trees, the fact that they are *inverses*.

PROPOSITION 9. *Let $\mathbb{T}$ be the set of terms of the initial algebra* `AlgT` *and $\mathbb{T}^+ = \mathbb{T} - \{T\}$. Then* `s`: $\mathbb{T} \to \mathbb{T}^+$ *is a bijection and* `s'`: $\mathbb{T}^+ \to \mathbb{T}$ *is its inverse.*

Proof We will proceed by induction on the structure of the terms of `AlgT`. Observe that `f` is the inverse of `f'` if and only if $\forall u \in \mathbb{T},\ \forall v \in \mathbb{T}^+,\ f\ u = v \iff f'\ v = u$. We will show this for the base case and the inductive steps for both `s` and `s'` as well as `d` and `d'`.

Observe that if `s` and `s'` are inverses, then `d` and `d'` are also inverses. This reduces to showing that: $d\ y = z \iff d'\ z = y$, or equivalently, that $d\ (C\ a\ b) = C\ c\ d \iff d'\ (C\ c\ d) = C\ a\ b$, which further reduces to $C\ (s\ a)\ b = C\ c\ d \iff C\ (s'\ c)\ d = C\ a\ b$ and $s\ a = c \iff s'\ c = a$, which holds based on the inductive hypothesis for `s` and `s'`.

We can now start our main induction proof, by case analysis. Observe that rules `k` and `k'` are such that rule `-- k --` is the unique match for function f if and only if rule `-- k' --` is the unique match for function f'.

We will show that $s\ u = v \iff s'\ v = u$, assuming it holds inductively forall a, b such that $v = C\ a\ b$. Note that case `k` corresponds to the application of rules `-- k --` and `-- k' --` in the definitions of `s` and `s'`.

1. $s\ u = s\ T = C\ T\ T = v \iff s'\ v = s'\ (C\ T\ T) = T = u$
2. $s\ u = s\ (C\ T\ y) = d\ (s\ y) = v \iff s\ y = d'\ v$
   $s'\ v = C\ T\ y$ where $y = s'\ (d'\ v) \iff s\ y = d'\ v$, given that d and d' are inverses under the inductive hypothesis covering their calls to s and s'.
3. $v = s\ u \iff v = C\ T\ y$ where $y = d'\ u$
   $u = s'\ v \iff v = C\ T\ y$ where $u = d\ y$, which holds, given that d and d' are inverses under the inductive hypothesis covering their calls to s and s'.

The functor `u2b` defined as

```
u2t ::AlgU → AlgT
u2t U = T
u2t (S x) = s (u2t x)
```

and its inverse

```
t2u ::AlgT → AlgU
t2u T = U
t2u x = S (t2u (s' x))
```

define an isomorphism between the two algebras which allows us to see `AlgT` as a model for an axiomatization of arithmetic on $\mathbb{N}$, e.g. Peano arithmetic or Robinson's weaker axiom system `Q`.

This property of the function `s` ensures that the infinite stream `treeNats` of binary trees, corresponding to successive natural numbers can be defined simply as:

```
treeNats = iterate s T
```

The following example illustrates the first 5 such trees:

```
*FreeAlg> take 5 treeNats
[T, C T T, C (C T T) T, C T (C T T), C (C (C T T) T) T]
```

5. Arithmetic Computations in `AlgT`

It has been shown in [19] that inductive definitions of arithmetic operations can be transported through isomorphisms to equivalent recursive functions computing directly on hereditarily finite sets and sequences, as well as Gödel's System **T** types represented as ordered rooted binary trees [20].

To exhibit in a simple form the main intuition of this methodology we will sketch here only how defining a `AlgB` "view" over the free algebra `AlgT` enables *arithmetic computations with binary trees* with complexity bound comparable to those acting on conventional bitstring representations.

We start by defining projection functions (`c', c''`) and a recognizer of non-empty trees `c_` on data type `AlgT`:

```
c',c'' :: AlgT → AlgT

c' (C x _) = x
c'' (C _ y) = y

c_ :: AlgT → Bool
c_ (C _ _) = True
c_ T = False
```

Next we emulate projection functions (`o', i'`) and recognizers (`o_, i_`) on `AlgT` equivalent to constructors on data type `AlgB`:

```
o,o' :: AlgT → AlgT
o = C T
o' (C T y) = y

o_ :: AlgT → Bool
o_ (C T _) = True
o_ _ = False
```

```
i,i' :: AlgT → AlgT
i = s . o
i' = o' . s'

i_ :: AlgT → Bool
i_ (C (C _ _) _) = True
i_ _ = False
```

Note that these emulations allow defining an iso-functor between the two free algebras `AlgB` and `AlgT`:

```
b2t :: AlgB → AlgT
b2t B = T
b2t (O x) = o (b2t x)
b2t (I x) = i (b2t x)

t2b :: AlgT → AlgB
t2b T = B
t2b x | o_ x = O (t2b (o' x))
t2b x | i_ x = I (t2b (i' x))
```

We are now ready for the magic: an addition operation working directly on binary trees.

```
add T y  = y
add x T  = 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)))
```

To facilitate testing that such computations are indeed isomorphic with the usual arithmetic operations on $\mathbb{N}$ (seen as Haskell's arbitrary size Integer type), we can write down the explicit conversion functions between $\mathbb{N}$ and `AlgT` objects:

```
type N = Integer

n2t :: N → AlgT
n2t 0 = T
n2t x | x>0 = C (n2t (nC' x)) (n2t (nC'' x)) where
  nC' x|x>0 =
    if odd x then 0 else 1+(nC'  (x `div` 2))
  nC'' x|x>0 =
    if odd x then (x-1) `div` 2 else nC'' (x `div` 2)
```

```
t2n :: AlgT → N
t2n T = 0
t2n (C x y) = nC (t2n x) (t2n y) where
  nC x y = 2^x*(2*y+1)
```

The following examples illustrate that arithmetic operations on trees match the usual ones:

```
*FreeAlg> add (n2t 1000) (n2t 2000)
C (C T (C T T)) (C T (C T (C (C T T)
  (C T (C T (C (C T T) T))))))
*FreeAlg> t2n (add (n2t 1000) (n2t 2000))
3000
```

Fig. 1 shows the representation of the binary tree associated to 2012, as a result of applying the function `t2n`. The conventional arithmetic equivalents of the (recursively generated) subtrees are used as labels for the nodes and `0,1` labels on the edges indicate left/right branches of the binary trees. Note also that a DAG representation is used by collapsing identical nodes.

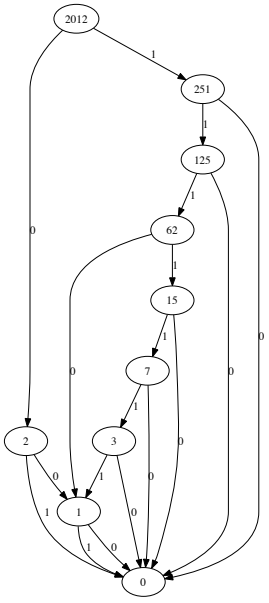

Figure 1. The DAG representation of the binary tree associated to 2012

Other arithmetic operations are defined quite naturally, e.g. subtraction (`sub`) and multiplication (`mul`).

```
sub x T = 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)))
```

While the algorithm for `sub` mimics how one would work on elements of `AlgB`, the algorithm for `mul` takes advantage of the fact that the left branches of the trees (corresponding to exponents of 2) require only an addition operation to compute the left branch of the result.

```
multiply T _ = T
multiply _ T = T
multiply x y = C (add (c' x) (c' y)) (add a m) where
  (x',y') = (c'' x,c'' y)
  a = add x' y'
  m = s' (o (multiply x' y'))
```

Note also that a `O(1)` complexity power of 2 operation `exp2` is simply

```
exp2 x = C x T
```

A remarkable property emerges: our tree representation supports operations with gigantic, tower of exponent numbers that would overflow even if each atoms in the known universe would be used as bit, with conventional bitstring representations. We illustrate this with the following example:

```
*FreeAlg> take 7 (iterate exp2 T)
[T,C T T,C (C T T) T,C (C (C T T) T) T,
 C (C (C (C T T) T) T) T,C (C (C (C (C T T) T) T) T) T,
 C (C (C (C (C (C T T) T) T) T) T) T]

*FreeAlg> map t2n it
[0,1,2,4,16,65536,20035299304068... -- 2-pages of digits --
  ...339445587895905719156736]
```

Any number after the last one (obtained by iterating `exp2` 7 times) would overflow any computer's memory with a conventional bitstring representation, while each step only adds a constant amount of memory to our tree representation. Note that "`it`" represents in Haskell the result of the previously evaluated interactive query.

6. Canonical Representations of Sets and Multisets

Along the lines of [19] one can also provide bijections between a model of $\mathbb{N}$ (in particular the terms of `AlgT`) and its corresponding sequences, multisets and sets, providing *canonical representations of each of these fundamental data types*. We will briefly overview how this works on the terms of `AlgT`.

The mapping between our binary trees and their corresponding lists is derived simply by using the projection functions `c'` and `c''` as head and tail operations.

```
to_list :: AlgT → [AlgT]
to_list T = []
to_list x = (c' x) : (to_list (c'' x))

from_list :: [AlgT] → AlgT
from_list [] = T
from_list (x:xs) = C x (from_list xs)
```

Multisets are represented through a simple addition/subtraction-based transformation:

```
list2mset,mset2list :: [AlgT] → [AlgT]

list2mset ns = tail (scanl add T ns)
mset2list ms =  zipWith sub ms (T:ms)

to_mset :: AlgT → [AlgT]
to_mset = list2mset . to_list

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

The transformation from lists to sets and back is similar:

```
list2set,set2list :: [AlgT] → [AlgT]

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

to_set :: AlgT → [AlgT]
to_set = list2set . to_list

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

We will illustrate them by first converting from usual numbers to trees. The following example shows that a canonical representation of subsets of $\mathbb{N}$ (ordered and with all elements distinct) is bijectively associated to each tree-represented natural number.

```
*FreeAlg> to_set (n2t 123)
[T,C T T,C T (C T T),C (C (C T T) T) T,
 C T (C (C T T) T),C (C T T) (C T T)]
*FreeAlg> map t2n it
[0,1,3,4,5,6]
*FreeAlg> from_set (map n2t it)
C T (C T (C (C T T) (C T (C T (C T T)))))
*FreeAlg> t2n it
123
```

Note that the bijections between sets, multisets and lists (sequences) of natural numbers make no assumptions about these objects being finite. Moreover, given Haskell's lazy evaluation, such computations can be used to interoperate between list representations of infinite streams consisting of any of these data types.

We refer to [19] and [21] for details on covering hereditarily finite sets and sequences, System **T** types, boolean logic, parenthesis languages as well as the **SKI** and Rosser's **X** combinator calculi in terms of bijective Gödel numberings that support computations independently of the concrete representation of the underlying type class acting as $\mathbb{N}$.

7. Generalized Constructors

The iso-functors supporting the equivalence between actual constructors and their recursively defined function counterparts suggest exploring programming language constructs that treat them in a similar way. For instance it makes sense to extend "constructor-only benefits" like pattern matching to their function counterparts.

Fortunately, constructors/deconstructors generalized to arbitrary functions are available in Scala through `apply/unapply` methods and in Haskell through a special notation implementing *views*, under the implicit assumption that they define inverse operations. One can immediately notice that our free algebras provide sufficient conditions under which this assumption is enforced. This suggests the possibility that such generalized constructor/deconstructor pairs could provide the combined benefits of pattern matching and data abstraction, with the implication that direct syntactic support for such constructs can bring significant expressiveness to functional programming languages.

7.1 Generalized Constructors with `apply/unapply` in Scala

Besides supporting `case classes` and `case objects` that are used (among other things) to implement pattern matching, Scala's `apply` and `unapply` methods [4, 11] allow definition of customized constructors and destructors (called *extractors* in Scala).

We will next describe how arithmetic operations with our `AlgT` terms, represented as ordered rooted binary trees, can benefit from the use such "generalized constructors".

Our `AlgT` free algebra will correspond in Scala to a `case object / case class` definition, combined with a mechanism to share actual code, encapsulated in the `AlgT` trait.

```
case object T extends AlgT

case class C(l: AlgT, r: AlgT) extends AlgT

trait AlgT {
  def s(z: AlgT): AlgT = z match {
    case T       ⇒ C(T, T)
    case C(T, y) ⇒ d(s(y))
    case z       ⇒ C(T, h(z))
  }

  def p(z: AlgT): AlgT = z match {
    case C(T, T) ⇒ T
    case C(T, y) ⇒ d(y)
    case z       ⇒ C(T, p(h(z)))
  }
```

Note the similarity with our Haskell code in section 4, except that the predecessor function is called `p` and our auxiliary functions are named `d` (which "doubles" its input, assumed different from `T`) and `h` (which "halves" its input, assumed "even" and different from `T`).

```
  def d(z: AlgT): AlgT = z match {
    case C(x, y) ⇒ C(s(x), y)
  }

  def h(z: AlgT): AlgT = z match {
    case C(x, y) ⇒ C(p(x), y)
  }
 }
```

We will define our *generalized constructor/destructor* `S` representing the successor function and predecessor function on rooted ordered binary trees of type `AlgT` by providing `apply` and `unapply` methods expressed in terms of our "real" constructors `T` and `C` and the actual algorithms defined in the (shared) trait `AlgT`.

```
object S extends AlgT {
  def apply(x: AlgT) = s(x)

  def unapply(x: AlgT) = x match {
    case C(_, _) ⇒ Some(p(x))
    case T       ⇒ None
  }
}
```

The definition of the generalized constructor/destructor `D` representing double / half is similar. Note the use of the method `d` defined in the trait `AlgT`.

```
object D extends AlgT {
  def apply(x: AlgT) = d(x)

  def unapply(x: AlgT) = x match {
    case C(C(_, _), _) ⇒ Some(h(x))
    case _             ⇒ None
  }
}
```

The definition of the generalized constructor/destructor `O` representing $\lambda x.2x+1$ and its inverse corresponds to the Haskell equivalent of `o`, `o'` and `o_` in section 5.

```
object O extends AlgT {
  def apply(x: AlgT) = C(T, x)

  def unapply(x: AlgT) = x match {
    case C(T, b) ⇒ Some(b)
    case _       ⇒ None
  }
}
```

The definition of the generalized constructor/destructor `O` representing $\lambda x.2x+2$ and its inverse corresponds to the Haskell equivalent of `i`, `i'` and `i_` in section 5. Note the use of the generalized constructors `S`, `D` and `O`, both on the left and right side of `match` statements as a proof of "scalability" of this mechanism.

```
object I extends AlgT {
  def apply(x: AlgT) = S(O(x))

  def unapply(x: AlgT) = x match {
    case D(a) ⇒ Some(p(a))
    case _    ⇒ None
  }
}
```

7.2 A Scala-based Arithmetic Package using `AlgT` Terms

We will now illustrate how the use of generalized constructors helps writing a fairly complete set of arithmetic algorithms on `AlgT`. For comparison purposes, the reader might want to look at the Haskell code in [21] where similar algorithms are expressed using a type class-based mechanism. However, while the use of type classes comes with the benefits of *data abstraction* it needs separate functions for constructing, deconstructing and recognizing terms to express the equivalent of the generalized constructors used here.

We start with a comparison function returning `LT, EQ, GT` supporting a `total order` relation on `AlgT`, isomorphic to the one on $\mathbb{N}$. Note here the use of the generalized constructors `O` and `I` providing a view of the terms of `AlgT` as terms of the free algebra `BinT`.

```
trait Tcompute extends AlgT {
  def cmp(u: AlgT, v: AlgT): Int = (u, v) match {
    case (T, T)       ⇒ EQ
    case (T, _)       ⇒ LT
    case (_, T)       ⇒ GT
    case (O(x), O(y)) ⇒ cmp(x, y)
    case (I(x), I(y)) ⇒ cmp(x, y)
    case (O(x), I(y)) ⇒ strengthen(cmp(x, y), LT)
    case (I(x), O(y)) ⇒ strengthen(cmp(x, y), GT)
  }

  val LT = -1
  val EQ = 0
  val GT = 1

  private def strengthen(rel: Int, from: Int) =
    rel match {
      case EQ ⇒ from
      case _  ⇒ rel
    }
```

Addition is expressed compactly in terms of generalized constructors `O`, `I` and `S`.

```
  def add(u: AlgT, v: AlgT): AlgT = (u, v) match {
    case (T, y)       ⇒ y
    case (x, T)       ⇒ x
    case (O(x), O(y)) ⇒ I(add(x, y))
    case (O(x), I(y)) ⇒ O(S(add(x, y)))
    case (I(x), O(y)) ⇒ O(S(add(x, y)))
    case (I(x), I(y)) ⇒ I(S(add(x, y)))
  }
```

The definition of subtraction is similar, except that the code of the predecessor function `p`, needed in this algorithm, is conveniently inherited directly from the trait `AlgT`, given that the trait `Tcompute` extends it.

```
  def sub(u: AlgT, v: AlgT): AlgT = (u, v) match {
    case (x, T)       ⇒ x
    case (O(x), O(y)) ⇒ p(O(sub(x, y)))
    case (O(x), I(y)) ⇒ p(p(O(sub(x, y))))
    case (I(x), O(y)) ⇒ O(sub(x, y))
    case (I(x), I(y)) ⇒ p(O(sub(x, y)))
  }
```

The multiplication operation is similar to the Haskell code in section 5, except for the use of the generalized constructor `O`.

```
  def multiply(u: AlgT, v: AlgT): AlgT = (u, v) match {
    case (T, _) ⇒ T
    case (_, T) ⇒ T
    case (C(hx, tx), C(hy, ty)) ⇒ {
      val v = add(tx, ty)
      val z = p(O(multiply(tx, ty)))
      C(add(hx, hy), add(v, z))
    }
  }
```

Similarly, a constant time complexity definition is given here for the exponent of 2 operation, by using the "real" constructor `C`.

```
  def exp2(x: AlgT) = C(x, T)
```

An efficient power operation takes advantage of the generalized constructors `O` and `I` on the left side of a `case` statement through the `AlgB` view of `AlgT`.

```
  def pow(u: AlgT, v: AlgT): AlgT = (u, v) match {
    case (_, T)    ⇒ C(T, T)
    case (x, O(y)) ⇒ multiply(x, pow(multiply(x, x), y))
    case (x, I(y)) ⇒ {
      val xx = multiply(x, x)
      multiply(xx, pow(xx, y))
    }
  }
```

Efficient division with remainder is a slightly more complex algorithm, where we take advantage of generalized constructors, direct inheritance from trait `AlgT` as well as number of previously defined functions:

```
  def div_and_rem(x: AlgT, y: AlgT): (AlgT, AlgT) =
    if (cmp(x, y) == LT) (T, x)
    else if (T == y) null // division by zero
    else {
      def try_to_double(x:AlgT, y:AlgT, k:AlgT): AlgT =
        if (cmp(x, y) == LT) p(k)
        else try_to_double(x, D(y), S(k))

      def divstep(n: AlgT, m: AlgT): (AlgT, AlgT) = {
        val q = try_to_double(n, m, T)
        val p = multiply(exp2(q), m)
        (q, sub(n, p))
      }
      val (qt, rm) = divstep(x, y)
      val (z, r) = div_and_rem(rm, y)
      val dv = add(exp2(qt), z)
      (dv, r)
    }
```

Division and reminder can be separated using Scala's projection functions:

```
  def divide(x: AlgT, y: AlgT) = div_and_rem(x, y)._1

  def reminder(x: AlgT, y: AlgT) = div_and_rem(x, y)._2
```

finally, the greatest common divisor `gcd` and the least common multiplier `lcm` are defined as follows:

```
  def gcd(x: AlgT, y: AlgT): AlgT =
    if (y == T) x else gcd(y, reminder(x, y))

  def lcm(x: AlgT, y: AlgT): AlgT =
    multiply(divide(x, gcd(x, y)), y)
}
```

The trait trait `Tconvert` implements efficiently conversion to/from Scala's `BigInt` arbitrary size integers using bit-level operations corresponding to power of 2 and recognition of odd and even natural numbers. The function `fromN` builds an `AlgT` tree representation equivalent to a `BigInt`.

```
trait Tconvert {
 def fromN(i: BigInt): AlgT = {
     def oddN(i: BigInt) =
       i.testBit(0)

     def evenN(i: BigInt) =
       i != BigInt(0) && !i.testBit(0)

     def hN(x: BigInt): BigInt =
       if (oddN(x))
         BigInt(0)
       else
         BigInt(1) + hN(x >> 1)

     def tN(x: BigInt): BigInt =
       if (oddN(x))
         (x - BigInt(1)) >> 1
       else
         tN(x >> 1)

   if (0 == i) T
   else C(fromN(hN(i)), fromN(tN(i)))
 }
```

The function `toN` converts an `AlgT` tree representation to a `BigInt`.

```
  def toN(z: AlgT): BigInt = z match {
    case T ⇒ 0
    case C(x, y) ⇒
      (BigInt(1) << toN(x).intValue()) *
        (BigInt(2) * toN(y) + 1)
  }
}
```

Note that for both these conversions we have used, for efficiency reasons, the "real constructors" `T` and `C`, although much simpler (and slower) converters can be built using either the `AlgB` or `AlgU` view of `AlgT` terms.

The use of Scala's generalized constructors inspired by our free algebra isomorphisms have shown the combined flexibility of inheritance as a mechanism for data abstraction and convenient pattern matching allowing the design of our algorithms in a functional style. The implicit use of `apply` and `unapply` methods in combination with our simple free algebra semantics has facilitated the safe use of fairly complex (mutually) recursive functions in the definition of the generalized constructors. The use of Scala's traits has facilitated flexible inheritance mechanisms supporting shared definitions without any additional syntactic clutter.

7.3 Expressing Generalized Constructors as Views in Haskell

An interesting question arises at this point: is it possible to express similar constructs combining the benefits of pattern matching and data abstraction in a more conventional functional language like Haskell?

Views have beed introduced to functional programming in [23] and have made it as an optional feature into GHC (enabled with the command line switch `-XViewPatterns`).

We will show that when combining *views* with a type class mechanism, one can implement (with a slight notational overhead) generalized constructors of comparable flexibility to our Scala ones.

We will describe them this time on the (simpler) `AlgB` terms, but the mechanism can be extended easily to computations in other free algebras.

First we define a data type `BWrapper` similar to `AlgB` parameterized by a type `n` to which it applies the constructors `O` and `I`.

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

The type class `Bins` specifies the operations that will support our generalized constructors for `AlgB`.

```
class (Read n,Show n,Eq n) ⇒ Bins n where
  b :: n

  b_,o_,i_ :: n→Bool
  b_ x = x==b

  o,i,o',i' :: n→n
```

It also defines, generically, the `apply` and `unapply` functions that act as converters between the actual constructors `B,O,I` and the corresponding generic function definitions.

```
  unapply :: n → BWrapper n
  unapply x | b_ x  = B
  unapply x | o_ x =  O (o' x)
  unapply x | i_ x = I (i' x)

  apply :: BWrapper n → n
  apply B = b
  apply (O x) = o x
  apply (I x) = i x
```

We will now show these constructs "in action" in the type class `Arith` implementing successor and predecessor functions `s` and `s'` with GHC's `view` construct (represented syntactically as `->`), that calls the function `unapply` in (possibly nested) patterns.

```
class (Bins n) ⇒  Arith n where
  s,s' :: n → n
  s (unapply→B) = apply (O (apply B))
  s (unapply→O x) = apply (I x)
  s (unapply→I x) = apply (O (s x))

  s' (unapply→O (unapply→B)) = apply B
  s' (unapply→O x) = apply (I (s' x))
  s' (unapply→I x) = apply (O x)
```

The same mechanism can be used for operations like addition `add`:

```
  add:: n→n→n
  add (unapply→B) y = y
  add x (unapply→B) = x
  add (unapply→O x) (unapply→O y) =
    apply (I (add x y))
  add (unapply→O x) (unapply→I y) =
    apply (O (s (add x y)))
  add (unapply→I x) (unapply→O y) =
    apply (O (s (add x y)))
  add (unapply→I x) (unapply→I y) =
    apply (I (s (add x y)))
```

Note that while benefitting from pattern matching our operations are placed in a type class and as such they are generic. We will first create an `instance` using the Haskell's arbitrary size `Integers`.

```
instance Bins Integer where
  b=0

  o x = 2*x+1
  i x = 2*x+2

  o_ x = x>0 && odd x
  i_ x = x>0 && even x

  o' x = (x-1) `div` 2
  i' x = (x-2) `div` 2
instance Arith Integer
```

Next, we create an instance using lists of booleans (this requires an extra GHC switch so to actually run this code snippet (assumed in file views.hs), one needs to type something like `ghci -XViewPatterns -XFlexibleInstances views`.

```
instance Bins [Bool] where
```

```
  b=[]

  o xs=  False:xs
  i xs = True:xs

  o_ (False:_) = True
  o_ _ = False

  i_ (True:_) = True
  i_ _ = False

  o' (False:xs) = xs
  i' (True:xs) = xs
instance Arith [Bool]
```

After defining the stream generator `allFrom` as

```
allFrom k = k : allFrom (s k)
```

one can observe the use of such generic operations on the two instances:

```
*Views> take 10 (allFrom 1)
[1,2,3,4,5,6,7,8,9,10]
*Views> take 5 (allFrom [False])
[[False],[True],[False,False],[True,False],[False,True]]
```

8. Related Work

Numeration systems on regular languages have been studied recently, e.g. [14] and specific instances of them are also known as bijective base-k numbers [25]. Arithmetic packages similar to `AlgU` and `AlgB` 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` [2] and `(W)S2S` [13].

Arithmetic computations based on the more complex recursive data types like the free magma of binary trees (essentially isomorphic to the context-free language of balanced parentheses) are described in [21] and [20], where they are seen as Gödel `System T` types and [19] where a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite functions. However, none of these papers provide proofs of the properties of the underlying free algebras or constructs similar to the generalized constructors described in this paper.

9. Conclusion

We have shown that free algebras corresponding to some basic data types in programming languages can be used for arithmetic computations isomorphic to the usual operations on $\mathbb{N}$.

Among the new contributions, we have worked-out details of proofs, based only on elementary mathematics, of essential properties of the mutually recursive successor and predecessor functions, on the free algebra of ordered rooted binary trees.

A concept of *generalized constructor*, for which we have found simple implementations in `Scala` and `Haskell`, has been introduced. By working in synergy with our free algebra isomorphisms we have described, using language constructs like Scala's `apply / unapply` and GHC's *views*, simple and safe means to combine data abstraction and pattern matching in modern-day functional and object oriented languages.

Future work is planned to investigate possible practical applications of our algorithms to symbolic and/or arbitrary length integer arithmetic packages and to parallel execution of arithmetic computations.

Note that the Haskell code given in this paper is self-contained as a "literate Haskell program" and it is also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2012/freealg.hs`. The code snippet using Haskell views is available as a separate file at `http://logic.cse.unt.edu/tarau/research/2012/views.hs`. The code snipped showing the use of Scala's `apply` and `unapply` methods is available as a separate file at `http://logic.cse.unt.edu/tarau/research/2012/AlgT.scala`.

References

[1] Nicolas Bourbaki. *Elements of Mathematics, Algebra I*. Springer, Berlin Heidelberg New York, 1989.

[2] J. R. Büchi. On a decision method in restricted second order arithmetic. *International Congress on Logic, Method, and Philosophy of Science*, 141:1–12, 1962.

[3] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[4] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 273–298. Springer Berlin / Heidelberg, 2007.

[5] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[6] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(280-287):12, 1958.

[7] Richard Kaye and Tin Lock Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.

[8] Naoki Kobayashi, Kazutaka Matsuda, and Ayumi Shinohara. Functional Programs as Compressed Data. *ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, January 2012. ACM Press.

[9] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, NY, USA, 1998.

[10] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008.

[12] Richard Pettigrew. On Interpretations of Bounded Arithmetic and Bounded Set Theory. Notre Dame J. Formal Logic Volume 50, Number 2 (2009), 141-151.

[13] Michael. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[14] Michel Rigo. Numeration systems on a regular language: arithmetic operations, recognizability and formal power series. *Theoretical Computer Science*, 269(12):469 – 498, 2001. ISSN 0304-3975.

[15] Dana Scott. Data Types as Lattices. *Siam J. Comput.*, 5(3):522–587, 1976.

[16] Claude E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993. ISBN 0-7803-0434-9.

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

[18] Paul Tarau. "Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. *Complex Systems*, (18):475–493, 2010.

[19] Paul Tarau. Declarative modeling of finite mathematics. In *PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0132-9. doi: http://doi.acm.org/10.1145/1836089.1836107.

[20] Paul Tarau. Declarative Specification of Tree-based Symbolic Arithmetic Computations. In *Proceedings of PADL'2012, Practical Aspects*

of Declarative Languages, pages 273–289, Philadelphia, PA, January 2012.

[21] Paul Tarau and David Haraburda. On Computing with Types. In *Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track*, Riva del Garda (Trento), Italy, March 2012.

[22] Dimitrios Vytiniotis and Andrew Kennedy. Functional Pearl: Every Bit Counts. *ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming*, September 2010. ACM Press.

[23] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: http://doi.acm.org/10.1145/41625.41653. URL `http://doi.acm.org/10.1145/41625.41653`.

[24] Philip Wadler. Recursive Types for Free! 1990. unpublished manuscript at `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.

[25] Wikipedia. Bijective numeration — wikipedia, the free encyclopedia, 2012. URL `http://en.wikipedia.org/w/index.php?title=Bijective_numeration&oldid=483462536`. [Online; accessed 2-June-2012].

Acknowledgement

This research has supported by NSF grant 1018172.