Shared Axiomatizations and Virtual Datatypes

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

Abstract

In the form of a literate Haskell program, we provide a "shared axiomatization" of Peano arithmetics, a bit-stack representation of bijective base-2 arithmetics, hereditarily finite sets (ZF-set theory with the negation of the axiom of infinity and ϵ-induction) and a few other equivalent constructs, that turn out to express basic programming language concepts ranging from lists, sets and multisets, to trees, graphs and hypergraphs.

The "axiomatization" is described as a progressive refinement of Haskell type classes with examples of instances converging to an efficient implementation in terms of arbitrary length integers and bit operations.

The resulting framework, extended with combinators providing isomorphisms between equivalent data representations, virtualizes data types as isomorphisms to a common representation supporting safe transfer of operations in the presence of polymorphic types.

The self-contained source code of the paper is available at *http://logic.cse.unt.edu/tarau/research/2009/sharedAxioms.hs* .

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

General Terms Algorithms, Languages, Theory

Keywords computational mathematics, data type transformations, hereditarily finite sets and functions, pairing functions, digraph, DAG and hypergraph encodings, Haskell type classes

1. Introduction

While axiomatizations of various formal systems are traditionally expressed in classic or intuitionistic predicate logic, equivalent formalisms, in particular the λ-calculus and the type theory used in modern functional languages like Haskell, can provide specifications in a sometime more readable, more concise, and more importantly, in a genuinely executable form. We will take the liberty in this paper to explore some interesting properties of finite arithmetics and finite set theory directly as Haskell code, while keeping in mind, and also assuming from the reader, some familiarity with the underlying predicate logic axiomatizations and their interdependencies, as described, for instance, in [Takahashi 1976, Kaye and Wong 2007, Abian and Lamacchia 1978, Kirby 2007, Cégielski and Richard 2001].

Natural numbers and finite sets have been used as sometimes competing foundations for mathematics, logic and consequently computer science. The de facto standard axiomatization for natural numbers is provided by second order Peano arithmetics. Finite set theory is axiomatized with the usual Zermelo-Fraenkel system (abbreviated ZF from now on) in which the Axiom of Infinity is replaced by its negation. When the axiom of ϵ-induction, (saying that if properties proven on elements also hold on sets containing them, then they hold for all finite sets) is added, the resulting finite set theory (abbreviated ZF^* from now on) is bi-interpretable with Peano arithmetics i.e. they emulate each other accurately through a bijective mapping that commutes with standard operations on the two sides ([Kaye and Wong 2007]).

This paper provides, in the form of a literate Haskell program, a "shared axiomatization" of Peano arithmetics, bit-stacks, hereditarily finite sets and a few other equivalent constructs to progressively build basic programming language concepts ranging from lists, sets and multisets, to trees, graphs and hypergraphs. As an interesting feature, successive refinements through a chain of type classes connected by inheritance is used. Instances are added progressively providing examples that illustrate various concepts.

While a number of novel algorithms (some fairly intricate like implementing arithmetic computations directly in terms of hereditarily finite sets and hereditarily finite functions in sections 4 and 5) are worth exploring in detail and analyzing in separate papers, we believe that the main contribution of this paper is the framework that unifies fundamental mathematical concepts in a genuinely constructive (i.e. directly executable) form, as well as the implicit software refinement methodology allowing the derivation of successive extensions as Haskell type classes enjoying the joint benefits of a higher order functional programming language and an object oriented coding style.

The following specific contributions might be also worth mentioning:

- a hierarchy of type classes describing common computational capabilities shared by bit-stacks, Peano natural numbers, hereditarily finite sets, hereditarily finite functions (sections 3-6)
- alternative finite set, function and list theories (sections 10-12) parameterized by distinct pairing functions (section 9)
- a new concept of virtual type, encapsulating concrete types as isomorphisms to a common representation (section 14)
- a uniform encoding of various graph types through set-encodings parameterized by pairing functions (section 15)

2. Choosing a starting point: BitStacks

Bitstrings provide a common and efficient computational representation for both sets and natural numbers. This recommends their operations as the right abstraction for deriving, in the form of a

Haskell type class, a "shared axiomatization" for Peano arithmetics and Finite Set Theory.

While the existence of such a common axiomatization can be seen as a consequence of the bi-interpretability described in [Kaye and Wong 2007], our distinct executable specification as a Haskell type class provides unique insights into the shared inductive constructions and ensures that computational complexity of operations is kept under control for a variety of instances, some with practical uses as highly parallel implementations of both theories.

We start by expressing bitstring operations as a Haskell data type, after defining our module and a few imports.

```
module SharedAxioms where
import Data.List
import Data.Bits
import CPUTime

data BitStack = Empty|Bit0 BitStack|Bit1 BitStack
  deriving (Eq, Show, Read)
```

on which we define the following operations

```
empty = Empty

push0 xs = Bit0 xs
push1 xs = Bit1 xs

pop (Bit0 xs)=xs
pop (Bit1 xs)=xs
```

and the predicates

```
empty_ x=Empty==x
bit0_ (Bit0 _)=True
bit0_ _ =False

bit1_ (Bit1 _)=True
bit1_ _=False
```

As a simple exercise in bijective base-2 arithmetics[1] one can now implement the successor function - and therefore provide a model of Peano's axioms

```
zero = empty
one = Bit0 empty

peanoSucc xs | empty_ xs = one
peanoSucc xs | bit0_ xs = push1 (pop xs)
peanoSucc xs | bit1_ xs = push0 (peanoSucc (pop xs))
```

working as follows:

```
*SharedAxioms> (peanoSucc . peanoSucc . peanoSucc) zero
Bit0 (Bit0 Empty)
```

One can verify by structural induction that Peano's axioms hold with this definition of the successor function. Using this representation, by contrast with successor based definitions, one can implement arithmetic operations like sum and product with low polynomial complexity in terms of the bitsize of their operands. We will defer defining these operations until the next section, where we will provide such implementations in a more general setting.

Note that as a mild lookahead step towards abstracting away operations on our bitstacks, we have replaced reference to data constructors by the corresponding predicates and functions.

We will spare the kind reader from a similar exercise showing basic set operations on our bitstacks seen as characteristic functions of sets, and just conclude this section by saying, that in a nutshell, our bitstacks promise to have the capabilities needed to emulate both Peano arithmetics and ZF-finite sets in a single framework.

[1] The best reference for this is the Wikipedia article. An important aspect of the representation is that *all* distinct strings in the regular language $\{0,1\}^*$ represent distinct numbers and 0 is represented as the empty string.

3. Sharing axiomatizations with Type Classes

Haskell's *type classes* [Jones et al. 1997] are a good approximation of axiom systems as they allow one to describe properties and operations generically i.e. in terms of their action on objects of a parametric type. Haskell's *instances* approximate *interpretations* [Kaye and Wong 2007] of such axiomatizations by providing implementations of primitive operations and by refining (and possibly overriding) derived operations with more efficient equivalents.

We will start by defining a type class that abstracts the operations on the `BitStack` datatype and provides an axiomatization of natural numbers first, and finite sets and a few other related datatypes later. In particular, we will cover theories of finite sets, multisets and lists as well as their hereditarily finite counterparts.

3.1 The 5 primitive operations

The class `Polymath` assumes only a theory of equality (as implemented by the class `Eq` in Haskell) and the `Read/Show` superclasses needed for input/output.

An instance of this class is required to implement the following 5 primitive operations:

```
class (Eq n,Read n,Show n)⇒Polymath n where
  e :: n
  o_ :: n→Bool
  o :: n→n
  i :: n→n
  r :: n→n
```

We have chosen single letter names `e,o_,o,i,r` for the abstract operations corresponding respectively to `empty, bit0_, push0, push1, pop` to help with a more algebraic view as some definitions will use fairly complex compositions of these operations. As a minimal definition, the class will also provide generic implementations of the following derived operations:

```
  e_ :: n→Bool
  e_ x = x==e

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

While not strictly needed at this point, it is convenient also to include in this class some additional derived operations, although as we will see, some instances will chose to override them later. We first define an object and a recognizer for `1`, the constant function `u` and the predicate `u_`.

```
  u :: n
  u = o e

  u_ :: n→Bool
  u_ x = o_ x && e_ (r x)
```

Next we implement the successor `s` and predecessor `p` functions:

```
  s :: n→n -- succ
  s x | e_ x = u
  s x | o_ x = i (r x)
  s x | i_ x = o (s (r x))

  p :: n→n -- pred
  p x | u_ x = e
  p x | o_ x = i (p (r x))
  p x | i_ x = o (r x)
```

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 `Polymath` as well as their associated lists.

```
view :: (Polymath a,Polymath b)⇒a→b
```

```
view x | e_ x = e
view x | o_ x = o (view (r x))
view x | i_ x = i (view (r x))

views :: (Polymath a,Polymath b)⇒[a]→[b]
views = map view
```

And for the reader curious by now about how this maps to arithmetics as usual, here is an instance built around the (arbitrary length) `Integer` type:

```
newtype A = A Integer deriving (Eq,Show,Read)

instance Polymath A where
  e = A 0
  o_ (A x) = odd x
  o (A x) = A (2*x+1)
  i (A x) = A (2*x+2)
  r (A x) | x/=0 = A ((x-1) `div` 2)
```

on which one can try out

```
*SharedAxioms> (o . i . o) (A 0)
A 9
*SharedAxioms> (r . r . r) (A 9)
A 0
```

It is important to observe at this point that Peano arithmetics is also an instance of the class `Polymath` i.e. that the class can be used to derive an "axiomatization" for Peano arithmetics through a straightforward mapping of Haskell's function definitions to axioms expressed in second order logic.

```
data Peano = Zero|Succ Peano deriving (Eq,Show,Read)

instance Polymath Peano where
  e = Zero

  o_ Zero = False
  o_ (Succ x) = not (o_ x)

  o x = Succ (o' x) where
    o' Zero = Zero
    o' (Succ x) = Succ (Succ (o' x))

  i x = Succ (o x)

  r (Succ Zero) = Zero
  r (Succ (Succ Zero)) = Zero
  r (Succ (Succ x)) = Succ (r x)
```

And one can now try out the polymorphic instance converter `view`:

```
*SharedAxioms> view (Succ (Succ Zero)) :: A
A 2
*SharedAxioms> view (A 2) :: Peano
Succ (Succ Zero)
```

Finally, we can add `BitStack` - which, after all, has inspired the operations of our type class, as an instance of `Polymath`:

```
instance Polymath BitStack where
  e=empty
  o=push0
  o_=bit0_
  i=push1
  r=pop
```

and observe that it behaves as expected

```
*SharedAxioms> view (A 42) :: BitStack
Bit1 (Bit1 (Bit0 (Bit1 (Bit0 Empty))))
```

So far we have seen that our instances implement syntactic variations of natural numbers equivalent to Peano's axioms. We will now provide an instance showing that our "axiomatization" covers the theory of hereditarily finite sets (assuming, of course, that extensionality, comprehension, regularity, ϵ-induction etc. are implicitely provided by type classes like `Eq` and implementation of recursion in the underlying programming language).

4. Computing with Hereditarily Finite Sets

Hereditarily finite sets are built inductively from the empty set (denoted `S []`) by adding finite unions of existing sets at each stage. We first define a tree datatype `S`:

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

To accurately represent sets, the type `S` would require a type system enforcing constraints on type parameters, saying that all elements covered by the definition are distinct and no repetitions occur in any list of type `[S]`. We will assume this and similar properties of our datatypes, when needed, from now on, and consider trees built with the constructor `S` as representing hereditarily finite sets.

We will now show that hereditarily finite sets can do "BitStack arithmetics" as instances of the class `Polymath` by implementing a successor (and predecessor) function. We start with the easier operations:

```
instance Polymath S where
  e = S []

  o_ (S (S []:_)) = True
  o_ _ = False

  o (S xs) = s (S (map s xs))

  i = s . o
```

Note that the `o` operation, that can be seen as pushing a `0` bit to a bitstack (or as a left shift on a bitstring) is implemented by applying `s` to each branch of the tree. We will now implement `r, s` and `p`.

```
  r (S xs) = S (map p (f ys)) where
    S ys = p (S xs)
    f (x:xs) | e_ x = xs
    f xs = xs

  s (S xs) = S (hLift (S []) xs) where
    hLift k [] = [k]
    hLift k (x:xs) | k==x = hLift (s x) xs
    hLift k xs = k:xs

  p (S xs) = S (hUnLift xs) where
    hUnLift ((S []):xs) = xs
    hUnLift (k:xs) = hUnLift (k':k':xs) where k'= p k
```

First note that successor and predecessor operations `s,p` are overridden and that the `r` operation is expressed in terms of `p`, as `o` and `i` were expressed in terms of `s`. Next, note that the `map` combinators and the auxiliary functions `hLift` and `hUnlift` are used to delegate work between successive levels of the tree defining a hereditarily finite set.

To summarize, let us observe that the successor and predecessor operations `s,p` at a given level are implemented through iteration of the same at a lower level and that the "left shift" operation implemented by `o,i` results in initiating `s` operations at a lower level. Thus the total number of operations is within a constant factor of the size of the trees.

And one can now also infer that as applying `s` and `p` on multiple branches are all independent operations, the algorithm begs for parallel execution, possibly in the form of FPGA hardware.

Finally, let us verify that these operations mimic indeed their more common counterparts on type `A`.

```
*SharedAxioms> view (A 42) :: S
S [S [S []],S [S [],S [S []]],S [S [],S [S [S []]]]]
*SharedAxioms> p it
S [S [],S [S [],S [S []]],S [S [],S [S [S []]]]]
*SharedAxioms> view it :: A
A 41

*SharedAxioms> view (A 5) :: S
S [S [],S [S [S []]]]
*SharedAxioms> o it
S [S [],S [S []],S [S [],S [S []]]]
*SharedAxioms> view it :: A
A 11
```

A proof by induction that types A and S implement indeed the same successor and predecessor operation as the instance Peano can be carried out with a proof assistant like Coq or ACL2.

Let us note that this implementation of the class Polymath implicitly uses the *Ackermann interpretation* of Peano arithmetics in terms of the theory of hereditarily finite sets, i.e. the natural number associated to a hereditarily finite set is given by the function

$$f(x) = \texttt{if}\ x = \emptyset\ \texttt{then}\ 0\ \texttt{else}\ \sum_{a \in x} 2^{f(a)}$$

We will see later, through a reflection mechanism that parameterizes the mapping from a set of natural numbers to a natural number, that we can generalize this to a family of interpretations.

Let us summarize what's unusual with instance S of the class Polymath: it shows that successor and predecessor operations can be performed with *hereditarily finite sets playing the role of natural numbers*. As natural numbers and finite ordinals are in a one-to-one mapping, this instance shows that hereditarily finite sets can be seen as finite ordinals directly, without using the simple but computationally explosive von Neumann construction (which defines ordinal n as the set $\{0, 1, \ldots, n-1\}$).

We will now provide an instance defined in terms of a more efficient hereditarily finite construct, likely to be usable for parallel hardware implementations of arithmetic operations.

5. Computing with Hereditarily Finite Functions

Hereditarily finite functions, described in detail in [Tarau 2009b], extend the inductive mechanism used to build hereditarily finite sets to finite functions on natural numbers (conveniently represented as finite sequences i.e. lists of natural numbers in Haskell). They are expressed using a similar datatype, denoted F here. The key difference is that, in this case, order is important, and that identical elements can occur at each level. Hereditarily finite functions can also be seen as compressed encodings of hereditarily finite sets, where, at each level, only increments between elements are represented. The first set of operations are similar to the ones on the type S:

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

instance Polymath F where
  e= F []

  o_ (F (F []:_))=True
  o_ _ = False

  o (F xs) = F (e:xs)

  i (F xs) = s (F (e:xs))
```

The code for r, s and p is also similar to the one given for hereditarily finite sets, except that this time s and p are co-recursive and r needs to do some padding with 0 and, as expected, some p operations.

```
  r (F (x:xs)) | e_ x = F xs
  r (F (k:xs)) = F (hzeros (p k) ++ (hnext xs)) where
    hzeros x | e_ x = []
    hzeros x = e : (hzeros (p x))

    hnext [] = []
    hnext (k:xs) = (s k):xs

  s (F xs) = F (hinc xs) where
    hinc ([]) = [e]
    hinc (x:xs) | e_ x= (s k):ys where (k:ys)=hinc xs
    hinc (k:xs) = e:(p k):xs

  p (F xs) = F (hdec xs) where
    hdec [x] | e_ x= []
    hdec (x:k:xs) | e_ x= (s k):xs
    hdec (k:xs) = e:(hdec ((p k):xs))
```

As with the type S, the total number of operations is proportional to the size of the trees. Given that F-trees are signifcantly smaller than S-trees, various operations will perform significantly faster, as in this representation only "increments" or "decrements" from one subtree to the next are computed (functions hinc and hdec). One can also observe that parallelization of the algorithm can be achieved by adapting *parallel prefix sum* computations as in [Misra 1994]. A few examples follow:

```
*SharedAxioms> view (A 42) :: S
S [S [S []],S [S [],S [S []]],S [S [],S [S [S []]]]]
*SharedAxioms> view (A 42) :: F
F [F [F []],F [F []],F [F []]]
*SharedAxioms> s it
F [F [],F [],F [F []],F [F []]]
*SharedAxioms> view it :: A
A 43
*SharedAxioms> view (A 5) :: F
F [F [],F [F []]]
*SharedAxioms> o it
F [F [],F [],F [F []]]
*SharedAxioms> view it :: A
A 11
```

6. Arithmetic operations

Our next refinement adds key arithmetic operations in the form of a type class extending Polymath. We start with addition:

```
class (Polymath n) => Polymath1 n where
  a :: n->n->n
  a x y | e_ x = y
  a x y | e_ y = x
  a x y | o_ x && o_ y = i (a (r x) (r y))
  a x y | o_ x && i_ y = o (s (a (r x) (r y)))
  a x y | i_ x && o_ y = o (s (a (r x) (r y)))
  a x y | i_ x && i_ y = i (s (a (r x) (r y)))
```

It is time to cheat on subtraction sb and comparison lt (standing for *less than*) - we only provide here simple/slow/short Peano-style implementations

```
  sb :: n->n->n
  sb x y | e_ x = e
  sb x y | e_ y = x
  sb x y = sb (p x) (p y)

  lt :: n->n->Bool
  lt x y | e_ x && e_ y = False
  lt x y | e_ x && not (e_ y) = True
  lt x y | not (e_ x) && e_ y = False
  lt x y = lt (p x) (p y)
```

and leave as an exercise to the reader to define (along the lines of a) more efficient ones. Note that sb is defined as a *total* function that computes the absolute value of the difference of the two numbers. Note also that lt implements a strict total order. We are now ready for a sorting operation, derived from Haskell's parametric sortBy. We define our sorting function nsort as follows:

```
nsort :: [n]→[n]
nsort ns = sortBy ncompare ns

ncompare :: n→n→Ordering
ncompare x y | x==y = EQ
ncompare x y | lt x y = LT
ncompare _ _ = GT
```

After adding the instances

```
instance Polymath1 A
instance Polymath1 Peano
instance Polymath1 BitStack
instance Polymath1 S
instance Polymath1 F
```

one can see that all operations extend naturally:

```
*SharedAxioms> a (Succ Zero) (Succ Zero)
Succ (Succ Zero)
*SharedAxioms> (s.s.s.s) Empty
Bit1 (Bit0 Empty)
*SharedAxioms> a (A 32) (A 10)
A 42
*SharedAxioms> lt (A 2009) (A 2010)
True
*SharedAxioms> nsort [A 3,A 2,A 1,A 2]
[A 1,A 2,A 2,A 3]
*SharedAxioms> lt (S []) (S [S [],S []])
True
```

The last operation shows now that we have a *total order* on hereditarily finite sets without recurse to the von Neumann ordinal construction used in [Kaye and Wong 2007] to complete the bi-interpretation from hereditarily finite sets to natural numbers. This replicates a recent result described in [Pettigrew] where a lexicographic ordering is used to simplify the proof of bi-interpretability of [Kaye and Wong 2007].

7. Shapeshifting between lists, multisets and sets - seen as reflections

We will now lay ground for reflecting sets, multisets and lists of natural numbers by first showing that one can freely move between them with as little as the operations defined so far. And we will leave the more difficult problem of fusing any of the above into a single natural number for later. The key idea is that *prefix sums* of lists of natural numbers can be used to express multisets and then sets. We refer to [Tarau 2009b] for more details on such mappings, but the computations involved are surprisingly straightforward:

```
class (Polymath1 n) ⇒ Polymath2 n where
  as_mset_list :: [n]→[n]
  as_mset_list ns = tail (scanl a e ns)

  as_list_mset :: [n]→[n]
  as_list_mset ms =
    zipWith sb ms' (e: ms') where ms'=nsort ms

  as_set_list :: [n]→[n]
  as_set_list = (map p) . as_mset_list . (map s)

  as_list_set :: [n]→[n]
  as_list_set = (map p) . as_list_mset . (map s)
```

After adding the instance declarations

```
instance Polymath2 A
instance Polymath2 Peano
instance Polymath2 BitStack
instance Polymath2 S
instance Polymath2 F
```

one can observe how the mappings work:

```
*SharedAxioms> as_mset_list [A 5,A 2,A 0,A 0,A 4]
[A 5,A 7,A 7,A 7,A 11]
*SharedAxioms> as_list_mset it
[A 5,A 2,A 0,A 0,A 4]
*SharedAxioms> as_set_list [A 5,A 2,A 0,A 0,A 4]
[A 5,A 8,A 9,A 10,A 15]
*SharedAxioms> as_list_set it
[A 5,A 2,A 0,A 0,A 4]
```

Note that only a weak subset of arithmetics has been used so far, i.e. no multiplications, divisions or exponentiations were involved in any of our previously described constructs.

We will proceed now with introducing more powerful operations. Needless to say, they will apply automatically to all instances of the type class Polymath.

8. Adding other arithmetic operations

We first define multiplication and integer division.

```
class (Polymath2 n) ⇒ Polymath3 n where
  m :: n→n→n  -- multiplication
  m x _ | e_ x = e
  m _ y | e_ y = e
  m x y = s (m0 (p x) (p y)) where
    m0 x y | e_ x = y
    m0 x y | o_ x = o (m0 (r x) y)
    m0 x y | i_ x = s (a y  (o (m0 (r x) y)))

  db :: n→n -- double
  db = p . o

  hf :: n→n -- half
  hf = r . s

  exp2 :: n→n -- power of 2

  exp2 x | e_ x = u
  exp2 x = db (exp2 (p x))

  -- simple (slow) division with reminder
  sd :: n→n→(n,n)
  sd x y = (q,p r) where
    (q,r) = sd' (s x) y
    sd' x y | e_ x = (e,e)
    sd' x y = z where
      x_y=sb x y
      z=if e_ x_y
        then (e,x)
        else (s q,r) where (q,r)=sd' x_y y
```

Next we define a mapping to conventional binary numbers - which support some operations more conveniently that our bijective base-2 representation used so far. Note that both representations use the "less significant digit first" convention.

```
  to_bin :: n→[n]
  to_bin x | e_ x = []
  to_bin x | o_ x = u: (to_bin (hf x))
  to_bin x = e:  (to_bin (hf x))

  from_bin :: [n]→n
  from_bin [] = e
  from_bin (x:xs) | u_ x = o (from_bin xs)
  from_bin (x:xs) | e_ x = db (from_bin xs)
```

After defining instances

```
instance Polymath3 A
instance Polymath3 Peano
instance Polymath3 BitStack
instance Polymath3 S
instance Polymath3 F
```

operations can be tested under various representations

```
*SharedAxioms> view (A 6) :: F
F [F [F []],F []]
*SharedAxioms> m it it
F [F [F [F []]],F [F [F []]]]
*SharedAxioms> view it :: A
A 36

*SharedAxioms> view it :: BitStack
Bit1 (Bit0 (Bit1 (Bit0 (Bit0 Empty))))
*SharedAxioms> sd it (Bit1 Empty) :: (BitStack,BitStack)
(Bit1 (Bit1 (Bit0 (Bit0 Empty))),Empty)
*SharedAxioms> view (fst it) :: A
A 18

*SharedAxioms> to_bin (A 3)
[A 1,A 1]
*SharedAxioms> from_bin it
A 3
*SharedAxioms> view it :: BitStack
Bit0 (Bit0 Empty)
```

We will next introduce *pairing functions*. They are used to parameterize mappings between finite sets and natural numbers.

9. Pairing functions

A *pairing* function is an bijection $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Its inverse is called *unpairing*. We represent a pairing/unpairing function as a record containing the pairing function p2 and its two unpairing counterparts p0 and p1.

```
data Pairing n =
     Pairing {p2 :: (n→n→n), p0 :: n→n, p1 :: n→n}
```

9.1 Basic pairing functions

Our next extension will provide a sampler of pairing functions, with emphasis on efficiently computable ones. We first define a classic pairing function, denoted **ppair**, together with its left and right unpairing companions **pfirst** and **psecond** that have been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem [Pepis 1938, Kalmar 1939, Robinson 1950]. The function **ppair** combines two numbers reversibly by multiplying a power of 2 derived from the first and an odd number derived from the second:

$$f(x, y) = 2^x(2y + 1) - 1 \tag{1}$$

It follows from the unique decomposition of a natural number as a product of prime factors that this function is invertible. Its inverse is provided by `pfirst` and `psecond` and the 3 functions are grouped together as the record `ppairing`.

```
class (Polymath3 n) ⇒ Polymath4 n where
  ppairing :: Pairing n
  ppairing = Pairing {p2=ppair,p0=pfirst,p1=psecond}

  ppair :: n→n→n
  ppair x y = p (lcons x y) where
    lcons x ys = s (lcons' x (db ys))
    lcons' x ys | e_ x = ys
    lcons' x ys = o  (lcons' (p x) ys)

  pfirst :: n→n
  pfirst z = lhead (s z) where
    lhead = h . p
    h xs | o_ xs = s (h (hf xs))
    h _ = e

  psecond :: n→n
  psecond z = ltail (s z) where
    ltail =  hf . t . p
    t xs | o_ xs = t (hf xs)
    t xs  = xs
```

The next pairing function works in a way similar to the `zip` operation on *powerlists* described in [Misra 1994]: it merges and unmerges two sequences of bits. In contrast to [Misra 1994], we are not enforcing the same length constraint on the two operands. Instead, padding with `e`, our null element, is used when needed.

```
  bpairing :: Pairing n
  bpairing = Pairing {p2=bpair,p0=bfirst,p1=bsecond}

  bpair :: n → n → n
  bpair x y = from_bin (bpair' (to_bin x) (to_bin y)) where
    bpair' [] [] = []
    bpair' [] ys = e:(bpair' ys [])
    bpair' (x:xs) ys = x:(bpair' ys xs)

  bfirst ::  n → n
  bfirst = from_bin . deflate . to_bin

  bsecond :: n → n
  bsecond = from_bin . second' . to_bin where
    second' [] = []
    second' (_:xs) = deflate xs

  deflate :: [n]→ [n]
  deflate [] = []
  deflate (x:_:xs) = x:(deflate xs)
  deflate [x] = [x]
```

After adding the instances

```
instance Polymath4 A
instance Polymath4 Peano
instance Polymath4 BitStack
instance Polymath4 S
instance Polymath4 F
```

one can observe the action of the pairing functions on various representations:

```
*SharedAxioms> bpair (A 3) (A 4)
A 37
*SharedAxioms> bfirst (A 37)
A 3
*SharedAxioms> bsecond (A 37)
A 4
*SharedAxioms> ppair (A 3) (A 4)
A 71
*SharedAxioms> pfirst (A 71)
A 3
*SharedAxioms> psecond (A 71)
A 4
```

9.2 Parameterizing on a pairing function

We will parameterize our next extension layer `Polymath5` to depend on a pairing/unpairing function that can be customized by various instances.

```
class (Polymath4 n) ⇒ Polymath5 n where
  pairing :: Pairing n
  pairing = ppairing -- default pairing - to override
```

```
first :: n→n
first = p0 pairing

second :: n→n
second = p1 pairing

pair :: n→n→n
pair = p2 pairing
```

We will now derive a list representation, parameterized by our pairing function. Set and multiset representations will be derived using their mappings to lists.

```
hd ::  n → n
hd n = first (p n)

tl :: n → n
tl n = second (p n)

cons :: n → n → n
cons x y = s (pair x y)

as_list_nat :: n→[n]
as_list_nat x | e_ x = []
as_list_nat x = hd x : as_list_nat (tl x)

as_nat_list :: [n]→n
as_nat_list [] = e
as_nat_list (x:xs) = cons x (as_nat_list xs)
```

As we have already a mapping between lists and sets, we will use it to map sets to natural numbers.

```
as_nat_set :: [n]→n
as_nat_set = as_nat_list . as_list_set

as_set_nat :: n→[n]
as_set_nat = as_set_list . as_list_nat
```

The mapping to multisets is derived in a similar way:

```
as_nat_mset :: [n]→n
as_nat_mset = as_nat_list . as_list_mset

as_mset_nat :: n→[n]
as_mset_nat = as_mset_list . as_list_nat
```

9.3 Deriving edge types from a pairing function

We will now put at work our transformers between sets, multisets and lists to derive, from a given pairing function, representations for specific edge types, i.e. ordered pairs for digraphs, unordered pairs for unordered graph and "upward pointing" edges for canonically represented DAGs. They will be used later to derive transformations to/from various graph types.

```
ordUnpair :: n→(n,n)
ordUnpair z = (first z,second z)

ordPair :: (n,n)→n
ordPair (x,y) = pair x y

unordUnpair :: n→(n,n)
unordUnpair z = (x',y') where
  (x,y)=ordUnpair z
  [x',y']=as_mset_list  [x,y]

unordPair :: (n,n)→n
unordPair (x,y) = ordPair (x',y') where
  [x',y']=as_list_mset [x,y]

upwardUnpair :: n→(n,n)
upwardUnpair z = (x',y') where
  (x,y)=ordUnpair z
  [x',y']=as_set_list [x,y]

upwardPair :: (n,n)→n
upwardPair (x,y) = ordPair (x',y') where
  [x',y']=as_list_set [x,y]
```

After adding the instances

```
instance Polymath5 A
instance Polymath5 Peano
instance Polymath5 BitStack
instance Polymath5 S
instance Polymath5 F
```

we can see their action as follows:

```
*SharedAxioms> ordUnpair 119
(3,7)
*SharedAxioms> unordUnpair 119
(3,10)
*SharedAxioms> upwardUnpair 119
(3,11)

*SharedAxioms> as_list_nat (A 2010)
[A 1,A 1,A 0,A 1,A 0,A 0,A 0,A 0]
*SharedAxioms> as_mset_nat (A 2010)
[A 1,A 2,A 2,A 3,A 3,A 3,A 3,A 3]
*SharedAxioms> as_set_nat (A 2010)
[A 1,A 3,A 4,A 6,A 7,A 8,A 9,A 10]
*SharedAxioms> as_nat_set it
A 2010

*SharedAxioms> view (A 6) :: S
S [S [S []],S [S [S []]]]
*SharedAxioms> as_set_nat it
  [S [S []],S [S [S []]]]
```

The last example also shows that the tree representing a hereditarily finite set maps to the forest growing out if its root. The deeper reason for this is that: *the default pairing function* `ppairing` *induces the Ackermann interpretation as the bijection between* $\mathbb{N}$ *and* ZF^*. This follows from the fact that the `hd` operation induced by `ppairing` computes at each step the distance to the next element at bit i that is on, corresponding to 2^i in Ackermann's mapping.

We are now ready to add a set theory layer. For convenience, we will use Haskell lists as intermediate representations, although they can be eliminated with deforestation transformations.

10. Deriving Set Operations

We first introduce combinators that will take advantage of our reflected set operations generically.

```
class (Polymath5 n) ⇒ Polymath6 n where
  setOp1 :: ([n]→[n])→(n→n)
  setOp1 f = as_nat_set . f . as_set_nat
  setOp2 :: ([n]→[n]→[n])→(n→n→n)
  setOp2 op x y = as_nat_set
    (op (as_set_nat x) (as_set_nat y))
```

We can now use them to "borrow" the usual set operations (provided in the Haskell package Data.List):

```
setIntersection :: n→n→n
setIntersection = setOp2 intersect

setUnion :: n→n→n
setUnion = setOp2 union

setDifference :: n→n→n
setDifference = setOp2 (\\)

setIncl :: n→n→Bool
setIncl x y = x==(setIntersection x y)
```

In a similar way, we define a powerset operation conveniently using actual lists, before reflecting it into an operation on natural numbers.

```
powset :: n→n
powset x = as_nat_set
  (map as_nat_set (subsets (as_set_nat x))) where
    subsets [] = [[]]
    subsets (x:xs) =
       [zs|ys←subsets xs,zs←[ys,(x:ys)]]
```

Next, the ϵ-relation defining set membership is given as the function `inSet`, together with the `augment` function used in various set theoretic constructs as a new set generator.

```
inSet :: n→n→Bool
inSet x y = setIncl (as_nat_set [x]) y

augment :: n→n
augment x = setUnion x (as_nat_set [x])
```

The nth von Neumann *ordinal* n is the set $\{0, 1, \ldots, n-1\}$) and is used to emulate natural numbers in finite set theory. It is implemented by the function `nthOrdinal`:

```
nthOrdinal :: n→n
nthOrdinal x | e_ x = e
nthOrdinal n = augment (nthOrdinal (p n))
```

After defining the appropriate instances

```
instance Polymath6 A
instance Polymath6 Peano
instance Polymath6 BitStack
instance Polymath6 S
instance Polymath6 F
```

we observe that set operations act naturally under the hereditarily finite set interpretation:

```
*SharedAxioms> view (A 6) :: S
S [S [S []],S [S [S []]]]
*SharedAxioms> inSet (S [S []]) it
True

*SharedAxioms> powset (S [])
S [S []]
*SharedAxioms> powset it
S [S [],S [S []]]

*SharedAxioms> augment (S [])
S [S []]
*SharedAxioms> augment it
S [S [],S [S []]]

*SharedAxioms> map nthOrdinal [0..4]
[0,1,3,11,2059]
*SharedAxioms> as_set_nat 2059
[0,1,3,11]
```

11. Deriving List Operations

Like in the case of sets, we first introduce combinators that will take advantage of our reflected set operations generically.

```
class (Polymath6 n) ⇒ Polymath7 n where
  listOp1 :: ([n]→[n])→(n→n)
  listOp1 f = as_nat_list . f . as_list_nat
  listOp2 :: ([n]→[n]→[n])→(n→n→n)
  listOp2 op x y = as_nat_list
    (op (as_list_nat x) (as_list_nat y))
```

We can now use them to "borrow" the usual list operations:

```
  listConcat :: n→n→n
  listConcat = listOp2 (++)

  listReverse :: n→n
  listReverse = listOp1 reverse
```

Another mechanism for defining list operations is to use a "structured recursion combinator" like `foldr` from which various other operations can be derived.

```
  listFoldr :: (n → n → n) → n → n → n

  listFoldr f z xs | e_ xs    =  z
  listFoldr f z xs =  f (hd xs) (listFoldr f z (tl xs))

  listConcat' :: n→n→n
  listConcat' xs ys = listFoldr cons ys xs

  listSum :: n→n
  listSum = listFoldr a u

  listProduct :: n→n
  listProduct = listFoldr m u
```

After defining the appropriate instances

```
instance Polymath7 A
instance Polymath7 Peano
instance Polymath7 BitStack
instance Polymath7 S
instance Polymath7 F
```

we observe that list operations act naturally under the hereditarily finite function interpretation:

```
*SharedAxioms> view (A 6) :: F
F [F [F []],F []]
*SharedAxioms> listReverse it
F [F [],F [F []]]

*SharedAxioms> listConcat (F [F []]) (F [F []])
F [F [],F []]
*SharedAxioms> listConcat' (F [F []]) (F [F []])
F [F [],F []]
```

12. Alternative List, Set and Multiset Interpretations

As our reflected list, set and multiset theories are parameterized by the pairing function, we can easily obtain alternative theories when instances make different choices. We now define a type `B` that mimics the type `A` introduced previously, except for the choice of its pairing function, as instance of `Polymath5`.

```
newtype B = B Integer deriving (Eq,Show,Read)

instance Polymath B where
  e = B 0
  o_ (B x) = odd x
  o (B x) = B (2*x+1)
  i (B x) = B (2*x+2)
  r (B x) | x/=0 = B ((x-1) `div` 2)

instance Polymath1 B
instance Polymath2 B
instance Polymath3 B
instance Polymath4 B
instance Polymath5 B where pairing=bpairing
instance Polymath6 B
instance Polymath7 B
```

As expected, the `pair` function acts differently:

```
*SharedAxioms> pair (A 4) (A 5)
A 175
*SharedAxioms> pair (B 4) (B 5)
B 50
```

One can see that this different behavior propagates to set and multiset operations:

```
*SharedAxioms> as_set_nat (A 42)
[A 1,A 3,A 5]
*SharedAxioms> as_set_nat (B 42)
[B 1,B 5]

*SharedAxioms> as_mset_nat (A 42)
[A 1,A 2,A 3]
*SharedAxioms> as_mset_nat (B 42)
[B 1,B 4]

*SharedAxioms> map (powset . A) [0..7]
[A 1,A 3,A 5,A 15,A 17,A 51,A 85,A 255]
*SharedAxioms> map (powset . B) [0..7]
[B 1,B 3,B 9,B 139,B 2057,B 515,B 521,B 651]
```

The last example is somewhat interesting in the sense that while Cantor's inequality $x < powset\ x$ holds (as expected from being a model for ZF^*), it is not true anymore that $x < y \Rightarrow powset\ x < powset\ y$. One can also observe that while ordinals look different in the interpretation B, their defining property still holds as expected:

```
*SharedAxioms> map (nthOrdinal.B) [0..4]
[B 0,B 1,B 3,B 131,B 10141359546691965155289593839747]
*SharedAxioms> as_set_nat
   (B 10141359546691965155289593839747)
[B 0,B 1,B 3,B 131]
```

i.e. the 4-th ordinal is in fact the set of its predecessors.

13. Deriving an instance with fast bitstring operations

We will now benefit from our shared axiomatization by designing an instance that takes advantage of bit operations to implement, through a few overrides, fast versions of various operations. For syntactic convenience, we will map this instance directly to Haskell's arbitrary length Integer type to benefit in GHC from the performance of the underlying C-based GMP package. First some arithmetic operations:

```
instance Polymath Integer where
  e = 0
  o_ x = testBit x 0

  o x = succ (shiftL x 1)
  i  = succ . o
  r x | x/=0 = shiftR (pred x) 1

  s = succ
  p = pred
  u = 1
  u_ = (== 1)

instance Polymath1 Integer where
  sb x y = abs (x-y)
  lt = (<)
  nsort = sort
  ncompare=compare

instance Polymath2 Integer
instance Polymath3 Integer where
  m = (*)
  hf x = shiftR x 1
  db x = shiftL x 1
  sd = quotRem
```

```
instance Polymath4 Integer
instance Polymath5 Integer
```

Next, some set operations:

```
instance Polymath6 Integer where
  setUnion = (.|.)
  setIntersection = (.&.)
  setDifference x y = x .&. (complement y)

  inSet x xs = testBit xs (fromIntegral x)

  powset 0 = 1
  powset x = y `xor` (shiftL y 1) where y=powset (pred x)

instance Polymath7 Integer
```

It is tempting to test for correctness, by comparing the "specification" given by the type `A` and the "implementation" provided by the type `Integer`:

```
*SharedAxioms> map powset [0..7]
[1,3,5,15,17,51,85,255]
*SharedAxioms> map (powset . A) [0..7]
[A 1,A 3,A 5,A 15,A 17,A 51,A 85,A 255]
```

In fact, we do not have a proof yet that the `xor` based instance of `powset` does in fact implement a powerset (assuming the Ackermann interpretation), except for finding out that they happen to map to the same sequence `A001317` in [Sloane 2006] - so this looks like an interesting *open problem*. On the other hand, a similar, purely arithmetic definition, has been shown equivalent under the Ackermann interpretation [Abian and Lamacchia 1965]:

```
powset' i = product (map (\k->1+2^(2^k)) (as_set_nat i))
```

Like the `xor` based definition this would also work differently under the interpretations induced by different pairing functions, for instance `bpairing`.

14. Virtualizing reflected datatypes with a groupoid of isomorphisms

We have seen that a number of conversion operations made it into our type classes, like `as_set_list, as_list_nat, as_nat_set`, etc. Clearly, this pattern begs for a more general combinator language. We will now adapt the construction described in [Tarau 2009b] while emphasizing its ability to virtualize our reflected datatypes as transformations to a shareable common representation.

We start by adapting to our type class chain the framework described in [Tarau 2009b] that provides bijective any-to-any conversions between various data types together with a general mechanism for transporting their operations.

14.1 Connecting data types with a groupoid of isomorphisms

A category in which every morphism is an *isomorphism* is called a *groupoid*. We represent *isomorphism* pairs as a data type `Iso`, together with the operations `compose, itself` and `invert` providing together a *groupoid* structure.

```
data Iso a b = Iso (a->b) (b->a)

compose :: Iso a b -> Iso b c -> Iso a c
compose (Iso f g) (Iso f' g') = Iso (f' . f) (g . g')

itself = Iso id id
invert (Iso f g) = Iso g f
```

We will put at work these combinators by designing bijections between various data types. They transport operations and are invertible. This justifies seeing them as *isomorphisms* between data types. Such bijections are typed, therefore f and g are composable morphisms only if the target of f is identical with the source of g. These two considerations make the “natural” structure hosting them a *groupoid*.

14.2 Any-to-any isomorphisms in a connected groupoid

Assuming our isomorphisms form a *connected groupoid* it makes sense at this point to route them through a *hub* data type to avoid having to provide $\frac{n(n-1)}{2}$ isomorphisms.

A possible choice for such a hub is $\mathbb{N}$ as provided by the efficiently implemented instance Integer of the Polymath type classes. In fact, any other instance can be chosen as hub. In a context where, for instance, a hardware based parallel implementation based on hereditarily finite functions is available, the datatype F would be a better choice to play this role.

We call *virtual datatype* an isomorphism from a concrete datatype to our hub. It can be seen as a more flexible reflection of its underlying datatype in the sense that it can shapeshift into any other virtual datatype connected to the hub. The type ``Type'' will be used for a virtual datatype to indicate the analogy with the concrete *type* it replaces.

```
type Type a = Iso a Integer
```

We first define a trivial virtual type representing the hub itself:

```
nat :: Type Integer
nat = itself
```

One can route isomorphisms between two virtual types through the hub using the combinator as:

```
as :: Type a → Type b → b → a
as that this x = g x where
   Iso _ g = compose that (invert this)
```

A one argument function f is transported between virtual types using the combinator borrow_from:

```
borrow_from :: Type b → (b → b) →
               Type a → a → a

borrow_from lender f borrower =
  (as borrower lender) . f . (as lender borrower)
```

Similarly, a two argument function op is transported between virtual types using the combinator borrow_from2:

```
borrow_from2 :: Type a → (a → a → a) →
                Type b → b → b → b

borrow_from2 lender op borrower x y =
    as borrower lender r where
       x'= as lender borrower x
       y'= as lender borrower y
       r = op x' y'
```

We will now populate our universe of virtual types with list, set and multiset types.

```
list :: Type [Integer]
list = Iso  as_nat_list as_list_nat

set :: Type [Integer]
set = Iso  as_nat_set as_set_nat

mset :: Type [Integer]
mset = Iso  as_nat_mset as_mset_nat
```

One can now try out the combinator ``as'' working exactly like its concrete counterparts:

```
*SharedAxioms> as_set_nat 2009
[0,3,4,6,7,8,9,10]
*SharedAxioms> as set nat 2009
[0,3,4,6,7,8,9,10]
*SharedAxioms> as list nat 2009
[0,2,0,1,0,0,0,0]
*SharedAxioms> as_list_nat 2009
[0,2,0,1,0,0,0,0]
```

15. Directed Graphs, DAGs, Undirected graphs and Hypergraphs

We will now show that more complex data types like digraphs, unordered graphs, DAGs and hypergraphs have extremely simple virtual types. The mechanism for deriving them is surprisingly uniform. And if one is a believer in Occam's Razor, this can be used as an a posteriori justification for their popularity.

15.1 Set Encodings of Directed Graphs

We can find a bijection from directed graphs to finite sets by fusing their list of ordered pairs representation into finite sets, with a pairing function. We will also add one more layer to our Polymath classes to allow sharing transformations to/from graphs among various implementations.

```
class (Polymath7 n) ⇒ Polymath8 n where

  as_set_digraph :: [(n,n)]→[n]
  as_set_digraph = map ordPair

  as_digraph_set :: [n]→[(n,n)]
  as_digraph_set = map ordUnpair
```

15.2 Set Encodings of Undirected Graphs

Likewise, we can find a bijection from undirected graphs to finite sets using unordered pairs.

```
  as_set_graph :: [(n,n)]→[n]
  as_set_graph = map unordPair

  as_graph_set :: [n]→[(n,n)]
  as_graph_set = map unordUnpair
```

15.3 Set Encodings of DAGs

One can derive an encoding as sets of natural numbers of directed acyclic graphs (DAGs) under the assumption that they are canonically represented by pairs of edges such that the first element of the pair is strictly smaller.

```
  as_set_dag :: [(n,n)]→[n]
  as_set_dag  = map upwardPair

  as_dag_set :: [n]→[(n,n)]
  as_dag_set = map upwardUnpair
```

15.4 Encoding Hypergraphs

A *hypergraph* (also called *set system*) is a pair $H = (X, E)$ where X is a set and E is a set of non-empty subsets of X.

We can easily derive a bijective encoding of em hypergraphs, represented as sets of sets (with $\emptyset$ taken out by applying s first).

```
  as_hypergraph_set :: [n]→[[n]]
  as_hypergraph_set = map (as_set_nat . s)

  as_set_hypergraph :: [[n]]→[n]
  as_set_hypergraph = map (p . as_nat_set)
```

We conclude this by updating our instance definitions

```
instance Polymath8 A
instance Polymath8 Peano
instance Polymath8 BitStack
instance Polymath8 S
instance Polymath8 F
instance Polymath8 B
instance Polymath8 Integer
```

15.5 Virtual Types for Various Graphs

After defining a pair type based on our most efficient instance of Polymath

```
type N2=(Integer,Integer)
```

(to which we commit from now on as a basis for our virtual graph types), we start with a virtual type for `digraphs`

```
digraph :: Type [N2]
digraph = compose (Iso as_set_digraph as_digraph_set) set
```

working as follows:

```
*SharedAxioms> as digraph nat 2010
[(1,0),(2,0),(0,2),(0,3),(3,0),(0,4),(1,2),(0,5)]
*SharedAxioms> as nat digraph it
2010
*SharedAxioms> as nat digraph [(0,0),(2,0)]
9
```

We can also derive a virtual type for unordered graphs

```
graph :: Type [N2]
graph = compose (Iso as_set_graph as_graph_set) set
```

working as follows:

```
*SharedAxioms> as graph nat 2010
[(1,1),(2,2),(0,2),(0,3),(3,3),(0,4),(1,3),(0,5)]
*SharedAxioms> as nat graph it
2010
```

Note that, as expected, the result is invariant to changing the order of elements in the pairs.

The virtual type for DAGs is:

```
dag :: Type [N2]
dag = compose (Iso as_set_dag as_dag_set) set
```

working as follows:

```
*SharedAxioms> as dag nat 2010
[(1,2),(2,3),(0,3),(0,4),(3,4),(0,5),(1,4),(0,6)]
*SharedAxioms> as nat dag it
2010
```

As digraphs, unordered graphs and DAGs with the same number of edges originate from the same set associated to a natural number, we can conclude that we have constructed pairs of bijections between them that preserve the number of edges.

Finally the derived virtual type for hypergraphs is:

```
hypergraph :: Type [[Integer]]
hypergraph =
  compose (Iso as_set_hypergraph as_hypergraph_set) set
```

working as follows

```
*SharedAxioms> as hypergraph nat 2010
[[1],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]]
*SharedAxioms> as nat hypergraph it
2010
```

Encoding graph types as natural numbers can provide succinct representations and perfect hash-keys for graph indexing. Our virtual types are also useful as iterators for enumerating progressively larger and larger objects and to generate random instances of a given type. We refer to [Tarau 2009a] for other encodings covering more than 60 data types.

16. A performance test: the Syracuse function

We will now use a variant of the 3x+1 problem / Collatz conjecture / Syracuse function [Lagarias 2008] that, somewhat surprisingly, can be expressed as a mix of arithmetic operations and reflected list / set 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` always terminates:

```
syr n = tl (a (m six n) four) where
  four = s (s (s (s e)))
  six = s (s four)

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

The first 8 sequences are computed as follows:

```
*SharedAxioms> 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]]
```

Timing `nsyr` for 123456780, and then the same digits repeated twice and three times, for functions `cI, cA, cK, cF` and `cS` shows low polynomial growth in the bitsize of the inputs for the respective instances. It also indicates significant gains for hereditarily finite functions (col. `cF`) vs. hereditarily finite sets (col. `cS`) and of symbolic BitStack computations `cK` vs. "unaccelerated" Integer operations `cA`. Integer operations accelerated with overridings and bit operations `cI` are faster by constant factors that are significant, but not as dramatic as one might expect.

```
cI c = c :: Integer
cA c=view (c :: Integer) :: A
cK c=view (c :: Integer) :: BitStack
cF c=view (c :: Integer) :: F
cS c=view (c :: Integer) :: S
```

bitsize of input	cI	cA	cK	cF	cS
31	7	80	62	153	311
64	11	163	120	327	1084
97	17	366	261	720	3604

Figure 1. Timings for `cI, cA, cK, cF, cS` in milliseconds

17. Related work

The paper makes use of the embedded data transformation language introduced in [Tarau 2009a], a large unpublished draft, also organized as a literate Haskell program, a small subset of which has been published as [Tarau 2009b]. The digraph and hypergraph virtual types described in this paper make use of encodings similar to those in [Tarau 2009b]. However, the derivation presented here places the encodings in a more general framework, as virtual types parameterized by arbitrary pairing functions and and a generic set/-multiset/list/natural number type class. Our paper also adds new encodings for unordered graphs and DAGs and derives them from a uniform edge encoding mechanism.

Natural number encodings of hereditarily finite sets (that have been the main inspiration for our concept of hereditarily finite functions) have triggered the interest of researchers in fields ranging from Axiomatic Set Theory to Foundations of Logic [Takahashi 1976, Kaye and Wong 2007, Abian and Lamacchia 1978, Kirby 2007].

Pairing functions have been used in work on decision problems as early as [Pepis 1938, Kalmar 1939, Robinson 1950]. A typical modern use in the foundations of mathematics is [Cégielski and

Richard 2001]. An extensive study of various pairing functions and their computational properties is presented in [Rosenberg 2003].

A number of papers of J. Vuillemin develop similar techniques aiming to unify various data types, with focus on theories of boolean functions and arithmetics [Vuillemin 1994, 2003]

The closest references on encapsulating bijections as a Haskell data type are [Alimarine et al. 2005] and Conal Elliott's composable bijections module [Conal Elliott], where, in a more complex setting, *arrows* [Hughes] are used as the underlying abstractions. [Kahl and Schmidt 2000] uses a similar category theory inspired framework implementing relational algebra, also in a Haskell setting.

Binary number-based axiomatizations of natural number arithmetics are likely to be folklore, but having access to the the underlying theory of the calculus of constructions [Coquand and Huet 1988] and the inductive proofs of their equivalence with Peano arithmetics in the libraries of the `Coq` [The Coq development team 2004] proof assistant has been particularly enlightening to the author. On the other hand we have not found in the literature any such axiomatizations in terms of hereditarily finite sets or hereditarily finite functions, as described in this paper.

Some other techniques are for sure part of the scientific commons. In that case our focus was to express them as elegantly as possible in a uniform framework.

18. Conclusion and Future Work

In the form of a literate Haskell program, we have built "shared axiomatizations" of finite arithmetics, hereditarily finite sets and a few equivalent constructs using successive refinements of type classes.

Besides introducing a few new (and unusual) algorithms expressing arithmetic computations in terms of "symbolic structures" like hereditarily finite sets and hereditarily finite functions, our framework unifies fundamental mathematical concepts in a directly executable form.

The derivation of successive extensions as Haskell type classes, enjoying the joint benefits of a higher order functional programming language and a simple and flexible object oriented coding style, has shown the expressiveness and robustness of polymorphically typed functional languages. This has materialized in the form of *virtual types* encapsulating the ability to shapeshift between data representations at will, while enjoying the safety mechanisms and the convenience of Haskell's type inference.

More future work is needed to evaluate through applications the flexibility and the performance of the resulting data transformation framework. In [Tarau 2009a] a bijective mapping between BDDs and the natural numbers representing the truth tables obtained through their parallel evaluation is given. We are planning an emulation of arithmetics in terms of BDDs, similar to the ones described in this paper, as they seem likely to provide interesting boolean circuit algorithms for arbitrary length arithmetic operations. In [Tarau 2009b] a concept of hereditarily finite permutations is described. We plan to try out if arithmetic operations can be carried out with them in a way similar to our hereditarily finite set and function based emulations. This is particularly interesting, given that quantum computations require reversible circuits that can be described as compositions of bitvector permutations [Maslov et al. 2007].

References

Alexander Abian and Samuel Lamacchia. Some consequences of the axiom of power-set. *J. Symb. Log.*, 30(3):293–294, 1965.

Alexander Abian and Samuel Lamacchia. On the consistency and independence of some set-theoretical constructs. *Notre Dame Journal of Formal Logic*, X1X(1):155–158, 1978.

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, New York, NY, USA, 2005. ACM Press.

Patrick Cégielski and Denis Richard. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. Comput. Sci.*, 257(1-2):51–77, 2001.

Conal Elliott. Module: Data.Bijections. Haskell source code library at: http://haskell.org/haskellwiki/TypeCompose.

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

John Hughes. Generalizing Monads to Arrows. Science of Computer Programming 37, pp. 67-111, May 2000.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In *Haskell Workshop*, 1997.

Wolfram Kahl and Gunther Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000. URL `http://ist.unibw-muenchen.de/Publications/TR/2000-02/`.

Laszlo Kalmar. On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. *The Journal of Symbolic Logic*, 4(1):1–9, mar 1939. ISSN 0022-4812.

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

Laurence Kirby. Addition and multiplication of sets. *Math. Log. Q.*, 53(1): 52–65, 2007.

Jeffrey C. Lagarias. The 3x+1 Problem: An Annotated Bibliography (1963-1999), 2008. http://arXiv.org, 0309224v11.

Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. Techniques for the synthesis of reversible Toffoli networks. *ACM Trans. Design Autom. Electr. Syst.*, 12(4), 2007.

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

Jayadev Misra. Powerlist: a structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16:1737–1767, 1994.

Jozef Pepis. Ein verfahren der mathematischen logik. *The Journal of Symbolic Logic*, 3(2):61–76, jun 1938. ISSN 0022-4812.

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

Julia Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950. ISSN 0002-9939.

Arnold L. Rosenberg. Efficient pairing functions - and why you should care. *International Journal of Foundations of Computer Science*, 14(1):3–17, 2003.

N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2006. published electronically at www.research.att.com/∼njas/sequences.

Moto-o Takahashi. A Foundation of Finite Mathematics. *Publ. Res. Inst. Math. Sci.*, 12(3):577–708, 1976.

Paul Tarau. Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell, January 2009a. http://arXiv.org/abs/0808.2953, unpublished draft, 104 pages.

Paul Tarau. A Groupoid of Isomorphic Data Transformations. In J. Carette et al., editor, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , Grand Bend, Canada, July 2009b. Springer, LNAI 5625.

Jean Vuillemin. Digital algebra and circuits. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 733–746. Springer, 2003.

Jean Vuillemin. On circuits and numbers. *IEEE Trans. Computers*, 43(8): 868–879, 1994.