# Declarative Modeling of Tree-based Arithmetic Computations

Paul Tarau

University of North Texas

Wed 11/14/2012

# 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

# Outline

# Free algebras

## Definition

*Let $\sigma$ 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 $A_\sigma$ of signature $\sigma$ is defined inductively as the smallest set such that:*

1. *if $c$ is a constant of $\sigma$ then $c \in A_\sigma$*
2. *if $f$ is an n-argument function symbol of $\sigma$, 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 $T$).

# 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 $f : \mathtt{AlgU} \to X$ that verifies*

$$f(U) = u \tag{1}$$

$$f(S(x)) = s(f(x)) \tag{2}$$

*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 $g = <\mathtt{U},\mathtt{S}>$ seen as a bijection $g : 1 + \mathbb{N} \to \mathbb{N}$.

# 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 $f : \texttt{AlgB} \to X$ that verifies*

$$f(B) = b \tag{3}$$

$$f(O(x)) = o(f(x)) \tag{4}$$

$$f(I(x)) = i(f(x)) \tag{5}$$

*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

# 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. □

# 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 $\Rightarrow$ formally verified with the proof assistant Coq

# 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 $s'$.

# 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

# Efficient arithmetic in AlgT: subtraction

```
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
```

# Constant time exponent of 2

$\Rightarrow$ 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):
$2^{2^{2^{\cdots 2}}} \Rightarrow$ `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

# Limitations

- the worse case is $2^{2^{2^{\cdots 2^{n}}}} - 1$
- 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

# Enumerating Positive Rationals with the Calkin-Wilf tree

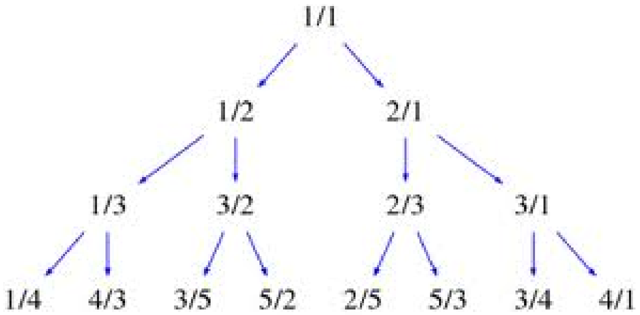


Figure : The Calkin-Wilf Tree

# 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.

$\mathbb{N} \rightarrow \mathbb{Q}^+$ 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)
```

$\mathbb{Q}^+ \rightarrow \mathbb{N}$ 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`

$\mathbb{N} \to \mathbb{Q}^+$ 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)
```

$\mathbb{Q}^+ \to \mathbb{N}$ 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

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
- computations can be extended to rationals – resulting in a practical arithmetic package

*Research support from NSF grant 1018172 is gratefully acknowledged.*