Declarative Modeling of Finite Mathematics

Paul Tarau

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

Abstract

A common foundation of finite arithmetic, hereditarily finite sets and sequences, binary trees and graphs is described as a progressive refinement of Haskell type classes.

We derive as instances symbolic implementations of arithmetic operations in terms of rooted ordered trees representing hereditarily finite sets and sequences. Conversely, arithmetic implementations of pairs, powersets, von Neumann ordinals shade new light on the bi-interpretability between Peano arithmetic and a theory of hereditarily finite sets.

As another instance, rooted ordered binary trees representing a simplified form of the type language of Gödel's System T are shown as directly emulating arithmetic and finite set operations.

The main contribution of the paper is a fully constructive unification of paradigms – a chain of type classes does it all: Peano arithmetic, sets, sequences, binary trees, bitstrings. Another contribution is that we do it "efficiently" (i.e. in time and space asymptotically comparable with standard binary representations).

The Haskell code in the paper is available at `http://logic.cse.unt.edu/tarau/research/2010/shared.hs`.

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

General Terms Algorithms, Languages, Theory

Keywords axiomatizations of arithmetic and set theoretical constructs, declarative modeling of finite mathematics, symbolic arithmetics, hereditarily finite sets and functions, arithmetic and set operations with types

1. Introduction

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 first order Peano arithmetic. Finite set theory is axiomatized with the usual Zermelo-Fraenkel system (abbreviated ZF) in which the Axiom of Infinity is replaced by its negation. When the axiom of ϵ-induction, (saying that if a property holds for all elements of a set then it holds for the set) is added, the resulting finite set theory (abbreviated ZF^*) is bi-interpretable with Peano arithmetic i.e. they emulate each other accurately through a bijective mapping that commutes with standard operations on the two sides ([Kaye and Wong 2007]).

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 take the liberty in this paper to explore some interesting properties of finite arithmetic 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 [Mayberry 2000, Takahashi 1976, Kaye and Wong 2007, Kirby 2007, Cégielski and Richard 2001].

More specifically, we provide a declarative modeling of of Peano arithmetic, bijective base-2 arithmetic, hereditarily finite sets and a few other equivalent constructs to progressively build basic programming language concepts like lists, sets, multisets, graphs. 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.

The resulting hierarchy of type classes describes incrementally *common computational capabilities* shared by data types representing Peano natural numbers, hereditarily finite sets, hereditarily finite functions and rooted ordered binary trees.

A number of *novel algorithms* (some fairly intricate like implementing arithmetic computations directly in terms of hereditarily finite sets, hereditarily finite functions in sections 4 and 5 or binary trees in section 15, are worth exploring in detail and analyzing in separate papers. However, *the main contribution of this paper is the framework that unifies fundamental mathematical concepts in a genuinely constructive (i.e. directly executable) form.*

The paper is organized as follows.

Section 2 provides some basic intuitions by introducing computations with *bijective base-2* bitstrings.

Section 3 explains the main idea of shared axiomatizations as type classes and introduces the class *Polymath*, its primitive operations and simple instances like bijective base-2 and Peano natural numbers.

Sections 4 and 5 describe computations with hereditarily finite sets and sequences.

Section 6 defines the ordering relation and section 7 studies related Galois connections.

Section 8 provides generic implementations of various arithmetic operations.

Sections 9 introduces pairing functions and section 10 describes transformations between collections, from which bijective mappings between sets, multisets, lists and natural numbers are derived in section 11. Generic set and list operations are derived in sections 12 and 13 followed by graph encodings in section 14.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

A new instance enabling arithmetic and set computations with binary trees is introduced in section 15.

Sections 16 and 17 discuss related work and conclude the paper.

2. Computing in bijective base-2

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 arithmetic and Finite Set Theory.

After defining our module and a few imports we start with the data type `OIs` seen as representing strings in the free monoid given by the regular language $\{0,1\}^*$.

```
module SharedAxioms where
import Data.List
import Test.QuickCheck

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

We define the following operations on 0-1 strings:

```
empty = E

withO xs = O xs
withI xs = I xs

reduce (O xs) = xs
reduce (I xs) = xs
```

and the predicates

```
isEmpty xs = xs == E

isO (O _) = True
isO _     = False

isI (I _) = True
isI _     = False
```

We remind the reader a few basic (but possibly not widely known) concepts related to the computation mechanism we use on bitstrings.

DEFINITION 1. *Bijective base-2 representation associates to $n \in \mathbb{N}$ a unique string in the regular language $\{0,1\}^*$ by removing the 1 indicating the highest exponent of 2 from the standard bitstring representation of $n+1$.*

We refer to `http://en.wikipedia.org/wiki/Bijective_numeration` for the historical origins of the concept and the more general *bijective base-k* case.

Using a list notation for bitstrings this gives: $0 = [], 1 = [0], 2 = [1], 3 = [0,0], 4 = [1,0], 5 = [0,1], 6 = [1,1]$ etc. which corresponds to $0 = E, 1 = OE, 2 = IE, 3 = O(OE), 4 = I(OE), 5 = O(IE), 6 = I(I(E))$ using the type `OIs`. Note that we assume here that bitstrings are mapped to numbers starting with the lowest exponent of 2 and ending with the highest. As a simple exercise in bijective base-2 arithmetic, one can implement the successor function - and therefore provide a model of Peano's axioms

```
zero = empty
one = withO empty

peanoSucc xs | isEmpty xs = one
peanoSucc xs | isO xs = withI (reduce xs)
peanoSucc xs | isI xs = withO (peanoSucc (reduce xs))
```

working as follows:

```
*SharedAxioms> (peanoSucc . peanoSucc . peanoSucc) zero
O (O E)
```

Using the `OIs` 0-1 bitstring representation (by contrast with naive "base-1" 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 defer defining these operations until the next sections, where we provide such implementations in a more general setting.

We will spare the reader from a similar exercise showing basic set operations on 0-1 bitstrings seen as characteristic functions of sets, and just conclude this section by saying, that in a nutshell, 0-1 strings as represented by the data type `OIs` promise to have the capabilities needed to emulate both Peano arithmetic and ZF-finite sets in a single framework.

3. The Analogy: Axiomatizations as Type Classes

We explore now in some detail the analogy between a programming language construct, Haskell's *type classes* [Wadler and Blott 1989], and a predicate logic construct with well established uses as a foundational tool, *axiom systems*.

While the two ontologies are delicately distinctive (and people having worked long enough in any of the two will be quick in pointing out salient differences), we believe that the results that follow can be "translated" with minimal effort to fit the specifics of the appropriate research paradigms. On the other hand, we do not claim or imply in any way that Haskell's type classes are a replacement to the well established formalisms used in the foundation of mathematics. If we did this in Agda or Coq, we could formulate rigorous proofs of the properties. However, formulating the rigorous proofs is a significant effort, and at this point we have opted for using Haskell and QuickCheck [Claessen and Hughes 2002] — this gives a lightweight approach that is a useful midpoint between an informal development and a machine-checked proof.

Within our analogy, Haskell's *type classes* can be seen as corresponding to *axiom systems* that describe properties and operations on multiple theories generically i.e. in terms of their action on objects of *parametric* types. Haskell's *instances* correspond to *concrete models* of such axiomatizations by providing implementations of primitive operations and by refining and possibly overriding derived operations. Mappings between different *instances* correspond, within this analogy, to *interpretations* seen as morphisms between theories [Kaye and Wong 2007].

We start by defining a type class that abstracts away the operations on the `OIs` 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 primitive operations

The class `Polymath` assumes only a theory of structural 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,i,r :: n → n
```

We have chosen single letter names `e,o_,o,i,r` for the abstract operations corresponding respectively to `empty, isO, withO, withI, reduce` to help with a more "algebraic" view as some definitions will use fairly complex compositions of these operations. As a minimal definition, the class also provides generic implementations of the following recognizer predicates:

```
  e_, i_ :: n → Bool
```

```
e_ x = x == e
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 shall see, some instances will override them. We first define a constructor 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, p :: n → n

s x | e_ x = u
s x | o_ x = i (r x)
s x | i_ x = o (s (r x))

p x | u_ x = e
p x | o_ x = i (p (r x))
p x | i_ x = o (r x)
```

We will sketch here an approach to formally validate such definitions. The following holds:

PROPOSITION 1. $\forall\, x\ p\ (s\ x) = x$ *and* $\forall\, x\ x \neq e \Rightarrow s\ (p\ x) = x$.

The inductive proof of this property uses the definitions directly. Clearly, p (s e) = p u = e (using the first pattern in s and p). Assume p (s x) = x. Then p (s (o x)) = p (i x) = o x. Also p (s (i x)) = p (o (s x)) = i (p (s x)) = i x. This proves $\forall\, x\ p(s\ x) = x$. The induction on the second part of the proposition is similar[1].

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. The function view allows importing a wrapped object of a different Polymath instance, generically.

```
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))
```

A generator for the infinite stream starting with k is obtained using s as follows:

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

We will now state a set of assertions that instances of the class Polymath must satisfy, testable using QuickCheck [Claessen and Hughes 2002].

QuickCheck tests a proposition on random instances of its arguments, returning ok if it passes the tests, or a counterexample if it fails one. The assertion prop_poly checks that r is the left inverse to o and i, that e_, o_, i_ are true for values built with their respective constructors, and that exactly one of e_, o_, and i_ holds for any value of an instance of Polymath; it follows that e, o, and i form a free algebra. The function withType returns its first argument with the type of its second, and here is used to ensure that the e has the same type as the random value x passed to prop_poly.

[1] Note that this proof is "generic" in the sense that it works with definitions provided by the type class. However, if an instance overrides some of these definitions, a separate proof is needed that the same properties hold.

```
prop_poly :: Polymath a ⇒ a → Bool
prop_poly x  = r (o x) == x && r (i x) == x &&
  e_ (e `withType` x) && o_ (o x) && i_ (i x) &&
  exactlyOneOf [e_ x, o_ x, i_ x]

exactlyOneOf xs = length [ x | x ← xs, x ] == 1

withType :: a → a → a
x `withType` y = x
```

The assertion prop_iso states (generically) that the conversion from instance a to instance b is the inverse of the conversion from b to a.

```
prop_iso :: (Polymath a, Polymath b) ⇒ a → b → Bool
prop_iso x y = view (view x `withType` y) == x &&
  view (view y `withType` x) == y
```

Random instances of a polymath instance are generated by the following:

```
arbitraryPolymath :: Polymath a ⇒ Gen a
arbitraryPolymath = oneof [return e,
  fmap o arbitraryPolymath,
  fmap i arbitraryPolymath]
```

QuickCheck defined a type Gen a for a generator that returns randomly selected values of type a. Here, we define a generator that returns instances of Polymath. These are defined by selecting values built with constructors e, o, or i, in the last two cases applying the constructor to another randomly generated value. (Functions oneof and fmap are defined in the QuickCheck library).

3.2 An intuition: the mapping to nonnegative Integers

And for the reader curious by now about how this maps to standard computer arithmetic here is an instance built around the (arbitrary length) Integer type (restricted to nonnegative numbers):

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

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

After adding

```
instance Arbitrary N where
  arbitrary = arbitraryPolymath
```

we can run QuickCheck

```
quickCheck (prop_poly :: N → Bool)
OK, passed 100 tests.
```

3.3 Peano arithmetic

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

One can verify by structural induction on the arguments of `s` and `p` that:

PROPOSITION 2. *Peano's axioms hold with the polymorphic definition of the successor function provided in class Polymath by* `s`, `p` *i.e.* `s` *works the same way as applying the constructor* `Succ` *and* `p` *works the same way as removing it from an expression of type* `Peano`.

After defining

```
instance Arbitrary Peano where
  arbitrary = arbitraryPolymath

prop_Peano :: Peano → Bool
prop_Peano x  =  Succ x == s x &&
  (x == Zero || Succ (p x) == x)
```

one can validate this instance with

```
test_Peano = quickCheck prop_Peano >>
             quickCheck (prop_poly :: Peano → Bool)
```

3.4 A "canonical instance" - the free monoid $\{0,1\}^*$

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

```
instance Polymath OIs where
  e=empty
  o=withO
  o_=isO
  i=withI
  r=reduce
```

One can verify by structural induction on the arguments of `s` and `p` that:

PROPOSITION 3. *Peano's axioms hold for the definition of the successor and predecessor functions* `s` *and* `p` *derived from the class Polymath for the instance* `OIs`.

After adding

```
instance Arbitrary OIs where
  arbitrary = arbitraryPolymath
```

one can also test that

```
*SharedAxioms> quickCheck (prop_poly :: OIs → Bool)
OK, passed 100 tests.
*SharedAxioms> quickCheck
  (prop_iso :: OIs → Peano → Bool)
OK, passed 100 tests.
*SharedAxioms> quickCheck (prop_iso :: OIs → N → Bool)
OK, passed 100 tests.
```

So far we have seen that our instances implement syntactic variations of natural numbers equivalent to Peano's axioms. We 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 implicitly 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 by adding finite unions of existing sets at each stage. We represent them as a rooted ordered tree datatype `S`

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

where the "empty leaf" `S []` denotes the empty set. To ensure that each set has a unique representation, the type `S` is assumed subject to the constraint that *at every level, elements are in ascending order*. The Ackermann mapping gives an isomorphism between hereditarily finite sets and the integers [Ackermann 1937]:

DEFINITION 2 (Ackermann mapping). *Objects of type* `S` *are such that the function* `f` *associating a natural number to a hereditarily finite set* x *of type* `S`*, given by the formula*

$$f(x) = \sum_{a \in x} 2^{f(a)}$$

is a bijection.

By using the Ackermann mapping

```
ackermann :: S → Integer
ackermann (S xs) = sum (map ((2^) . ackermann) xs)
```

one can borrow the ordering relation on natural numbers and define

```
isAscending :: [S] → Bool
isAscending []  =  True
isAscending [x]  =  ackermann x ≥0
isAscending (x:y:zs)  =  0 ≤ ackermann x &&
  ackermann x < ackermann y && isAscending (y:zs)
```

At this point one can validate that an object of type `S` represents canonically a hereditarily finite set by stating that its elements are in ascending order at each level:

```
canonicalSet :: S→Bool
canonicalSet (S []) = True
canonicalSet (S xs) =
  (all canonicalSet xs) && (isAscending xs)
```

We will now show that hereditarily finite sets *can do arithmetic* as instances of the class `Polymath`, by implementing a successor function `s` and a predecessor function `p`. 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 bit-stack (or as a "left shift" on a bitstring) is implemented by applying `s` to each branch of the tree. The key intuition here is that a "left shift" (at a given level) is the same thing as incrementing by `1` each exponent in the Ackermann mapping.

We implement `s`, `p` and `r` as follows:

```
  s (S xs) = S (lift (S []) xs)

  p (S (x:xs)) = S (lower x xs)

  r (S xs) = S (map p (adjust (p (S xs))))

lift k (x:xs) | k == x = lift (s x) xs
lift k xs = k:xs
```

```
lower (S []) xs = xs
lower k xs = lower (p) (p k:xs)

adjust (S (S []:xs)) = xs
adjust (S xs) = xs
```

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 `lift` and `lower` are used to delegate work between successive levels of the tree defining a hereditarily finite set. Note that the successor and predecessor operations `s` and `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. Note also that applying `s` and `p` to multiple branches of a tree of type `S` are independent operations, subject to *parallel execution*.

After adding

```
instance Arbitrary S where
  arbitrary = arbitraryPolymath
```

and defining the assertions

```
prop_lift (S []) = True
prop_lift (S (x:xs)) =
  lift e ((e `upto` x) ++ xs) == x : xs

prop_lower (S []) = True
prop_lower (S (x:xs)) =
  lower x xs == (e `upto` x) ++ xs

x `upto` y | x == y  = []
x `upto` y = x : (s x `upto` y)
```

where, if $0 \leq x \leq y$, then `x `upto` y` is a generator for the equivalent of the "interval" `[x..y-1]` expressed in terms of successive applications of `s`, one can validate this instance with

```
test_S = quickCheck canonicalSet >>
         quickCheck prop_lift >>
         quickCheck prop_lower >>
         quickCheck (prop_poly :: S → Bool) >>
         quickCheck (prop_iso :: OIs → S → Bool)
```

as follows

```
*SharedAxioms> quickCheck test_S
OK, passed 100 tests.
....
```

It can be proven by structural induction [2] on the arguments of `s` and `p` that:

PROPOSITION 4. *Hereditarily finite sets, as represented by the data type* `S`*, implement the same successor and predecessor operation as the instance* `Peano`*.*

Moreover, the implementation of `s,p` and `r` is compatible with the *Ackermann interpretation of Peano arithmetic in terms of the theory of hereditarily finite sets*. For instance, one can observe on a few examples that `s` and `p` operations on objects of type `S` match their natural number counterparts through Ackermann's bijection:

```
*SharedAxioms> ackermann (s (s (S [])))
2
*SharedAxioms> ackermann (s (s (s (S []))))
3
*SharedAxioms> ackermann (p (s (s (s (S [])))))
2
```

[2] When working on objects of type `S` this can be seen as an application of ϵ-induction i.e. one has to show that a given property holds for `S []` and that if it holds for all members of the list `S xs`, it also holds for `S xs`.

We can also state that the results of our operations are canonical representations of sets:

PROPOSITION 5. *If the predicate* `canonicalSet` *holds for objects of type* `S` *that are the arguments of the operations* `s, p` *and* `r`*, then it also holds for their results (when defined).*

Let us refocus at this point on what's unusual with instance `S` of the class `Polymath`: we have shown 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 elaborate more on this after defining a total order on our Polymath type.

We now provide an instance defined in terms of a more efficient hereditarily finite construct.

5. Computing with Hereditarily Finite Functions

Hereditarily finite functions, described in detail in [Tarau 2009b,c], 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 rooted ordered tree datatype denoted `F` here. In contrast with the type `S`, no constraints are assumed about `F` (i.e. `F` is a *free* algebraic type).

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

Note that for objects of type `F` order is significant, and identical elements can occur at any level.

Hereditarily finite functions can also be seen as compressed encodings of hereditarily finite sets, where, w.r.t the Ackermann mapping, at each level, increments to exponents replace the exponents.

More precisely, while objects of type `S` are built recursively from a natural number (through the inverse of the Ackermann mapping) by associating to a number the exponents in its binary representation, objects of type `F` are built by computing the *increments* between such exponents.

For instance, the first level expansion for `n = 100` when generating type `S` is `[2,5,6]`, corresponding to $100 = 2^2 + 2^5 + 2^6$, while for type `F` it is `[2,2,0]`.

The transformation between the two lists can be expressed as $\{2 \to 2,\ 5 \to 2 = 5-2-1,\ 6 \to 0 = 6-5-1\}$ and as the following code shows (assuming that the input to diffs is in ascending order) it is a bijection from sets to lists, i.e. given the definitions

```
diffs []  =  []
diffs (x:xs)  =  x : dffs x xs where
  dffs x []  =  []
  dffs x (y:ys)  =  y-x-1 : dffs y ys

sums []  =  []
sums (x:xs)  =  x : sms x xs where
  sms x []      =  []
  sms x (y:ys)  =  x+y+1 : sms (x+y+1) ys
```

the following holds:

PROPOSITION 6. *diffs . sums = id sums . diffs = id*

After defining

```
prop_diffs :: [Integer] → Bool
prop_diffs xs  =  sums (diffs xs) == xs &&
  diffs (sums xs) == xs
```

we can verify

```
*SharedAxioms> quickCheck prop_diffs
OK, passed 100 tests.
```

Given this bijection relating sets to sequences, we define an instance of the class `Polymath` based on `F` in a way similar to its counterpart on data type `S`:

```
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 `s, p` and `r` 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.

A key difference with the operations on type `S`, is that, for instance, the function `s` does not have the exponents of the Ackermann mapping at hand, and it has to compute them on the fly before incrementing them (by using `hinc`). The same applies to the function `p` and the auxiliary function `hdec` that "decrements" such exponents computed on the fly to achieve the equivalent of a "right shift" operation.

```
  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)

  r (F (x : xs)) | e_ x = F xs
  r (F (k : xs)) = F (hpad (p k) (hnext xs)) where
    hpad x xs | e_ x = xs
    hpad x xs = e : hpad (p x) xs

    hnext [] = []
    hnext (k : xs) = s 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 significantly smaller than `S`-trees, various operations perform significantly faster, as in this representation only "increments" or "decrements" from one subtree to the next are computed by the functions `hinc` and `hdec`.

It can be proven by structural induction on the arguments of `s` and `p` that:

PROPOSITION 7. *Hereditarily finite functions as represented by the data type* `F` *implement the same successor and predecessor operation as the instance* `Peano`.

After adding

```
instance Arbitrary F where
  arbitrary = arbitraryPolymath
```

one can test that various assertions hold for this instance as well.

6. Ordering

Efficient comparison uses the fact that with our representation only sequences of equal lengths can be equal. We start by comparing lengths:

```
class (Polymath n) ⇒ PolyOrdering n where
  lcmp :: n → n → Ordering
  lcmp x y | e_ x && e_ y = EQ
  lcmp x y | e_ x && not(e_ y) = LT
  lcmp x y | not(e_ x) && e_ y = GT
  lcmp x y = lcmp (r x) (r y)
```

Comparison can now proceed by case analysis, the interesting case being when lengths are equal (function `samelen_cmp`):

```
  cmp :: n → n → Ordering
  cmp x y = ecmp (lcmp x y) x y where
     ecmp EQ x y = samelen_cmp x y
     ecmp b _ _ = b

  samelen_cmp :: n → n → Ordering

  samelen_cmp x y | e_ x && e_ y = EQ
  samelen_cmp x y | e_ x && not(e_ y) = LT
  samelen_cmp x y | not(e_ x) && e_ y = GT
  samelen_cmp x y | o_ x && o_ y =
    samelen_cmp (r x) (r y)
  samelen_cmp x y | i_ x && i_ y =
    samelen_cmp (r x) (r y)
  samelen_cmp x y | o_ x && i_ y =
    downeq (samelen_cmp (r x) (r y)) where
      downeq EQ = LT
      downeq b = b
  samelen_cmp x y | i_ x && o_ y =
    upeq (samelen_cmp (r x) (r y)) where
      upeq EQ = GT
      upeq b = b
```

Finally, boolean comparison operators are defined as follows:

```
  lt, gt, eq :: n → n → Bool

  lt x y = LT == cmp x y
  gt x y = GT == cmp x y
  eq x y = EQ == cmp x y
```

We are now ready for a sorting operation, derived from Haskell's parametric `sortBy` borrowed from module `Data.List`. We define our sorting function `nsort` as follows:

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

7. Polymath Combinators and Galois Connections

Galois connections [Erné et al. 1993] describe weaker than isomorphic relations between partial orders. As such they have been used in fields like *abstract interpretation* to infer propagation of program properties. We will use them here to highlight some abstract properties induced by the interaction of the Polymath operators with the order relations. We also study Galois connections induced by representation size measures like bitlength.

DEFINITION 3. *Let* $(A, \leq)$ *and* $(B, \leq)$ *be two partially ordered sets. A monotone Galois connection is a pair of monotone functions* $f : A \to B$ *and* $g : B \to A$ *such that* $\forall a \in A, \forall b \in B, f(a) \leq b$ *if and only if* $a \leq g(b)$.

DEFINITION 4. *Let* $(A, \leq)$ *and* $(B, \leq)$ *be two partially ordered sets. An antitone Galois connection is a pair of antitone functions* $f : A \to B$ *and* $g : B \to A$ *such that* $\forall a \in A, \forall b \in B, b \leq f(a)$ *if and only if* $a \leq g(b)$.

7.1 Galois Connections induced by Polymath primitives

PROPOSITION 8. *o, i, r are monotone. Also, o and r are, respectively, the lower and higher adjuncts of a (monotone) Galois connection i.e.*

$$\forall a\ \forall b\ a \leq b \Rightarrow o\ a \leq o\ b \tag{1}$$

$$\forall a\ \forall b\ a \leq b \Rightarrow i\ a \leq i\ b \tag{2}$$

$$\forall a\ \forall b\ a \leq b \Rightarrow r\ a \leq r\ b \tag{3}$$

$$\forall a\ \forall b\ o\ a \leq b \Leftrightarrow a \leq r\ b \tag{4}$$

Moreover, o and r form a Galois embedding on every instance from which e [3] *is excluded, i.e. o, i are injective and r is surjective on each such instance.*

It follows that $o \circ r$ is deflationary i.e.

$$\forall a\ o\ (r\ a)\ \leq a \tag{5}$$

and idempotent, i.e.

$$(o \circ r) \circ (o \circ r) \equiv o \circ r \tag{6}$$

and

$$o \circ r \circ o \equiv o \tag{7}$$

$$r \circ o \circ r \equiv r \tag{8}$$

Note also that as

$$r \circ o \equiv r \circ i \equiv \lambda x.x \tag{9}$$

$r \circ o$ is trivially inflationary and idempotent.

PROPOSITION 9. *The following identities hold for all class level definitions and overrides of the combinators involved.*

$$s \circ o \equiv i \equiv p \circ o \circ s \tag{10}$$

$$o \circ s \equiv s \circ i \equiv s \circ s \circ o \tag{11}$$

$$o \equiv p \circ i \equiv s \circ i \circ p \tag{12}$$

$$p \circ s \equiv \lambda x.x \tag{13}$$

$$\forall a((a \neq e) \Rightarrow s(p\ a) \equiv a) \tag{14}$$

7.2 Galois-connections induced by bitlength

Bitlength is computed by applying one successor `s` for each "bit" i.e. each application of `o` or `i`.

```
bitlen :: n → n
bitlen x | e_ x = e
bitlen x = s (bitlen (r x))
```

Let `oS` be the function that iterates `o` operations `l` times, and let `iS` be the function that iterates `i` operations `l` times.

```
oS :: n → n
oS l | e_ l = e
oS l = o (oS (p l))

iS :: n → n
iS l | e_ l = e
iS l = i (iS (p l))
```

The following holds:

[3] for which r is not defined

PROPOSITION 10. `bitlen` *and* `oS` *form an antitone Galois connection and* `bitlen` *and* `iS` *form a monotone Galois connection, i.e.*

$$\forall a\ \forall b\ a \leq b \Rightarrow bitlen\ a \leq bitlen\ b \tag{15}$$

$$\forall a\ \forall b\ a \leq b \Rightarrow oS\ a \leq oS\ b \tag{16}$$

$$\forall a\ \forall b\ a \leq b \Rightarrow iS\ a \leq iS\ b \tag{17}$$

$$\forall a\ \forall b\ bitlen\ a \leq b \Leftrightarrow a \geq oS\ b \tag{18}$$

$$\forall a\ \forall b\ bitlen\ a \leq b \Leftrightarrow a \leq iS\ b \tag{19}$$

One can add tests that f and g act as a (monotone or antitone) Galois connections on a given value after defining

```
isGalois :: (n → n) → (n → n) → n → n → Bool
isGalois f g x y = v1 == v2 where
  v1=le (f x) y
  v2=le x (g y)

isGaloisAnti :: (n → n) → (n → n) → n → n → Bool
isGaloisAnti f g x y = v1 == v2 where
  v1=le (f x) y
  v2=le (g y) x

le :: n → n → Bool
le a b = not (lt b a)
```

After adding the instance declarations

```
instance PolyOrdering N
instance PolyOrdering Peano
instance PolyOrdering OIs
instance PolyOrdering S
instance PolyOrdering F
```

one can verify Galois connection properties as follows:

```
*SharedAxioms> isGalois o r 2009 2010
True
*SharedAxioms> isGalois o r (S []) (S [S []])
True
*SharedAxioms> isGalois o r (N 1) (N 1)
True
*SharedAxioms> isGalois bitlen iS 22 4
True
*SharedAxioms> isGaloisAnti bitlen oS 22 4
True
```

One can also test at this point that various arithmetic operations and ordering relations work as expected on several instances.

```
*SharedAxioms> lt (N 2009) (N 2010)
True
*SharedAxioms> nsort [N 3,N 2,N 1,N 2]
[N 1,N 2,N 2,N 3]
*SharedAxioms> lt (S []) (S [S [],S []])
True
```

The last operation shows now that we have a *total order* on hereditarily finite sets without recourse 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 (constructively) a recent result described in [Pettigrew] where a lexicographic ordering is used to simplify the proof of bi-interpretability of [Kaye and Wong 2007].

8. Adding Arithmetic Operations

Our next refinement adds key arithmetic operations in the form of a type class extending `Polymath`.

```
class (PolyOrdering n) ⇒ PolyArithmetic 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)))
```

We next define multiplication operations.

```
  m :: n → n → n
  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)))

  double,half :: n → n

  double = p . o
  half = r . s
```

After defining the instances

```
instance PolyArithmetic N
instance PolyArithmetic Peano
instance PolyArithmetic OIs
instance PolyArithmetic S
instance PolyArithmetic F
```

operations can be tested under various representations

```
*SharedAxioms> a (N 32) (N 10)
N 42
*SharedAxioms> view (N 6) :: F
F [F [F []],F []]
*SharedAxioms> m it it
F [F [F [F []]],F [F [F []]]]
*SharedAxioms> view it :: N
N 36
```

9. Adding a Pairing Function

A *pairing* function is an bijection $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Its inverse is called *unpairing*.

Our next extension provides an efficiently computable pairing function. This classic pairing function, denoted `pair`, together with its left and right unpairing companions `first` and `second` is attributed to Pepis [Pepis 1938] and has been used in work on recursion theory, decidability and Hilbert's Tenth Problem [Robinson 1950]. The function `pair` combines two numbers reversibly by multiplying a power of 2 derived from the first and an odd number derived from the second:

$$f(x, y) = 2^x(2y + 1) - 1 \tag{20}$$

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 `first` and `second`.

```
class (PolyArithmetic n) ⇒ PolyPair n where
  pair :: n → n → n
  pair x y = h x (double y) where
    h x ys | e_ x = ys
    h x ys = o  (h (p x) ys)

  first, second :: n → n

  first z | o_ z = s (first (half z))
  first _ = e

  second z = half (t z) where
    t xs | o_ xs = t (half xs)
    t xs  = xs
```

After adding the instances

```
instance PolyPair N
instance PolyPair Peano
instance PolyPair OIs
instance PolyPair S
instance PolyPair F
```

one can test that various instances work as expected with

```
prop_pair x y z =
 first (pair x y) == x &&
 second (pair x y) == y &&
 pair (first z) (second z) == z
```

10. Morphing between List, Multiset and Set views

We now lay ground for reflecting *collection types* like sets, multisets and lists of natural numbers by first showing that one can freely move between them through simple bijective transformations. 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,c] for more details on such mappings, but the computations involved are surprisingly straightforward.

```
class (PolyPair n) ⇒ PolyCollection 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 d 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 PolyCollection N
instance PolyCollection Peano
instance PolyCollection OIs
instance PolyCollection S
instance PolyCollection F
```

one can see how increments between successive terms relate lists, multisets and then sets:

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

11. Deriving bijective mappings between lists, multisets, sets and natural numbers

We first derive a list representation based on our pairing function. Next, set and multiset representations are derived using their mappings to lists.

```
class (PolyCollection n) ⇒ PolyCons n where
  hd, tl ::  n → n

  hd = first . p
  tl = second . p

  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
```

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

The last example also shows that `as_set_nat` maps the tree representing a hereditarily finite set to the forest growing out if its root. The deeper reason for this is that *our pairing function induces the Ackermann interpretation as the bijection between* $\mathbb{N}$ *and* ZF^*. This follows from the fact that the `hd` operation induced by `first` computes at each step the distance to the next element at bit i that is 1, corresponding to 2^i in Ackermann's mapping. A somewhat unusual application is deriving an efficient computation for the power function `pow` by observing that, under the natural number interpretation, `hd x` is the largest exponent of 2 dividing x and `tl x` is half of the quotient of x by `hd x`.

```
  pow :: n → n → n
  pow _ y | e_ y = u
  pow x _ | e_ x = e
  pow x y = z where
    b=pow2n x (hd y)
    b'=pow b (tl y)
    z=m b (m b' b')

    pow2n x n | e_ n = x
    pow2n x n = pow2n (m x x) (p n)
```

After adding the instances

```
instance PolyCons N
instance PolyCons Peano
instance PolyCons OIs
instance PolyCons S
instance PolyCons F
```

everything works as as expected:

```
pow (view (N 123) :: F) (view (N 123) :: F)
F [F [], F [],..., [F []]],F [],F [],F []]]
*SharedAxioms> view it:: N
N 11437436...583828037071007667402208392 67
*SharedAxioms> 123 ^ 123
11437436...583828037071007667402208392 67
```

We are now ready to add a "reflected" set theory layer, i.e. we use Haskell lists as intermediate representations, while keeping in mind that they can be eliminated with deforestation transformations.

12. Deriving Set Operations

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

```
class (PolyCons n) ⇒ PolySet n where
  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,setUnion,setDifference :: n → n → n

  setIntersection = setOp2 intersect
  setUnion = setOp2 union
  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 n-th *von Neumann ordinal* n is the set $\{0, 1, \ldots, n-1\}$) and it 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 PolySet N
instance PolySet Peano
instance PolySet OIs
instance PolySet S
instance PolySet F
```

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

```
*SharedAxioms> view (N 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]
```

13. Deriving List Operations

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

```
class (PolySet n) ⇒ PolyList 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 PolyList N
instance PolyList Peano
instance PolyList OIs
instance PolyList S
instance PolyList F
```

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

```
*SharedAxioms> view (N 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 []]
```

14. Directed Graphs, DAGs and Undirected Graphs

We show that building more complex data types like digraphs, unordered graphs, DAGs follows naturally. The mechanism for deriving them is surprisingly uniform.

14.1 Deriving edge types from a pairing function

We use 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 graphs and "upward pointing" edges for canonically represented DAGs. They will be used later to derive transformations to/from various graph types. We will also add one more layer to our Polymath classes to allow sharing transformations to/from graphs among various implementations.

```
class (PolyList n) ⇒ PolyGraph n where

  ordUnpair, unordUnpair, upwardUnpair :: n → (n,n)
  ordPair, unordPair, upwardPair :: (n,n) → n

  ordUnpair z = (first z,second z)

  ordPair (x,y) = pair x y

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

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

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

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

14.2 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 `ordPair`.

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

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

14.3 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
```

14.4 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 using `upwardPair` and `upwardUnpair`:

```
  as_set_dag :: [(n,n)] → [n]
```

```
as_set_dag  = map upwardPair

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

We conclude this by updating our instance definitions

```
instance PolyGraph N
instance PolyGraph Peano
instance PolyGraph OIs
instance PolyGraph S
instance PolyGraph F
```

Encoding graph types as natural numbers can provide succinct representations and perfect hash-keys for graph indexing.

15. Computing with Binary Trees

We show here how our shared axiomatization framework can be extended with ordered rooted binary trees with empty leaves, which can also be seen as describing the type language of an important ancestor of modern type systems: Gödel's System **T**.

DEFINITION 5. *In Gödel's System* **T** *[Gödel 1958] a type is either the basic type t_0 or $t_1 \to t_2$ where t_1 and t_2 are types.*

The basic type t_0 is usually interpreted as the type of natural numbers. Clearly we recognize here a member of the *Catalan family* isomorphic with rooted ordered binary trees. We show now that Polymath operations can be emulated directly with such trees, by using a single constant T as the leaf type, (seen as representing 0 or an "empty leaf"). Consequently, this shows that our shared axiomatization framework (already providing views as natural numbers, sets or sequences) also covers the type language of System **T**.

First, we observe that, guided by the known isomorphism between rooted ordered trees and rooted ordered binary trees (that we will use to represent types) [4] we can bring with a *functor* defined from hereditarily finite sequences to binary trees the definitions of `s`, `p` and `r` into corresponding definitions in the language of system **T** types.

First, we define the ordered binary tree with empty leaves data type for **T** objects:

```
infixr 5 :→
data T = T|T :→ T deriving (Eq, Read, Show)
```

As in the case of hereditarily finite sets and functions we start by defining the first 4 primitive operations:

```
instance Polymath T where
  e = T

  o_ (T :→ x ) = True
  o_ _       = False

  o x = T :→ x
  i x = s (T :→ x)
```

To be able to inherit various generic operations, we need also to implement the 5-th primitive operation `r` along the lines of its equivalent for the datatype `F` for hereditarily finite functions:

```
  r (T :→ y) = y
  r (x :→ y) = hpad (p x) y where
    hpad T y = hnext y
    hpad x y = T :→ hpad (p x) y
    hnext T = T
    hnext (x :→ y) = s x :→ y
```

[4] That manifests itself in languages like Prolog or LISP as the dual view of lists as a representation of sequences or ordered binary CONS-cell trees.

Note that the successor and predecessor functions `s` and `p` are used in the definition of `i` and `r`. We implement them as overrides for `s` and `p`

```
  s T = T :→ T
  s (T :→ y) = s x :→ y' where x :→ y' = s y
  s (x :→ y) = T :→ (p x :→ y)

  p (T :→ T) = T
  p (T :→ (x :→ y)) = s x :→ y
  p (x :→ y) = T :→ p (p x :→ y)
```

Note that, as in the case of similar operations in sections 4 and 5, one could have simply defined two auxiliary functions `s'` and `p'` to be used in implementing `i, r` and then reuse the generic definition of `s` and `p` given in section 3, to ensure that the instance `T` matches formally the concept of *interpretation* of our axioms expressed in terms of the 5 primitive operations `e, o, o_, i, r`.

After observing that $z = x :\to y$ if and only if (under a natural number interpretation) $z = 2^x(2y+1)$, it can be proven by structural induction on the arguments of `s` and `p` that:

PROPOSITION 11. *Rooted ordered binary trees, as represented by the data type* `T`*, implement the same successor and predecessor operation as the instance* `Peano`*.*

We are ready now to turn objects of type `T` into everything else (natural numbers, finite sets, finite functions, etc.)

```
instance PolyOrdering T
instance PolyCollection T
instance PolyArithmetic T
instance PolyPair T
instance PolyCons T
instance PolySet T
instance PolyList T
instance PolyGraph T
instance Arbitrary T where
  arbitrary = arbitraryPolymath
```

Arithmetic and set computations, operating directly on them, are now derived automatically:

```
*SharedAxioms> let two = s (s (T))
*SharedAxioms> let three = s two
*SharedAxioms> two
(T :→ T) :→ T
*SharedAxioms> three
T :→ (T :→ T)
*SharedAxioms> m two three
(T :→ T) :→ (T :→ T)
*SharedAxioms> powset three
T :→ (T :→ (T :→ (T :→ T)))
*SharedAxioms> view it :: S
S [S [],S [S []],S [S [S []]],S [S [],S [S []]]]
*SharedAxioms> view three :: S
S [S [],S [S []]]
```

While we have only used here the "binary tree with empty leaves" view of `System T` types, it follows that one can replace Gödel's original assumption that the base type is $\mathbb{N}$ with the weaker assumption that it is the empty type `T`, as natural numbers (and their operations in accordance with Peano's axioms) can be represented as types.

16. Related Work

Natural number encodings of hereditarily finite sets and definitions of arithmetic operations in terms of them have triggered the interest of researchers in fields like Axiomatic Set Theory and Foundations of Logic [Takahashi 1976, Kaye and Wong 2007, Kirby 2007, Pettigrew]. However, it is a novel contribution of this paper to

show that such operations can be implemented constructively by working directly on symbolic representations of hereditarily finite sets, functions and binary trees, and that all operations work in time and space proportional to the size of the representations. Also, in contrast with [Kaye and Wong 2007] we do not use von Neumann ordinals for the reverse bijection from sets to natural numbers given that we are able to do arithmetics directly on sets.

In contrast with [Pettigrew] where such a bijection is defined implicitly through the existence of a lexicographic order allowing the enumeration of sets of a given size, we define the bijection as an actual function computable both ways in time proportional to the size of the representation and define a compatible order relation computable in time proportional to the size of the operands.

While both [Kaye and Wong 2007] and [Pettigrew] work exclusively with sets and natural numbers, our polymorphic representation covers at the same time, hereditarily finite functions, bijective base-2 numbers and rooted ordered binary trees.

The importance of extending arithmetic operations to sets is recognized in [Kirby 2007] where addition and multiplication operations are defined. However these operations diverge significantly from their natural number counterparts - for instance addition is non-commutative. In contrast with [Kirby 2007] our implementation can be seen as an *isofunctor* that preserves the usual commutative semigroup structure of $\mathbb{N}$ with addition and multiplication.

The encodings of hereditarily finite sets and sequences described in this paper originate in [Tarau 2009c,b,d,a]. The key difference is that while in our previous work we use pairs of bijections encapsulated as combinators to define various isomorphisms directly, here we let the type class mechanism do most of the work by instantiating generic operations. Moreover, here we provide actual algorithms for arithmetic operations, ordering etc. while in our previous work the existence of such algorithms was only implied "non-constructively".

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [Kiselyov et al. 2008]. Their approach is similar as far as a symbolic representation is used. The key difference with this paper is that our operations work on tree structures, and as such, they are not based on previously known algorithms. Pairing functions have been used in work on decision problems as early as [Pepis 1938, Robinson 1950]. A typical modern use in the foundations of mathematics is [Cégielski and Richard 2001].

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

Binary number-based axiomatizations of natural number arithmetic 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 arithmetic in the libraries of the `Coq` [The Coq development team 2004] proof assistant has been particularly enlightening to us. On the other hand we have not found in the literature any such axiomatizations in terms of hereditarily finite sets, hereditarily finite functions or binary trees, as described in this paper.

17. Conclusion

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

Besides introducing algorithms expressing arithmetic computations in terms of "symbolic structures" like hereditarily finite sets and hereditarily finite functions with asymptotic complexity similar to standard binary arithmetic, our framework unifies fundamental mathematical concepts in a directly executable form.

Acknowledgment

We thank Philip Wadler for his careful reading and salient suggestions on clarifying the content and improving the structure of this paper as well as for his help with ensuring the correctness and for validating the algorithms in sections 4 and 5 using QuickCheck. We thank the anonymous reviewers of PPDP 2010 for their comments, suggestions and constructive criticism.

References

Wilhelm Friedrich Ackermann. Die Widerspruchsfreiheit der allgemeinen Mengenlhere. *Mathematische Annalen*, (114):305–315, 1937.

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.

Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

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

M. Erné, J. Koslowski, A. Melton, and G. E. Strecker. A Primer on Galois Connections. *New York Academy Sciences Annals*, 704:103–125, December 1993.

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

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.

Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In Jacques Garrigue and Manuel V. Hermenegildo, editors, *FLOPS*, volume 4989 of *Lecture Notes in Computer Science*, pages 64–80. Springer, 2008. ISBN 978-3-540-78968-0.

J.P. Mayberry. *The Foundations of Mathematics in the Theory of Sets*. Cambridge University Press, 2000.

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

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.

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, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , pages 170–185, Grand Bend, Canada, July 2009b. Springer, LNAI 5625.

Paul Tarau. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*, pages 171–182, Coimbra, Portugal, September 2009c. ACM.

Paul Tarau. Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In *Proceedings of ACM SAC'09*, pages 1898–1903, Honolulu, Hawaii, March 2009d. ACM.

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

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.