Arithmetic with Free Algebras and Hereditarily Finite Sets: a Natural Bridge between Numeric and Symbolic Computations

Paul Tarau

University of North Texas

Friday 09/28/2012,13:30-14:20

Motivation

- we answer positively two questions that one might be curious about:
  - can we do arithmetic directly with some “symbolic” mathematical objects - e.g. binary trees, balanced parenthesis languages, hereditarily finite sets?
  - is this alternative arithmetic efficient enough to be practical?
- background: bijective Gödel numberings for fundamental data types => we can borrow computations
- here we will use isomorphisms of free algebras to actually build our computations from scratch
  - free algebras are widely used in programming languages: they correspond to recursive data types like lists or trees
  - bijections from free algebras provide compact representations for non-free data types like sets, multisets, graphs and Turing-equivalent computational mechanisms like combinators
- methodology: fully replicable research (by contrast: “cold fusion” :-))
  - $\Rightarrow$ “literate programming”: the code is extracted directly from these slides
  - $\Rightarrow$ an executable guided tour to an alternative view of arithmetic

A Freedom Quote

No one is more of a slave than he who thinks himself free without being so.

JOHANN WOLFGANG VON GOETHE, The Maxims and Reflections of Goethe

Outline

1. Free algebras and data types
2. The Peano algebra `AlgU`
3. Successor and predecessor in the two-successor algebra `AlgB`
4. Binary Arithmetic in `AlgB`
5. Successor and predecessor in the algebra of binary trees `AlgT`
6. Arithmetic computations in `AlgT`
7. Representing lists, sets, multisets
8. Arithmetic with Hereditarily Finite Sets
9. Arithmetic with balanced parenthesis languages
10. Computing with the Calkin-Wilf bijection
11. Conclusion

Free algebras

Definition

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

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

- alternatively, free algebras can be seen as *initial objects* in the category of algebraic structures
- free algebras can be axiomatized in predicate logic by defining constructors, deconstructors and recognizers
- conversely, the language of predicate logic itself is built from:
  - function constructors (generating the *Herbrand Universe*)
  - predicate constructors (generating the *Herbrand Base*)

Free algebras as data types

the Haskell declarations

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

correspond, respectively to

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

note: a copy of these slides is at http://logic.cse.unt.edu/tarau/research/2012

Magmas: a "classic" set-theoretical view

Definition

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

Definition

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

Definition

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

Morphisms of magmas

Proposition

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

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

Proposition

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

It follows that

Proposition

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

The AlgT datatype as a free magma

```
data AlgT = T | C AlgT AlgT
```

We will identify the data type `AlgT` with the free magma generated by the set `{T}` and denote its binary operation $x * y$ as `C x y`. It corresponds to the free algebra defined by the signature `{T/0, C/2}`.

Proposition

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

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

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

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

Peano algebra

- it also occurs under a few alternate names:
  - the *one successor* free algebra
  - unary natural numbers
  - the language of the monoid $\{0\}^*$
  - the language of the decidable systems WS1S and S1S
  - "cave-man's" numbering system: I, II, III, IIII, ... ~20000 years ago
- it is defined by the signature `{U/0, S/1}`, where `U` is a constant (seen as zero) and `S` is the unary successor function symbol
- we denote it `AlgU` and identify it with its corresponding Haskell data type

```
data AlgU = U | S AlgU
```

The data type `AlgU` as a free algebra

Proposition

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

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

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

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

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

The *two successor* free algebra

- it also occurs under a few alternate names:
  - bijective base-2 natural numbers
  - the language of the monoid $\{0, 1\}^*$
  - the language of the decidable systems WS2S and S2S
- it is defined by the signature `{B/0, O/1, I/1}` where
  - `B` is a constant (seen as denoting the empty sequence)
  - `O,I` are two unary successor function symbols
- we denote `AlgB` this algebra and identify it with its corresponding Haskell data type

```
data AlgB = B | O AlgB | I AlgB
```

The data type `AlgB` as a free algebra

Proposition

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

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

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

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

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

Borrowing Arithmetic from the Peano Algebra

- we know how to do (unary) arithmetic in Peano algebra `AlgU`
- defining isomorphisms between `AlgU, AlgB` and `AlgT` will enable such arithmetic operations on `AlgB` and `AlgT`
- we need to define bijections that commute with
  - the successor operation
  - the predecessor operation
  - the predicate recognizing the zero element `U`
- one can think about these functions as bijective Gödel numberings connecting objects of `AlgB` and `AlgT` to natural numbers, seen as objects of `AlgU`
- one can also think about emulating constructor operations in one algebra with equivalent (possibly more complex) computations in another algebra

A Freedom Quote

Freedom's just another word for nothing left to lose.

KRIS KRISTOFFERSON, "Me and Bobby McGee"

- $\Rightarrow$ no information will be lost by "commuting" between algebras - we will ensure that our morphisms are bijections

Successor and predecessor in AlgB

The intuition for designing these operations is their conventional arithmetic interpretation, as 0 for B, $\lambda x.2x+1$ for O and $\lambda x.2x+2$ for I.

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

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

language notes:

- one can think about our Haskell code simply as equational rewriting rules
- pattern matching: the first match activates the “rewritng rule”
- or, inductive definitions / recursion equations working on a free algebra

Correctness of our successor and predecessor emulation

Proposition

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

Proof.

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

The isomorphism between AlgU and AlgB

The functor u2b defined as

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

and its inverse

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

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

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

```
binNats = iterate sB B
```

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

A Freedom Quote

Freedom is something that dies unless it's used.

HUNTER S. THOMPSON, Ancient Gonzo Wisdom

⇒ we will use the free algebra `AlgB` to define binary arithmetic

Binary arithmetic in `AlgB`

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

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

- performance moves from $O(n)$ in the Peano algebra to $O(log(n))$
- effort is now proportional to the size of the binary representation!
- structural recursion ⇒ formally verified with the proof assistant Coq

The intuitions behind the arithmetic operations on AlgT

The intuitions we have used for designing the successor (s) and predecessor operations (s') in AlgT and their helper functions d and d': their "conventional" arithmetic interpretations!

- $\lambda x.x+1$ for s
- $\lambda x.x-1$ for s' assuming $x>0$
- 0 for T
- $\lambda x.\lambda y.2^x(2y+1)$ for C
- $\lambda x.2x$ for d (assuming $x>0$)
- $\lambda x.x/2$ (assuming x even and $x>0$) for d'

(somewhat) related:

- hereditary base-k notation in the proof of Goodstein's theorem
- good old floating point + recursion on the representation of the exponent
- run-length compression of 0's in a binary string

Defining the Successor and Predecessor on `AlgT`

This time, the definitions of successor `s` and predecessor `s'`, together with the helper functions `d` (double) and `d'` (half of an even) are mutually recursive:

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

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

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

Correctness of the successor and predecessor definitions

Proposition

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

To prove this we will use the structural induction principle on `AlgT`:

Proposition

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

The Proof

Proof.

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

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

Our main induction proof, by case analysis: rules k and k' are such that rule "- k -" is the unique match for function f if and only if rule "- k' -" is the unique match for function f'. □

The Proof - continued

We will show that $s\ u = v \iff s'\ v = u$, assuming it holds inductively forall a, b such that $v = C\ a\ b$. Note that case `k = 1, 2, 3, 4` corresponds to the application of rules "`- k -`" and "`- k' -`" in the definitions of `s`, `s'` and `d, d'`.

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

The isomorphism between AlgU and AlgT

The functor u2b defined as

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

and its inverse

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

define an isomorphism which allows us to see AlgT as a model for an axiomatization of arithmetic on $\mathbb{N}$. The infinite stream treeNats of binary trees, corresponding to successive natural numbers is defined as:

```
treeNats = iterate s T
```

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

Conversion between ordinary and binary tree naturals

```
data AlgT = T | C AlgT AlgT

type N = Integer

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

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

Can we do arithmetic computations in AlgT?

- as we have emulated the successor operations we can do easily (slow) unary arithmetic
- defining a AlgB "view" over the free algebra AlgT enables *fast arithmetic computations with binary trees*
- complexity will be comparable to operations acting on conventional bitstring representations

projection functions (c', c'') and a recognizer of non-empty trees c_:

```
c',c'' :: AlgT → AlgT

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

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

Emulating AlgB in AlgT

```
data AlgB = B | O AlgB | I AlgB
data AlgT = T | C AlgT AlgT
```

constructors (o, i), destructors (o' , i') and recognizers (o_, i_):

```
o, o', i, i' :: AlgT → AlgT
o_, i_ :: AlgT → Bool

o = C T
o' (C T y) = y
o_ (C x _) = x == T

i = s . o
i' = o' . s'
i_ (C x _) = x /= T
```

The isomorphism between AlgB and AlgT

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

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

- note that interplay between actual constructors and their emulation
- a constructor symbol F/n is emulated by a recognizer predicate f_/n, a constructor function f/n and a destructor function f'/n

Efficient arithmetic in AlgT: addition

We are now ready for the magic: arithmetic operations working directly on binary trees.

```
add T y  = y
add x T  = x
add x y | o_ x && o_ y = i (add (o' x) (o' y))
add x y | o_ x && i_ y = o (s (add (o' x) (i' y)))
add x y | i_ x && o_ y = o (s (add (i' x) (o' y)))
add x y | i_ x && i_ y = i (s (add (i' x) (i' y)))
```

- everything happens naturally through the emulation of AlgB
- once we have defined i, i', o, o', o_, i_, the operations on AlgT look syntactically identical to those on AlgB
- using type classes one can actually share the implementation

```
sub x T = x
sub y x | o_ y && o_ x = s' (o (sub (o' y) (o' x)))
sub y x | o_ y && i_ x = s' (s' (o (sub (o' y) (i' x))))
sub y x | i_ y && o_ x = o (sub (i' y) (o' x))
sub y x | i_ y && i_ x = s' (o (sub (i' y) (i' x)))
```

a generic tester:

```
testop f n m = t2n (f (n2t n) (n2t m))
```

```
> testop sub 20 15
5
> testop add 20 15
35
> add (n2t 20) (n2t 15)
C T (C T (C (C T (C T T)) T))
```

Efficient arithmetic in AlgT: comparison

```
cmp T T  = EQ
cmp T _ = LT
cmp _ T = GT
cmp x y | o_ x && o_ y = cmp (o' x) (o' y)
cmp x y | i_ x && i_ y = cmp (i' x) (i' y)
cmp x y | o_ x && i_ y = strengthen (cmp (o' x) (i' y)) LT
cmp x y | i_ x && o_ y = strengthen (cmp (i' x) (o' y)) GT

strengthen EQ stronger = stronger
strengthen rel _ = rel
```

Efficient arithmetic in AlgT: multiplication

we optimize a bit, using the arithmetic interpretation of our binary trees

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

```
> multiply (n2t 42) (n2t 10)
C (C (C T T) T) (C (C (C T T) T) (C (C T T) (C T T)))
> testop multiply 42 10
420
> testop multiply 1234567890 9876543210
12193263111263526900
```

A Freedom Quote

Liberty, when it begins to take root, is a plant of rapid growth.

GEORGE WASHINGTON

⇒ a $O(1)$ complexity power of 2 operation `exp2` is simply:

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

this leads to a compact representation of towers of exponents of 2 (tetration):

$2^{2^{2^{\dots 2}}}$ ⇒ `C(C(C(...(C T T))),T)`

An emergent property: operations with towers of exponents

- our tree representation supports operations with gigantic, tower of exponent numbers
- with conventional bitstring representations, such numbers would overflow even if each atom in the known universe were used as bit ...

iterating `exp2` 7 times):

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

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

note: “`it`” represents in Haskell the result of the previous query

A Freedom Quote

Every general increase of freedom is accompanied by some degeneracy, attributable to the same causes as the freedom.

CHARLES HORTON COOLEY, Human Nature and the Social Order

- this can indeed happen, the worse case is $2^{2^{2^{\dots 2^n}}} - 1$
- it means that we can (sometime) fall back to the same thing as with the usual binary string computations
- good news - from a result proven by Legendre on the number of occurrences of a prime p in $n!$:
  - the average number of iterations for successor and predecessor in `AlgB` for k between 0 and $2^n - 1$ is $1 + \frac{2^n-1}{2^n} < 2$
  - the analysis for `AlgT` is more convoluted but (empirically) the complexity of `s` and `s'` is close to a constant factor
- even better news - see the slide after the conclusion !

Representing lists

we encode lists by by repeated application of constructors and deconstructors

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

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

> n2t 888
C (C T (C T T)) (C T (C T (C T (C (C T T) (C T T)))))
> to_list it
[C T (C T T),T,T,T,C T T,T]
> from_list it
C (C T (C T T)) (C T (C T (C T (C (C T T) (C T T)))))
> t2n it
888
```

Representing multisets

to encode multisets we go through a bijection between lists and multisets

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

list2mset ns = tail (scanl add T ns)
mset2list ms =  zipWith sub ms (T:ms)
to_mset :: AlgT → [AlgT]
to_mset = list2mset . to_list

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

> (map t2n . list2mset . map n2t) [2,0,1,2]
[2,2,3,5]
> (map t2n . mset2list . map n2t) it
[2,0,1,2]
```

Representing sets

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

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

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

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

> (map t2n . list2set . map n2t) [2,0,1,2]
[2,3,5,8]
> (map t2n . set2list . map n2t) it
[2,0,1,2]
```

Hereditarily Finite Sets

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

Ackermann's encoding of Hereditarily Finite Sets as natural numbers:

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

same in Haskell - quite easy to invert

```
hfs2nat t = rank set2nat t
rank g (H ts) = g (map (rank g) ts)
set2nat ns = sum (map (2^) ns)
```

- not a free algebra anymore - sets are constrained to have distinct elements and assumed to be canonically represented using an ordering relation between elements
- but Ackermann's mapping allows us to exploit the bijection with $\mathbb{N}$ and define operations that are total on canonically represented sets

A Freedom Quote

For you who no longer posses it, freedom is everything, for us who do, it is merely an illusion.

EMIL CIORAN, History & Utopia

- we can derive arithmetic operations on Hereditarily Finite Sets through a series of transformations to the free algebra `AlgT`
- the derivation steps proceed along the lines of Ackermann's bijection

The acyclic digraph representing a Hereditarily Finite Set

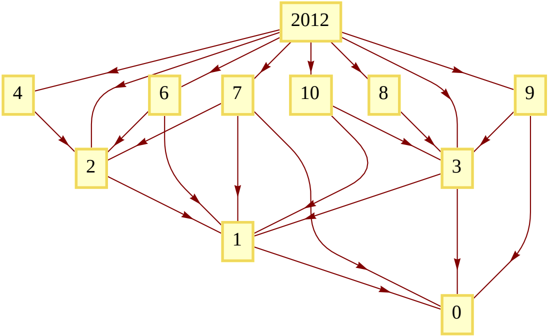

Figure : 2012 as a Hereditarily Finite Set through Ackermann's bijection

Defining Successor `sH` and Predecessor `sH'` on a multiway tree representation of Hereditarily Finite Sets

```
sH (H xs) = H (lift (H []) xs)

sH' (H (x:xs)) = H (lower x xs)

lift k (x:xs) | k == x = lift (sH k) xs
lift k xs = k:xs

lower (H []) xs = xs
lower k xs = lower l (l:xs) where l = sH' k
```

Emulating the two successor algebra AlgB

```
-- "empty" and its recognizer
eH = H []
eH_ x = x == H []

-- constructors
oH (H xs) = sH (H (map sH xs))
iH = sH . oH

-- destructors
oH' x | oH_ x = H (map sH' ys) where H ys = sH' x
iH' x = oH' (sH' x)

-- recognizers
oH_ (H (x:_)) = eH_ x
iH_ x = not (eH_ x || oH_ x)
```

⇒ (fast) arithmetic computations are similar to those on AlgB, AlgT

A Catalan isomorphism: modeling `AlgT` with a balanced parenthesis language

```
data Par = L | R deriving (Eq,Show,Read)

-- deconstructs a list of balanced parentheses into (head,tail)
decons (L:ps) = (reverse hs, ts) where
   (hs,ts) = count_pars 0 ps []
  count_pars 1 (R:ps) hs = (R:hs,L:ps)
  count_pars k (L:ps) hs = count_pars (k+1) ps (L:hs)
  count_pars k (R:ps) hs = count_pars (k-1) ps (R:hs)

-- constructs a list of balanced parentheses from (head,tail)
cons (xs, L:ys) = L:xs ++ ys

-- constructor + recognizer for empty
eP = [L,R]
eP_ x = (x == eP)
```

Successor (sP) and Predecessor (sP')

```
-- successor
sP z | eP_ z = cons (eP,eP)                              -- 1  --
sP z | eP_ x = dP (sP y) where (x,y) = decons z          -- 2  --
sP z = cons (eP, dP' z)                                  -- 3  --

-- predecessor
sP' z | eP_ x && eP_ y = eP where (x,y) = decons z       -- 1' --
sP' z | eP_ x = dP y where (x,y) = decons z              -- 3' --
sP' z = cons (eP, sP' (dP' z))                           -- 2' --

-- double
dP z = cons (sP a,b) where (a,b) = decons z              -- 4  --

-- half of non-zero even
dP' z = cons (sP' a,b) where (a,b) = decons z            -- 4' --
```

Enumerating Positive Rationals with the Calkin-Wilf tree

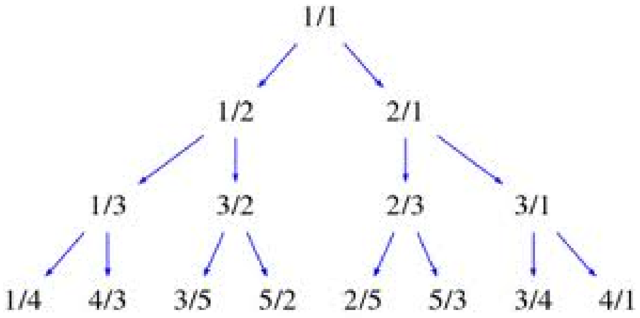

Figure : The Calkin-Wilf Tree

The Calkin-Wilf bijection: encoding paths as AlgB elements

Positive rationals in $\mathbb{Q}^+$ are represented as pairs of positive co-prime natural numbers. We first show the bijection using ordinary integers.

$\mathbb{N} \to \mathbb{Q}^+$ using the path in the Calkin-Wilf tree starting with the root

```
n2q 0 = (1,1)
n2q x | odd x =  (f0,f0+f1) where
  (f0,f1) = n2q (div (x-1) 2)
n2q x | even x  = (f0+f1,f1) where
  (f0,f1) = n2q ((div x 2)-1)
```

$\mathbb{Q}^+ \to \mathbb{N}$ using the path in the Calkin-Wilf tree ending with the root

```
q2n (1,1) = 0
q2n (a,b) = f ordrel where
  ordrel = compare a b
  f GT = 2*(q2n (a-b,b))+2
  f LT = 2*(q2n (a,b-a))+1
```

Rationals with binary trees in AlgT

both natural numbers and rationals are represented as binary trees in `AlgT`

$\mathbb{N} \rightarrow \mathbb{Q}^+$ using the path in the Calkin-Wilf tree starting with the root

```
t2q T = (o T,o T)
t2q n | o_ n = (f0,add f0 f1) where (f0,f1)=t2q (o' n)
t2q n | i_ n = (add f0 f1,f1) where (f0,f1)=t2q (i' n)
```

$\mathbb{Q}^+ \rightarrow \mathbb{N}$ using the path in the Calkin-Wilf tree ending with the root

```
q2t q | q == (o T,o T) = T
q2t (a,b) = f ordrel where
  ordrel = cmp a b
  f GT = i (q2t (sub a b,b))
  f LT = o (q2t (a,sub b a))
```

```
> (t2n . q2t . t2q . n2t) 1234567890
1234567890
```

Computing with Rationals

a few more steps are needed:

- extending the bijection to signed rationals
- implementing various operations
- the code, as a Scala package is at: `http://logic.cse.unt.edu/tarau/research/2012/AlgT.scala`

Conclusion

This is fully replicable research: the (self-contained) Haskell code shown in these slides is at: `http://logic.cse.unt.edu/tarau/research/2012/slides_SYNASC_freealg.hs`

- it is possible to implement efficient arithmetic computations on top of free algebras corresponding to data types like binary trees
- isomorphisms between free algebras provide bridges connecting "numeric" and "symbolic" objects
- interesting properties emerge: ability to work with huge numbers – represented as towers of exponents of 2
- non-free data-types like hereditarily finite sets are covered too
- computations can be extended to rationals – resulting in a practical arithmetic package

Research support from NSF grant 1018172 is gratefully acknowledged.

Warm thanks to the SYNASC'2012 organizers for the invitation!

Can we do better? YES, with constructors implicitly compressing contiguous sequences of both Os and Is!

The two binary constructor free algebra, of signature `U/0, V/2, W/2`!

```
data M = M|V M M |W M M deriving (Show, Read, Eq)

eM = M
eM_ x = x == eM

sM x | eM_ x = oM x
sM x | oM_ x = iM (oM' x)
sM x | iM_ x = oM (sM (iM' x))

sM' x | x == oM eM = eM
sM' x | oM_ x = iM (sM' (oM' x))
sM' x | iM_ x = oM (iM' x)
```

the code mimics closely `AlgB`, but the two constructors emulate run-length encodings of sequences of `O` and `I` constructors respectively as `V`, `W`.

Emulating O, I with the two binary constructor free algebra

```
oM (V x y) = V (sM x) y
oM w = V M w

iM (W x y) = W (sM x) y
iM v = W M v

oM' (V M y) = y
oM' (V x y) = V (sM' x) y

iM' (W M y) = y
iM' (W x y) = W (sM' x) y

oM_ (V _ _ ) = True
oM_ _ = False

iM_ (W _ _ ) = True
iM_ _ = False
```

intuition: `oU`, `iU` emulate “single step” operations with the `V`, `W` constructors

The first "natural numbers" with 2 binary constructor trees

```
> mapM_ print (zip [0..] (take 16 (iterate sM M)))
(0,  M)
(1,  V M M)
(2,  W M M)
(3,  V (V M M) M)
(4,  W M (V M M))
(5,  V M (W M M))
(6,  W (V M M) M)
(7,  V (W M M) M)
(8,  W M (V (V M M) M))
(9,  V M (W M (V M M)))
(10, W (V M M) (V M M))
(11, V (V M M) (W M M))
(12, W M (V M (W M M)))
(13, V M (W (V M M) M))
(14, W (W M M) M)          -- note: n iterates of W(x,M) give 2^(n+2)-2
(15, V (V (V M M) M) M)    -- note: n iterates of V(x,M) give 2^(n+1)-1
```

intuition: `oU`, `iU` emulate "single step" operations with the `V`, `W` constructors