Declarative Modeling of Finite Mathematics

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PPDP'10, July 2010

Motivation

(Some) competing foundations for Finite Mathematics (going back to Kronecker vs. Cantor)

- Natural Numbers: Peano's axioms (and equivalent theories of binary numbers)
- (Hereditarily) Finite Sets: ZF - Infinity + ε induction
- Types: - Gödel's System **T** $\rightarrow$ Martin Löf Type Theory$\rightarrow$ System **F** $\rightarrow$ Calculus of Constructions $\rightarrow$ Coq

Why does this matter? Because decisions on Foundations of Finite Mathematics entail decisions on Foundations of Computer Science!

- Can we provide a unified formalism covering them all?
- Can this formalism be strongly “constructive"?
- Can we make it executable?
- Can we make it efficiently executable?

Outline

- axiomatizations of various formal systems are traditionally expressed in classic or intuitionistic predicate logic
- we use an equivalent formalism: λ-calculus + type theory as provided by Haskell
- $\Rightarrow$ our "specifications" are *executable*
  - type classes are seen as (approximations of) *axiom systems*
  - instances of the type classes are seen as *interpretations*
- a hierarchy of type classes describes common computational capabilities shared by
  - Peano natural numbers, bijective base-2 arithmetics,
  - hereditarily finite sets
  - System **T** types

Bijective base-2 natural numbers

Definition

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 bitstring representation of $n+1$.

Using a list notation for bitstrings this gives:
$0 = [], 1 = [0], 2 = [1], 3 = [0,0], 4 = [1,0], 5 = [0,1], 6 = [1,1]$

Arithmetic operations using bijective base-2 numbers

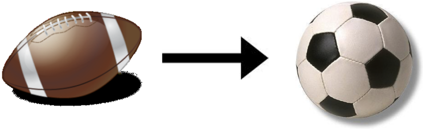

Figure: A thought on reinventing the wheel: some wheels are just better :-)

5 primitive BitStack / bijective base 2 operations

```
data OIs = E | O OIs | I OIs deriving (Eq, Show, Read)
empty = E                 -- op 1

withO xs = O xs           -- op 2

withI xs = I xs           -- op 3

reduce (O xs) = xs        -- op 4
reduce (I xs) = xs

isO (O _) = True          -- op 5
isO _     = False

isEmpty xs = xs == E                        -- derived op (from Eq)
isI x = not (isEmpty x) && not (isO x) -- derived op (from other)
```

Emulating Peano Arithmetic with Bijective Base-2 Arithmetic

```
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))
```

Proposition

BitStacks with peanoSucc are a model *of Peano's axioms.*

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

Abstracting away bijective base-2 operations as a type class: the 5 Primitive Polymath Operations

```
class (Eq n,Read n,Show n) ⇒ Polymath n where
  e :: n                      -- zero
  o_ :: n→ Bool               -- is it zero?
  o :: n→n                    -- x → 2x+1
  i :: n→n                    -- x → 2x+2
  r :: n→n                    -- 2x + {1,2} → x
```

Derived Polymath Operations

```
e_ :: n→Bool
e_ x = x==e

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

u :: n
u = o e

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

Successor s and predecessor p functions

```
s :: n→n

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

p :: n→n

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

Generic Inductive Proofs of Program Properties

Proposition

$\forall\ x\ p\ (s\ x) = x$ *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.

Likely to be easy to implement in Coq with the (new) type classes.

A polymorphic converter between Polymath instances

The function `view` allows converting between two different Polymath instances, 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))

views xs = map view xs
```

The reference instance: Peano arithmetic

We define an instance by implementing the primitive Polymath operations. This shows that Peano arithmetic provides an *interpretation* of the “axioms” provided by the class Polymath.

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

instance Polymath Peano where
  e = Zero

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

Instance Peano (continued)

```
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)
```

Representing Hereditarily Finite Sets (HFS)

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.

Definition (Ackermann mapping)

Objects of type `S` *are subject to the constraint 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.

A HFS and its successor

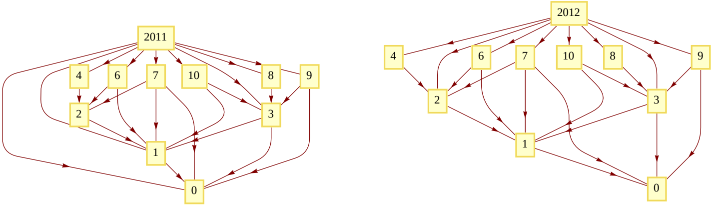

Figure: HFS: *2011* and *2012*

The operations s' and p' on type S (representing HFSs)

```
s' (S xs) = S (lift (S []) xs) where
  lift k (x:xs) | k == x = lift (s' x) xs
  lift k xs = k:xs

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

HFS as a Polymath instance

Hereditarily finite sets *can do arithmetic* as instances of the class `Polymath`. Here are the 5 primitives:

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

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

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

  i = s' . o

  r x | o_ x = S (map p' ys) where (S ys) = p' x
  r x = r (p' x)
```

s’ and p’ are implementations of s and p

Proposition

$s \equiv s'$ *and* $p \equiv p'$

```
*SharedAxioms> s (S [])
S [S []]
*SharedAxioms> s it
S [S [S []]]
*SharedAxioms> s it
S [S [],S [S []]]
*SharedAxioms> p' it
S [S [S []]]
*SharedAxioms> p' it
S [S []]
*SharedAxioms> p' it
S []
```

More examples of HFS operations

Let us verify that these operations mimic indeed their more common counterparts on type `Peano`.

```
*SharedAxioms> o (i (S []))
S [S [],S [S [S []]]]
*SharedAxioms> s it
S [S [S []],S [S [S []]]]
*SharedAxioms> view it :: Peano
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
*SharedAxioms> p it
Succ (Succ (Succ (Succ (Succ Zero))))
*SharedAxioms> view it :: S
S [S [],S [S [S []]]]
```

Polymorphic Ordering: shared by sets, Peano numbers etc.

```
class (Polymath n) ⇒ PolyOrdering n where
  lcmp :: n→n→Ordering   -- comparing "bit-lengths" first

  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)
```

if two sequences have different length, the longer is the larger one

```
  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
```

Arithmetic done efficiently i.e. O(size of the representation)

```
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

polyAdd :: n→n→n
polyAdd x y | e_ x = y
polyAdd x y | e_ y = x
polyAdd x y | o_ x && o_ y =    i (polyAdd (r x) (r y))
polyAdd x y | o_ x && i_ y = o (s (polyAdd (r x) (r y)))
polyAdd x y | i_ x && o_ y = o (s (polyAdd (r x) (r y)))
polyAdd x y | i_ x && i_ y = i (s (polyAdd (r x) (r y)))

--- polySubtract:: n→n→n ....
```

Galois Connections with i, o, r

Definition

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

Definition

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

Galois Connections induced by Polymath primitives

Proposition

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 is excluded, i.e. o, i are injective and r is surjective on each such instance.

Derived properties holding for all Polymath instances

$$r \circ o \equiv r \circ i \equiv \lambda x.x \tag{5}$$

$$s \circ o \equiv i \equiv p \circ o \circ s \tag{6}$$

$$o \circ s \equiv s \circ i \equiv s \circ s \circ o \tag{7}$$

$$o \equiv p \circ i \equiv s \circ i \circ p \tag{8}$$

$$p \circ s \equiv \lambda x.x \tag{9}$$

$$\forall x((x \neq e) \Rightarrow s(p\ x) \equiv x) \tag{10}$$

Set Operations

```
exp2 :: n→n -- power of 2
exp2 x | e_ x = u
exp2 x = double (exp2 (p x))

class (PolyCalc n) ⇒ PolySet n where
  as_set_nat :: n→[n]
  as_set_nat n = nat2exps n e where
    nat2exps n _ | e_ n = []
    nat2exps n x = if (i_ n) then xs else (x:xs) where
      xs=nat2exps (half n) (s x)

  as_nat_set :: [n]→n
  as_nat_set ns = foldr polyAdd e (map exp2 ns)
```

Examples

Given that natural numbers and hereditarily finite sets, when seen as instances of our generic axiomatization, are connected through Ackermann's bijections, one can shift from one side to the other at will:

```
*SharedAxioms> as_set_nat (s (s (s Zero)))
[Zero,Succ Zero]
*SharedAxioms> as_nat_set it
Succ (Succ (Succ Zero))
*SharedAxioms> as_set_nat (s (s (s (S []))))
[S [],S [S []]]
*SharedAxioms> as_nat_set it
S [S [],S [S []]]
```

Powerset, set membership, augmentSet

```
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)]]

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

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

Ordinals

- The n-th *von Neumann ordinal* is the set $\{0,1,\ldots,n-1\}$
- 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 = augmentSet (nthOrdinal (p n))
```

Note that as hereditarily finite sets and natural numbers are instances of the class `PolyOrdering`, an order preserving bijection can be defined between the two, which makes it unnecessary to resort to von Neumann ordinals to show bi-interpretability.

A practical outcome: representing some very large numbers

Note, as a more practical outcome, that one can now use arbitrary length integers as an efficient representation of hereditarily finite sets. Conversely, a computation like

```
*SharedAxioms> s (S [S [S [S [S [S [S [S [S [S []]]]]]]]]])
S [S [],S [S [S [S [S [S [S [S []]]]]]]]]
```

computing easily the successor of a tower of exponents of 2, in terms of hereditarily finite sets, would overflow any computer's memory when using a conventional integer representation.

Deriving Digraphs, DAGs, Undirected Graphs

Just a sketch - it is all in the paper...

- deriving ordered, unordered pairs - using a pairing function
- digraphs: as sets with elements split into ordered pairs
- undirected graphs: as sets with elements split into unordered pairs
- DAGs: as a simple arithmetic transformation of digraphs edges

Computing with Binary Trees representing System **T** types

Gödel System **T** types: a minimalist ancestor of modern type systems.

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

instance Polymath T where
  e = T

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

  o x = T :→ x
  i = s . o

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

Successor s and predecessor p with System **T** types

```
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)
```

An interesting consequence:

- no need to add natural numbers as a base type to System **T**, given that types can emulate them (actually, in an efficient way!)
- this holds for virtually all type systems - as System **T** is their minimal common ancestor ...

Examples for System **T** arithmetics

```
*SharedAxioms> view 2012 :: T
((T :→ T) :→ T) :→ (T :→ (T :→
 ((T :→ T) :→ (T :→ (T :→ (T :→ (T :→ T)))))))
*SharedAxioms> s it
T :→ ((T :→ T) :→ (T :→ (T :→
  ((T :→ T) :→ (T :→ (T :→ (T :→ (T :→ T))))))))
*SharedAxioms> view it :: N
2013
*SharedAxioms> s T
T :→ T
*SharedAxioms> s it
(T :→ T) :→ T
*SharedAxioms> s it
T :→ (T :→ T)
*SharedAxioms> s it
((T :→ T) :→ T) :→ T
```

Defining the System **T** Recursor

```
rec :: (T → T → T) → T → T → T

rec f T y = y
rec f x y = f (p x) (rec f (p x) y)

itr f t u = rec g t u where
  g _ y = f y

recAdd = itr s

recMul x y = itr f y T where
  f y = recAdd x y

recPow x y = itr f y (T :→ T) where
  f y = recMul x y
```

Conclusion

- In the form of a literate Haskell program, we have built "shared axiomatizations" of finite arithmetic and hereditarily finite sets using successive refinements of type classes.
- $\Rightarrow$ a well-ordering for hereditarily finite sets that maps them to ordinals directly, without using the von Neumann construction.
- $\Rightarrow$ we can do arithmetics without "numbers" by computing symbolically, with trees representing hereditarily finite sets, functions and system T types
- $\Rightarrow$ a framework providing a uniform construction mechanism for key concepts of finite mathematics: finite functions, sets, trees, graphs, digraphs, DAGs etc.
- the code shown in the paper is at: `http://logic.cse.unt.edu/tarau/research/2010/shared.hs`.

Future work/Open problems

General open problem:

- Given a bijective mapping between two datatypes, transport recursive algorithms between them i.e. can we derive automatically such algorithms?
- Can we make this derivation such that correctness and termination proofs can be automatically extracted from the derivation process?

(Manually) solved instances described in the paper:

- The successor and predecessor operations s and p, order relations and arithmetics operations on HFS, HFF and System **T** types can be seen as (manually) derived from their counterparts working on BitStacks.