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 $\epsilon$-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 $\lambda$-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, Cegielski 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 $\epsilon$-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-interpretablity described in [Kaye and Wong 2007], our distinct executable specification as a Haskell type class provides unique insights into the shared inductive con- structs 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.

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

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 arithmetical 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 \(\{0, 1\}^\ast\) 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 over-riding) 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:

```haskell
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, \(\text{bit0}_{-}\), \(\text{push0}_{-}\), \(\text{push1}_{-}\), \(\text{pop}_{-}\) 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:

```haskell
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 \(u_\).

```haskell
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:

```haskell
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.

```haskell
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 arith-
metics 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

a SharedAxioms> view (A 2) :: Peano
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 ax-
ioms 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’ = 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:

a SharedAxioms> view (Succ (Succ Zero)) :: A
2

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

a SharedAxioms> view (A 42) :: BitStack
Bit1 (Bit1 (Bit0 (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, \(\epsilon\)-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
denoted \(\emptyset\) 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 xs)) where
S ys = p (S xs)
f (x : xs) | e_ x = xs
f xs = xs
s (S xs) = s (hLift (S [] :_)) where
hLift k [] = [k]
hLift k (x : xs) | k = x = hLift (s xs)
hLift k xs = k : xs
p (S xs) = S (hUnLift xs) where
hUnLift (S []) :_ = 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 a 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.
A proof by induction that types A and S implement indeed the same
successor and predecessor operations 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 interpreration 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
\[ f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x + 1 & \text{otherwise} \end{cases} \]

Will we see later, through a reflection mechanism that parametersizes 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 arithmetical 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
\( \text{SharedAxioms} \) or \( \text{hinc} \)). They are
extend the inductive mechanism used to build hereditarily finite sets
Hereditarily finite functions, described in detail in [Tarau 2009b],
are co-recursive
A 43
\[ \text{SharedAxioms} \] view (A 5) :: F
\[ \text{F} \] [\[ F \],F [\[ F \]]] [F [\[ F \]]]
\[ \text{SharedAxioms} \] o it
\[ \text{F} \] [\[ F \],F [\[ F \]]] [F [\[ F \]]]
\[ \text{SharedAxioms} \] view it :: A
\[ \text{A} \] 11

6. Arithmetic operations

Our next refinement adds key arithmetic operations in the form of
a type class extending Polymath. We start with addition:
\[ \text{class} \ (\text{Polymath} \ n) \Rightarrow \text{Peyano} \ n \text{ where} \]
\[ a :: n \rightarrow n \] 4
\[ a x y | e_ x = y \]
\[ a x y | e_ y = x \]
\[ a x y | o_ x \&\& o_ y = 1 (a (r x) (r y)) \]
\[ a x y | i_ x \&\& i_ y = 0 (a (s (a (r x) (r y)))) \]
\[ a x y | i_ x \&\& i_ y = 1 (a (s (a (r x) (r y)))) \]

It is time to cheat on subtraction \( \text{sb} \) and comparison \( \text{lt} \) (standing for less than) - we only provide here simple/slow/short Peano-style implementations
\[ \text{sb} :: n \rightarrow n \] 4
\[ \text{sb} x y | e_ x = e \]
\[ \text{sb} x y | e_ y = y \]
\[ \text{sb} x y = \text{sb} (p x) (p y) \]
\[ \text{lt} :: n \rightarrow n \rightarrow \text{Bool} \]
\[ \text{lt} x y | e_ x \&\& e_ y = \text{False} \]
\[ \text{lt} x y | e_ x \&\& \text{not (e_ y)} = \text{True} \]
\[ \text{lt} x y | \text{not (e_ x)} \&\& e_ y = \text{False} \]
\[ \text{lt} x y = \text{lt} (p x) (p y) \]
and leave as an exercise to the reader to define (along the lines of a)
more efficient ones. Note that \( \text{sb} \) is defined as a total function that
computes the absolute value of the difference of the two numbers. Note also that it 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:

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

```haskell
ncompare :: n → n → Ordering
ncompare x y | x == y = EQ
ncompare x y | x < y = LT
ncompare x y | otherwise = GT
```

After adding the instances

```haskell
instance Polymath1 F
instance Polymath1 S
instance Polymath1 BitStack
instance Polymath1 A
```

one can see that all operations extend naturally:

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

8. Adding other arithmetic operations

We first define multiplication and integer division.

```haskell
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)))
```

```haskell
db :: n → n → double
db = p . o
```

```haskell
hf :: n → n → half
hf = r . s
```

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

```haskell
-- simple (slow) division with remainder
sd :: n → n → (n, n)
sd x y = (q, p r) where
(q, r) = sd' (a x) y
sd' x y | e_ x = (e, e)
sd' x y = z where
x_ y = m x y
z = if e_ x then (e, x) else (s q, r) where (q, r) = sd' x_ 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.

```haskell
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))
```

```haskell
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)
```

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

\[
\begin{align*}
\text{instance Polymath3 } & A \\
\text{instance Polymath3 } & \text{Peano} \\
\text{instance Polymath3 } & \text{BitStack} \\
\text{instance Polymath3 } & S \\
\text{instance Polymath3 } & F
\end{align*}
\]

operations can be tested under various representations

\text{sharedAxioms} view (A 6) :: F
\text{F} \text{F} \text{F} \text{F}
A
A
A
36

\text{sharedAxioms} view it :: BitStack
Bit1 (Bit0 (Bit1 (Bit0 Empty)))

\text{sharedAxioms} ad it (Bit0 Empty) :: (BitStack, BitStack)
(Bit1 (Bit0 (Bit0 Empty))), Empty

\text{sharedAxioms} view it :: A
A
A
18

\text{sharedAxioms} to_bin (A 3)
[A 1, A 1]

\text{sharedAxioms} from_bin it
A
3

\text{sharedAxioms} view it :: BitStack
Bit0 (Bit0 Empty)

We will next introduce \textit{pairing functions}. They are used to parameterize mappings between finite sets and natural numbers.

\section{Pairing functions}

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

\begin{align*}
\text{data Pairing } n &= \\
\text{Pairing} (p2 :: (n \rightarrow n \rightarrow n), p0 :: n \rightarrow n, p1 :: n \rightarrow n)
\end{align*}

\subsection{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 \texttt{ppair}, together with its left and right unpairing companions \texttt{pfirst} and \texttt{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 \texttt{ppair} combines two numbers reversibly by multiplying a power of 2 derived from the first and an odd number derived from the second:

\[
f(x, y) = 2^x(2y + 1) - 1
\]

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 \texttt{pfirst} and \texttt{psecond} and the 3 functions are grouped together as the record \texttt{ppairing}.

\begin{align*}
\text{class (Polymath3 n) }& \Rightarrow \text{Polymath4 n where} \\
\text{ppairing} & : \text{Pairing } n \\
\text{ppairing} & = \text{Pairing} (p2=\text{ppair}, p0=\text{pfirst}, p1=\text{psecond}) \\
\text{ppair} & : n \rightarrow n \\
\text{ppair} x y & = p (\text{lcons } x y) \text{ where} \\
\text{lcons } x y s & = s (\text{lcons'} x (\text{db } y s)) \\
\text{lcons'} x y s & \mid e_0. x = y s \\
\text{lcons'} x y s & = o (\text{lcons'} (p x) y)
\end{align*}

pfirst :: n \rightarrow n
pfirst z = lhead (s z) where
lhead = h . p
h x s | o_. x s = s (h (hf xs))
h_ = e

psecond :: n \rightarrow n
psecond z = ltail (s z) where
ltail = h f . t . p
t x s | o_. x s = t (hf xs)
t x s = xs

The next pairing function works in a way similar to the zip operation on \texttt{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 \texttt{e}, our null element, is used when needed.

\begin{align*}
\text{bpairing} & : \text{Pairing } n \\
\text{bpairing} & = \text{Pairing} (p2=\text{bpair}, p0=\text{bfirst}, p1=\text{bsecond}) \\
bpair & : n \rightarrow n \rightarrow n \\
bpair x y & = \text{fromBin } (\text{bpair'} \ (\text{toBin } x) \ (\text{toBin } y)) \text{ where} \\
\text{bpair'} & \ [\ ] = [] \\
\text{bpair'} & [y] s = e: (\text{bpair'} \ y s ) \\
\text{bpair'} & (x:xs) y s = x: (\text{bpair'} \ y s xs)
\end{align*}

bfirst :: n \rightarrow n
bfirst = \text{fromBin . deflate . toBin}

bsecond :: n \rightarrow n
bsecond = \text{fromBin . second' . toBin where}
second' [] = []
second' (x:xs) = deflate xs

deflate :: [n] \rightarrow [n]
deflate [] = []
deflate (x:xs) = x: (deflate xs)
deflate [x] = [x]

After adding the instances

\begin{align*}
\text{instance Polymath4 } & A \\
\text{instance Polymath4 } & \text{Peano} \\
\text{instance Polymath4 } & \text{BitStack} \\
\text{instance Polymath4 } & S \\
\text{instance Polymath4 } & F
\end{align*}

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

\text{sharedAxioms} bpair (A 3) (A 4)
A
37

\text{sharedAxioms} bfirst (A 37)
A
3

\text{sharedAxioms} bsecond (A 37)
A
4

\text{sharedAxioms} ppair (A 3) (A 4)
A
71

\text{sharedAxioms} pfirst (A 71)
A
3

\text{sharedAxioms} psecond (A 71)
A
4

\subsection{Parameterizing on a pairing function}

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

\begin{align*}
\text{class (Polymath4 n) }& \Rightarrow \text{Polymath5 n where} \\
\text{pairing} & : \text{Pairing } n \\
\text{pairing} & = \text{ppairing } \rightarrow \text{default pairing } \rightarrow \text{override}
\end{align*}
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_mset_list :: [n] → [n]
as_mset_list x = cons x (as_list_list x)
```

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

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

The mapping to multisets is derived in a similar way:

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

### 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 specific edge types, i.e. ordered pairs for digraphs, unordered pairs and lists to derive, from a given pairing function, representations for

```
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_list [x,y]

upwardUnpair :: n → (n,n)
upwardUnpair z = (x',y') where
  (x,y)=ordUnpair z
```

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)
```

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 induces the Ackermann interpretation as the bijection between \( N \) and \( Z^+ \). 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]) → n
  setOp2 :: (n → n) → (n → n) → n
  setOp2 op x y = as_set_nat (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 → 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.

\[
\text{powset} :: n \rightarrow n
\]
\[
\text{powset } x = \text{as_nat_set} \left(\text{map } \text{as_nat_set} \left(\text{subsets } \text{as_set_nat } x\right)\right) \text{ where }
\]
\[
\text{subsets } [ ] = [ [ ] ]
\]
\[
\text{subsets } (x:xs) = [xs|ys=y\leftarrow\text{subsets } xs,xs\rightarrow[y),(x:ys)]
\]

Next, the \(e\)-relation defining set membership is given as the function \(\text{inSet}\), together with the augment function used in various set theoretic constructs as a new set generator.

\[
\text{inSet} :: n \rightarrow n \rightarrow \text{Bool}
\]
\[
\text{inSet } x y = \text{setIncl } \left(\text{as_set_nat } x\right) y
\]
\[
\text{augment} :: n \rightarrow n
\]
\[
\text{augment } x = \text{setUnion } x \left(\text{as_set_nat } x\right)
\]

The \(n\)th 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 \(\text{nthOrdinal}\):

\[
\text{nthOrdinal} :: n \rightarrow n
\]
\[
\text{nthOrdinal } x | e_\cdot x = e
\]
\[
\text{nthOrdinal } n = \text{augment } \text{nthOrdinal } (p\ n)
\]

After defining the appropriate instances

\[
\text{instance Polymath6 A}
\]
\[
\text{instance Polymath6 Peano}
\]
\[
\text{instance Polymath6 BitStack}
\]
\[
\text{instance Polymath6 S}
\]
\[
\text{instance Polymath6 F}
\]

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

\[
\text{listOp2} \ (op \ (\text{as_list_nat } x) \ (\text{as_list_nat } y))
\]

Another mechanism for defining list operations is to use a “structured recursion combinator” like \(\text{foldr}\) from which various other operations can be derived.

\[
\text{listFoldr} :: (n \rightarrow n \rightarrow n) \rightarrow n \rightarrow n \rightarrow n
\]
\[
\text{listFoldr } f z xs | e_\cdot xs = z
\]
\[
\text{listFoldr } f z xs = f \ (\text{hd} \ xs) \ (\text{listFoldr } f z (\text{tl} \ xs))
\]

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.

\[
\text{newtype } B = \text{Integer deriving } (\text{Eq}, \text{Show}, \text{Read})
\]

\[
\text{instance Polymath7 B where}
\]
\[
e = B 0
\]
\[
o_\cdot (B x) = \text{odd } x
\]
\[
o (B x) = B (2*x+1)
\]
\[
i (B x) = B (2*x+2)
\]
\[
r (B x) | x/=0 = B ((x-1) \ 'div' \ 2)
\]

\[
\text{instance Polymath1 B}
\]
\[
\text{instance Polymath2 B}
\]
\[
\text{instance Polymath3 B}
\]
\[
\text{instance Polymath4 B}
\]
\[
\text{instance Polymath5 B where pairing=bpairing}
\]
\[
\text{instance Polymath6 B}
\]
\[
\text{instance Polymath7 B}
\]

As expected, the pair function acts differently:
instance Polymath 3 Integer  where
\[ \text{arithmetic operations:} \]
\[ \text{performance of the underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
\[ \text{13. Deriving an instance with fast bitstring} \]
\[ \text{operations} \]
\[ \text{We will now benefit from our shared axiomatization by designing} \]
\[ \text{an instance that takes advantage of bit operations to implement,} \]
\[ \text{through a few overrides, fast versions of various operations.} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{underlying C-based GMP package.} \]
\[ \text{First some} \]
\[ \text{Haskell's arbitrary length Integer type to benefit in GHC from the} \]
\[ \text{For syntactic convenience, we will map this instance directly to} \]
\[ \text{ment, through a few overrides, fast versions of various operations.} \]
\[ \text{ing an instance that takes advantage of bit operations to imple-} \]
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 \( \frac{\binom{n}{2}}{2} \) 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 con-
text 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
that is transported between virtual types

\[ \text{borrow(x,y)} = \text{hub using the combinator} \]

One can route isomorphisms between two virtual types through the
hub using the combinator \( \text{borrow} \):

\[ \text{borrow_from lender f borrower x y} = \]

A one argument function \( f \) is transported between virtual types
using the combinator \( \text{borrow_from} \):

\[ \text{borrow_from :: Type b \to (b \to b) \to Type a \to a} \]

Similarly, a two argument function \( \text{op} \) is transported between vir-
tual types using the combinator \( \text{borrow_from2} \):

\[ \text{borrow_from2 :: Type a \to (a \to a \to a) \to Type b \to b \to b} \]

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

\[ \text{list :: Type [Integer]} \]
\[ \text{list = Iso as_nat_list as_list_nat} \]
\[ \text{set :: Type [Integer]} \]
\[ \text{set = Iso as_nat_set as_set_nat} \]
\[ \text{mset :: Type [Integer]} \]
\[ \text{mset = Iso as_nat_mset as_mset_nat} \]

One can now try out the combinator ‘as’ working exactly like
its concrete counterparts:

\[ \text{as :: Type a} \]
\[ \text{as = \text{hub using the combinator} as} \]
\[ \text{One can route isomorphisms between two virtual types through the} \]

\[ \text{nat :: Type Integer} \]
\[ \text{nat = itself} \]

\[ \text{class (Polymath7 n) \Rightarrow Polymath8 n where} \]
\[ \text{as_set_digraph :: [(n,n)] \to [n]} \]
\[ \text{as_set_digraph = map ordPair} \]
\[ \text{as_digraph_set :: [n] \to [(n,n)]} \]
\[ \text{as_digraph_set = map ordUnpair} \]

\[ \text{15. Directed Graphs, DAGs, Undirected graphs} \]
\[ \text{and Hypergraphs} \]

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.

\[ \text{15.1 Set Encodings of Directed Graphs} \]

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.

\[ \text{15.2 Set Encodings of Undirected Graphs} \]

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

\[ \text{15.3 Set Encodings of DAGs} \]

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

\[ \text{15.4 Encoding Hypergraphs} \]

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).

\[ \text{as_hypergraph_set :: [n] \to [n]} \]
\[ \text{as_hypergraph_set = map (as_set_nat . s)} \]
\[ \text{as_set_hypergraph :: [n] \to [n]} \]
\[ \text{as_set_hypergraph = map (p . as_set_nat_set)} \]

We conclude this by updating our instance definitions
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 if the function nsyr always terminates:

\[ \text{syr } n = 1 \text{ (a (m six n) four) where} \]
\[ \text{four } = s \text{ (s (s four))} \]
\[ \text{six } = s \text{ (s four)} \]

\[ \text{nsyr } n \mid e_ n = [e] \]
\[ \text{nsyr } n = n : \text{nsyr (syr } n) \]

The first 8 sequences are computed as follows:

\[ n \text{sy}r \text{ } n = \text{map (nsyr) } [0..7] \]
\[ [[0],[1,2,0],[2,0],[3,5,8,6,2,0],[4,3,5,8,6,2,0],[5,8,6,2,0],[6,2,0],[7,11,17,26,2,0]] \]

Timing \text{nsyr} for 123456780, and then the same digits repeated twice and three times, for functions \text{cI}, \text{cA}, \text{cK}, \text{cF} and \text{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. \text{cF}) vs. hereditarily finite sets (col. \text{cS}) and of symbolic BitStack computations \text{cK} vs. “unaccelerated” Integer operations \text{cA}. Integer operations accelerated with overridings and bit operations \text{cI} are faster by constant factors that are significant, but not as dramatic as one might expect.

\text{cI} \text{ c } = \text{c } :: \text{Integer}
\text{cA } = \text{view (c } :: \text{Integer}) : : \text{A}
\text{cK } = \text{view (c } :: \text{Integer}) : : \text{BitStack}
\text{cF } = \text{view (c } :: \text{Integer}) : : \text{F}
\text{cS } = \text{view (c } :: \text{Integer}) : : \text{S}

<table>
<thead>
<tr>
<th>\text{bitsize of input}</th>
<th>\text{cI}</th>
<th>\text{cA}</th>
<th>\text{cK}</th>
<th>\text{cF}</th>
<th>\text{cS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>7</td>
<td>80</td>
<td>62</td>
<td>153</td>
<td>311</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>163</td>
<td>120</td>
<td>327</td>
<td>1084</td>
</tr>
<tr>
<td>97</td>
<td>17</td>
<td>366</td>
<td>261</td>
<td>720</td>
<td>3604</td>
</tr>
</tbody>
</table>

Figure 1. Timings for \text{cI}, \text{cA}, \text{cK}, \text{cF}, \text{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


