Modeling Finite Mathematics with Isomorphic Data Transformations and Type Classes

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

PART I. A Groupoid of Isomorphic Data Transformations

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

Motivation

- analogies (and analogies between analogies) emerge when we transport objects and operations on them
- this is a creative process - one of the most rewarding ones in terms of interesting outcomes (geometry and coordinates, primes and complex functions, cryptography and number theory, Turing machines and combinators, types and proofs etc.)
- $\rightarrow$ what about a computational catalyst?

- to be able to encode something as something else we need isomorphisms $\rightarrow$ bijections that transport structures
- $\rightarrow$ this is about how we can (somewhat) automate this process by organizing such encodings . . . nicely

Practical uses of datatype isomorphisms?

- the tip of the iceberg:
  - we can transfer operations between datatypes
  - “free algorithms” can emerge
  - sharing opportunities across heterogeneous data types
  - free iterators and random instance generators
  - data compression and succinct representations
  - a general mechanism for serialization and persistence
  - cryptography: encrypt and decrypt are bijections - all our transformations are bijections - some synergy expected!
- more generaly: automate the process of finding “computational analogies” and experimenting with them $\rightarrow$ *calculemus!*

Outline

- a functional programming framework to encode and propagate isomorphisms between elementary data types
- we borrow a few *basic* concepts and results from a few different fields: combinatorics, foundations of set theory, categories, number theory, type theory, graphs theory
- ranking/unranking operations (bijective Gödel numberings)
- pairing/unpairing operations
- generating new isomorphisms through hylomorphisms (folding/unfolding into hereditarily finite universes)
- applications

literate Haskell program – 150++ pages version at `http://logic.csci.unt.edu/tarau/research/2009/fISO.pdf`

The Groupoid of Isomorphisms

```
data Iso a b = Iso (a→b) (b→a)
from (Iso f _) = f
to (Iso _ g) = g

compose :: Iso a b → Iso b c → Iso a c
compose (Iso f g) (Iso f' g') = Iso (f' . f) (g . g')

itself = Iso id id
invert (Iso f g) = Iso g f
```

Proposition

`Iso` *is a groupoid: when defined,* `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Transporting Operations

```
borrow :: Iso t s → (t → t) → s → s
borrow (Iso f g) h x = f (h (g x))
borrow2 (Iso f g) h x y = f (h (g x) (g y))
borrowN (Iso f g) h xs = f (h (map g xs))

lend :: Iso s t → (t → t) → s → s
lend = borrow . invert
lend2 = borrow2 . invert
lendN = borrowN . invert
```

Examples will follow as we populate the universe.

Choosing a Root

```
type Nat = Integer
type Root = [Nat]
```

We can now define an *Encoder* as an isomorphism connecting an object to *Root*

```
type Encoder a = Iso a Root
```

the combinators *with* and *as* provide an *embedded transformation language* for routing isomorphisms through two *Encoders*:

```
with :: Encoder a→Encoder b→Iso a b
with this that = compose this (invert that)

as :: Encoder a → Encoder b → b → a
as that this = to (with that this)
```

The combinator `as`

```
as :: Encoder a→Encoder b→b→a
as that this = to (with that this)
```

```
a2b x = as B A x
b2a x = as A B x
```

$$\begin{tikzcd}
A \arrow[rr, shift left, "a2b = as\ B\ A"] \arrow[dr, shift left, "a"] & & B \arrow[ll, shift left, "b2a = as\ A\ B"] \arrow[dl, shift left, "b^{-1}"] \\
& Root \arrow[ul, shift left, "a^{-1}"] \arrow[ur, shift left, "b"] &
\end{tikzcd}$$

as $[Nat]$ has been chosen as the root, we will define our finite function data type **fun** simply as the identity isomorphism on sequences in $[Nat]$:

```
fun :: Encoder [Nat]
fun = itself
```

Finite Functions to/from Sets

```
*ISO> as set fun [0,1,0,0,4]
[0,2,3,4,9]
*ISO> as fun set [0,2,3,4,9]
[0,1,0,0,4]
```

How we do it?

```
[0, 1,0,0,4]→[0, 2,1,1,5]→[0, 2,3,4,9]
    next slide: 541=2^0+2^2+2^3+2^4+2^9
```

Map lists of natural numbers to strictly increasing sequences of natural numbers representing sets.

Folding sets into natural numbers

We can fold a set, represented as a list of distinct natural numbers into a single natural number, reversibly, by observing that it can be seen as the list of exponents of 2 in the number's base 2 representation.

```
*ISO> as nat set [0, 2,3,4,9]
541
*ISO> as nat fun [0, 1,0,0,4]
541
*ISO> as nat set [3,4,6,7,8,9,10]
2008
*ISO> lend nat reverse 2008 -- order matters
1135
*ISO> lend nat_set reverse 2008 -- order independent
2008
*ISO> borrow nat_set succ [1,2,3]
```

Generic unranking and ranking hylomorphisms

- The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*.
- The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.
- *unranking anamorphism* (*unfold* operation): generates an object from a simpler representation - for instance the seed for a random tree generator
- *ranking catamorphism* (a *fold* operation): associates to an object a simpler representation - for instance the sum of values of the leaves in a tree
- together they form a mixed transformation called *hylomorphism*

Ranking/unranking hereditarily finite datatypes

```
data T = H Ts deriving (Eq,Ord,Read,Show)
type Ts = [T]
```

The two sides of our hylomorphism are parameterized by two transformations `f` and `g` forming an isomorphism `Iso f g`:

```
unrank f n = H (unranks f (f n))
unranks f ns = map (unrank f) ns

rank g (H ts) = g (ranks g ts)
ranks g ts = map (rank g) ts
```

"structured recursion": propagate a simpler operation guided by the structure of the data type obtained as:

```
tsize = rank (λx→1 + (sum x))
```

Extending isomorphisms with hylomorphisms

We can now combine an anamorphism+catamorphism pair into an isomorphism `hylo` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

```
hylo :: Iso b [b] → Iso T b
hylo (Iso f g) = Iso (rank g) (unrank f)

hylos :: Iso b [b] → Iso Ts [b]
hylos (Iso f g) = Iso (ranks g) (unranks f)
```

Hereditarily finite sets

```
hfs :: Encoder T
hfs = compose (hylo nat_set) nat
```

```
*ISO> as hfs nat 42
  H [H [H []],H [H [],H [H []]],H [H [],H [H [H []]]]]
*ISO> as nat hfs it
  42
```

we have just derived as a “free algorithm” *Ackermann’s encoding* from hereditarily finite sets to natural numbers and its inverse!

```
ackermann = as nat hfs
inverse_ackermann = as hfs nat
```

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

Hereditarily Finite Set associated to **2008**

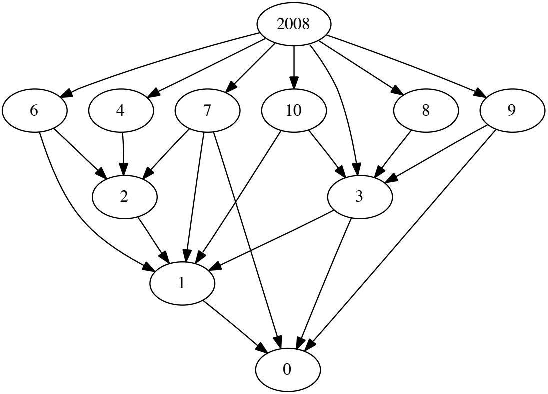


Figure: 2008 as a HFS

Hereditarily finite functions

```
hff :: Encoder T
hff = compose (hylo nat) nat
```

this `hff` Encoder can be seen as another (new this time!) “free algorithm”, providing data compression/succinct representation for hereditarily finite sets (note the significantly smaller tree size):

```
*ISO> as hfs nat 42
  H [H [H []],H [H [],H [H []]],H [H [],H [H [H []]]]]
*ISO> as hff nat 42
  H [H [H []],H [H []],H [H []]]
```

Hereditarily finite function associated to **2008**

Note that edges are labeled to indicate order.

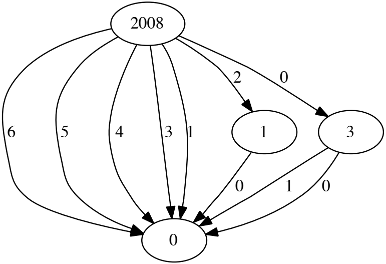


Figure: 2008 as a HFF

Ranking/Unranking of Permutations: factoradics

- The factoradic numeral system replaces digits multiplied by a power of a base n with digits that multiply successive values of the factorial of n.
- In the increasing order variant `fr` the first digit d_0 is 0, the second is $d_1 \in \{0,1\}$ and the n-th is $d_n \in [0..n]$.
- $42 = 0*0! + 0*1! + 0*2! + 3*3! + 1*4!$.

```
fr 42
  [0,0,0,3,1]
rf [0,0,0,3,1]
  42
```

Ranking/Unranking of Permutations: Lehmer codes

The Lehmer code of a permutation f of size n is defined as the sequence $l(f) = (l_1(f) \dots l_i(f) \dots l_n(f))$ where $l_i(f)$ is the number of elements of the set $\{j > i | f(j) < f(i)\}$

Proposition

The Lehmer code of a permutation determines the permutation uniquely.

```
*ISO> nth2perm (5,42)
  [1,4,0,2,3]
*ISO> perm2nth [1,4,0,2,3]
  (5,42)
```

Ranking/unranking of arbitrary finite permutations

To extend the mapping from permutations of a given length to arbitray permutations we “shift towards infinity” the starting point of each new block of permutations as permutations of larger and larger sizes are enumerated.

```
perm :: Encoder [Nat]
perm = compose (Iso perm2nat nat2perm) nat

*ISO> as perm nat 2008
[1,4,3,2,0,5,6]
*ISO> as nat perm it
2008
```

Hereditarily finite permutations

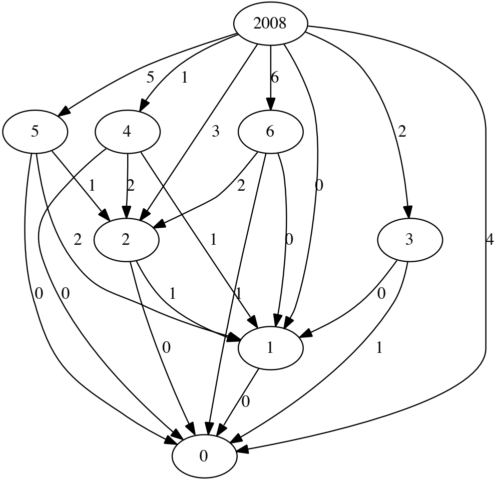
```
hfp :: Encoder T
hfp = compose (Iso hfp2nat nat2hfp) nat

*ISO> as hfp nat 42
H [H [],H [H [],H [H []]],H [H [H []],H []],
   H [H []],H [H [],H [H []],H [H [],H [H []]]]]
*ISO> as nat hfp it
42
```

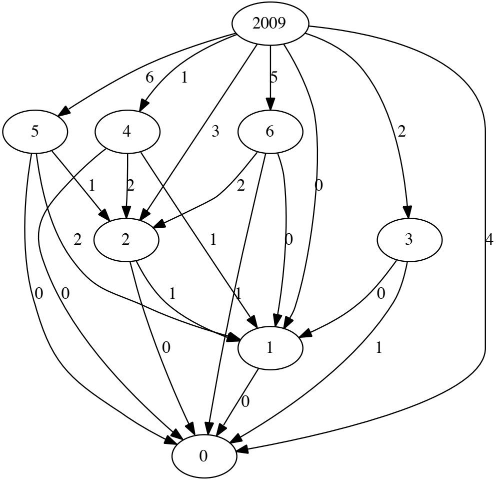
Interesting to note:

- sets: order unimportant
- permutations: content unimportant
- sequences - encodings use both order and content

Hereditarily finite permutation associated to **2008**



Hereditarily finite permutation associated to **2009**



HFS vs. HFP

It is interesting to see how “information density” of HFS and HFP compares. Intuitively that would answer the question: which is more efficient - codifying information as pure “content” or as pure “order”?

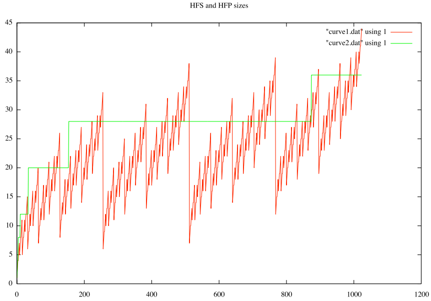


Figure: Comparison of curve1=HFS and curve2=HFP sizes up to 2^{10}

Pairing/Unpairing

pairing function: isomorphism $f : Nat \times Nat \rightarrow Nat$; inverse: *unpairing*

```
type Nat2 = (Nat,Nat)
*ISO> bitunpair 2008
  (60,26)
*ISO> bitpair (60,26)
  2008
-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
--   60:[0,    0,    1,    1,    1,    1]
--   26:[   0,    1,    0,    1,    1   ]
*ISO> as nat2 nat 2008
(60,26)
*ISO> as nat nat2 (60,26)
2008
```

Recursive unpairing graph

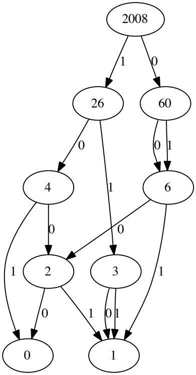


Figure: Graph obtained by recursive application of `bitunpair` for 2008

Encoding directed graphs

```
digraph2set ps = map bitpair ps
set2digraph ns = map bitunpair ns
```

The resulting Encoder is:

```
digraph :: Encoder [Nat2]
digraph = compose (Iso digraph2set set2digraph) set
```

working as follows:

```
*ISO> as digraph nat 2008
[(1,1),(2,0),(2,1),(3,1),(0,2),(1,2),(0,3)]
*ISO> as nat digraph it
2008
```

Digraph encoding

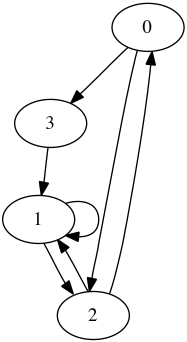


Figure: 2008 as a digraph

Encoding hypergraphs

Sets of non-empty sets:

```
set2hypergraph = map (nat2set . succ)
hypergraph2set = map (pred . set2nat)
```

The resulting Encoder is:

```
hypergraph :: Encoder [[Nat]]
hypergraph = compose (Iso hypergraph2set set2hypergraph) set
```

working as follows

```
*ISO> as hypergraph nat 2009
[[0],[2],[0,2],[0,1,2],[3],[0,3],[1,3],[0,1,3]]
*ISO> as nat hypergraph it
2009
```

Beyond data structures: a simple Turing equivalent functional language

Proposition

$\forall z \in \mathbb{N} - \{0\}$ the diophantic equation

$$2^x(2y+1) = z \tag{1}$$

has exactly one solution $x, y \in \mathbb{N}$.

hd, tl, cons, 0

```
type N = Integer

cons :: N→N→N
cons x y  = (2^x)*(2*y+1)

hd :: N→N
hd n | n>0 = if odd n then 0 else 1+hd  (n `div` 2)

tl :: N→N
tl n = n `div` 2^((hd n)+1)

*ISO> hd 2008 ⇒ 3
*ISO> tl 2008 ⇒ 125
*ISO> cons 3 125 ⇒ 2008
```

Revisiting “as”

```
as_fun_nat :: N→[N]
as_fun_nat 0 = []
as_fun_nat n = hd n : as_fun_nat (tl n)

as_nat_fun :: [N]→N
as_nat_fun [] = 0
as_nat_fun (x:xs) = cons x (as_nat_fun xs)

*ISO> as_fun_nat 2008
[3,0,1,0,0,0,0]
*ISO> as_nat_fun [3,0,1,0,0,0,0]
2008
```

Some classic functions

```
append 0 ys = ys
append xs ys = cons (hd xs) (append (tl xs) ys)

lst x = cons x 0
```

Higher order functions: fold and scan

```
nfoldl _ z 0    =  z
nfoldl f z xs =  nfoldl f (f z (hd xs)) (tl xs)

nfoldr f z 0     =  z
nfoldr f z xs =  f (hd xs) (nfoldr f z (tl xs))

nscanl _ q 0 = lst q
nscanl f q xs = cons q (nscanl f (f q (hd xs)) (tl xs))
```

Using foldl and scanl

```
*ISO> nfoldl (+) 0 8466
10
*ISO> as ns n (nscanl (+) 0 8466)
[0,1,3,6,10]
```

Applications: Random Generation

Combining `nth` with a random generator for *nat* provides free algorithms for random generation of complex objects of customizable size:

```
*ISO> random_gen set 11 999 3
[[0,2,5],[0,5,9],[0,1,5,6]]
*ISO> head (random_gen hfs 7 30 1)
H [H [],H [H [],H [H []]],H [H [H [H []]]]]
```

- a promising phenotype-genotype connection in Genetic Programming: isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side

Conclusion

- an embedded combinator language that shapeshifts datatypes at will using a small groupoid of isomorphisms
- lifting isomorphisms to hereditarily finite datatypes
- a practical tool to experiment with various universal encoding mechanisms

Literate Haskell program at
`http://logic.csci.unt.edu/tarau/research/2009/fISO.zip`

What else we can do?

Figure: Anywhere is possible: *shapeshifting* between datatypes

magic made easy - and also safe: we build bijective mappings between datatypes using a strongly typed language as a watchdog (Haskell)

PART II. Efficient Bijective Gödel Numberings for Term Algebras

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

Motivation

Gödel's incompleteness results (relying on Gödel numberings) had a huge impact on logic, foundations of mathematics, number theory, computer science and quite a few other fields.

- some infelicities of the original Gödel numberings:
  - encoding individual symbols rather then expression trees - using exponents of distinct prime numbers
  - computing the inverse is intractable (based on factoring)
  - encodings of syntactically ill-formed terms are possible

none of those shortcomings matter when focus is on computability only, but they do when one cares about computational complexity

Revisiting Gödel numberings - with “efficiency" in mind

- we design Gödel numberings with the following properties:
  - bijective
  - natural numbers always decode to syntactically valid terms
  - work in linear time in the bitsize of the representations
  - the bitsize of the encoding is within constant factor of the syntactic representation of the input
  - encodings on Term Algebras $\Rightarrow$ good for both code and data!

to be able to encode something as something else we need isomorphisms $\rightarrow$ bijections that transport structures

The Groupoid of Isomorphisms

```
data Iso a b = Iso (a→b) (b→a)
from (Iso f _) = f
to (Iso _ g) = g

compose :: Iso a b → Iso b c → Iso a c
compose (Iso f g) (Iso f' g') = Iso (f' . f) (g . g')

itself = Iso id id
invert (Iso f g) = Iso g f
```

Proposition

`Iso` *is a groupoid: when defined,* `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Connecting through a Hub

```
type N = Integer
isN n = n≥0
type Hub = [N]
```

We can now define an *Encoder* as an isomorphism connecting an object to *Root*

```
type Encoder a = Iso a Hub
```

This avoids having to provide $\frac{n*(n-1)}{2}$ isomorphisms!
The combinator *"as"* routes isomorphisms through two *Encoders*:

```
as :: Encoder a → Encoder b → b → a
as that this x = g x where
  Iso _ g = compose that (invert this)
```

An Example: Lists to/from Sets

```
*Goedel> as set nats [0,1,0,0,4]
[0,2,3,4,9]
*Goedel> as nats set [0,2,3,4,9]
[0,1,0,0,4]
```

How we do it? We can map lists of natural numbers to strictly increasing sequences of natural numbers representing sets!

```
   List               List'          Set
[0, 1,0,0,4] → [0, 2,1,1,5] → [0, 2,3,4,9]
```

$\Rightarrow$ Ackermann's encoding to $\mathbb{N}$: $2^0 + 2^2 + 2^3 + 2^4 + 2^9 = 541$

Morphing between Lists/Multisets/Sets

```
nats :: Encoder [N]
nats = itself

mset :: Encoder [N]
mset = compose (Iso as_nats_mset as_mset_nats) nats

as_mset_nats ns = tail (scanl (+) 0 ns)
as_nats_mset ms = zipWith (-) (ms) (0:ms)

set :: Encoder [N]
set = compose (Iso as_nats_set as_set_nats) nats

as_set_nats = (map pred) . as_mset_nats . (map succ)
as_nats_set = (map pred) . as_nats_mset . (map succ)
```

Uncovering the implicit list structure of a natural number

Proposition

$\forall z \in \mathbb{N} - \{0\}$ *the diophantic equation*

$$2^x(2y+1) = z \tag{2}$$

has exactly one solution $x, y \in \mathbb{N}$.

hd, tl, cons, 0

```
cons :: N→N→N
cons x y  = (2^x)*(2*y+1)

hd :: N→N
hd n | n>0 = if odd n then 0 else 1+hd  (n `div` 2)

tl :: N→N
tl n = n `div` 2^((hd n)+1)

*Goedel> hd 2008 ⇒ 3
*Goedel> tl 2008 ⇒ 125
*Goedel> cons 3 125 ⇒ 2008
```

Morphing between $\mathbb{N}$ and $[\mathbb{N}]$

```
as_nats_nat :: N→[N]
as_nats_nat 0 = []
as_nats_nat n = hd n : as_nats_nat (tl n)

as_nat_nats :: [N]→N
as_nat_nats [] = 0
as_nat_nats (x:xs) = cons x (as_nat_nats xs)

*Goedel> as_nats_nat 2008
[3,0,1,0,0,0,0]
*Goedel> as_nat_nats [3,0,1,0,0,0,0]
2008
```

A problem - exponential in the size of the input [$\mathbb{N}$]

```
nat1 :: Encoder N
nat1 = Iso as_nats_nat as_nat_nats

*Goedel> as nat1 nats [50,20,50]
531691198313966585279959557585082776
```

Pairing Functions as Encoders

Definition

An isomorphism $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is called a pairing function *and its inverse f^{-1} is called an* unpairing *function.*

Given the definitions:

```
unpair z = (hd (z+1), tl (z+1))
pair (x,y) = (cons x y)-1
```

Proposition

unpair $: \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ *is a bijection and pair* $=$ *unpair*$^{-1}$.

An encoder for tuples

```
to_tuple k n = map (from_base 2) (
    transpose (
      map (to_maxbits k) (
        to_base (2^k) n
      )
    )
  )
```

Simple: first bit to the first number, next bit to the next etc.

```
*Goedel> to_tuple 5 2012
[4,2,3,3,3]
```

An decoder for tuples

```
from_tuple ns = from_base (2^k) (
    map (from_base 2) (
      transpose (
        map (to_maxbits l) ns
      )
    )
  ) where
      k=genericLength ns
      l=max_bitcount ns
```

Just merging back the bits (but some padding is needed)!

```
*Goedel> from_tuple [4,2,3,3,3]
2012
```

Encoding with Tuples

1. split $n \in \mathbb{N}$ with unpair $n = 2^x(2y+1)-1$ giving (x,y)
2. use the first element x as the length of the tuple
3. split the second element y to a *tuple* with x elements

```
nat2ftuple 0 = []
nat2ftuple n = to_tuple (succ x) y where
  (x,y)=unpair (pred n)

ftuple2nat [] = 0
ftuple2nat ns = succ (pair (pred k,t)) where
  k=genericLength ns
  t=from_tuple ns
```

Encoding of lists proportional to the total bitsize of their elements

```
nat :: Encoder N
nat = Iso nat2ftuple ftuple2nat

*Goedel> as nats nat 2008
[3,2,3,1]
*Goedel> as nat nats it
2008
```

One can see that the first argument of the pairing function controls the length of the tuple while the second controls the bits defining the tuple.

A compact encoding of lists

Proposition

The encoder `nat` *works in space and time proportional to the bitsize of the largest element of the list multiplied by the length of the list.*

```
*Goedel> as nat nats [2009,2010,4000,0,5000,42]
485513619123942740473456O
*Goedel> as nats nat it
[2009,2010,4000,0,5000,42]

*Goedel> as nat1 nats [2009,2010,4000,0,5000,42]
1811020413277O6984...
....2 pages more ....
......53964009455616
```

Term Algebras

```
data Term var const =
   Var var |
   Fun const [Term var const]
   deriving (Eq,Ord,Show,Read)
```

From Terms to Natural Numbers

1. separate encodings of variable and function symbols i.e. map them, respectively, to even and odd numbers
2. to deal with function arguments, use the bijective encoding of sequences recursively

```
type NTerm = Term N N

nterm2code :: Term N N → N

nterm2code (Var i) = 2*i
nterm2code (Fun cName args) = code where
  cs=map nterm2code args
  fc=as nat nats (cName:cs)
  code = 2*fc-1
```

From Natural Numbers, back to Terms

1. recurse over the sequence associated to a natural number by the `as nats nat` combinator
2. associate variables to even numbers

```
code2nterm :: N → Term N N
code2nterm n | even n = Var (n `div` 2)
code2nterm n = Fun cName args where
  k = (n+1) `div` 2
  cName:cs = as nats nat k
  args = map code2nterm cs
```

The Encoder `nterm`

We can encapsulate our transformers as the Encoder:

```
nterm :: Encoder NTerm
nterm = compose (Iso nterm2code code2nterm) nat

*Goedel> as nat nterm (Fun 1 [Fun 0 [],Var 0])
55
*Goedel> as nterm nat 55
Fun 1 [Fun 0 [],Var 0]
```

Encoding strings with bijective base-k numbers

```
*Goedel> map (as (bijnat 3) nat) [0..12]
[[],[0],[1],[2],[0,0],[1,0],[2,0],[0,1],
        [1,1],[2,1],[0,2],[1,2],[2,2]]

*Goedel> as nat string "hello"
7073802
*Goedel> as string nat it
"hello"
```

More realistic terms - with strings as function names

```
*Goedel> as nat sterm (Fun "b" [Fun "a" [],Var 0])
2215

*Goedel> as sterm nat it
Fun "b" [Fun "a" [],Var 0]

*Goedel> as nat sterm (Fun "forall" [Var 0, Fun "f" [Var 0]])
38696270040102961756579399

*Goedel> as sterm nat it
Fun "forall" [Var 0,Fun "f" [Var 0]]
```

A view as bijective base-2 bitstrings

```
*Goedel> as bits nterm
     (Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1])
[0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,
 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

*Goedel> as nterm bits
  [0,0,0,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,
   0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Fun 0 [Fun 1 [Var 0,Fun 1 [Var 0]],Var 1]
```

Conclusion

- literate Haskell – a powerful tool for “experimental" theoretical computer science
- the original field for Gödel numberings is computability theory
- our Gödel numberings are“complexity aware" - possible uses in encodings relevant for complexity theory
  - encodings work in space and time proportional to the bitsize of the representations
  - natural numbers always decode to syntactically valid terms
- a possible more practical application: generate random terms - useful for QuickCheck-style testing
- also - natural numbers represent terms succinctly $\Rightarrow$ serialization of data and code, compression of terms sent over a network etc.

PART III. Pairing/Unpairing Functions and Boolean Evaluation

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

Outline

- by using pairing functions (bijections $N \times N \to N$) on natural number representations of truth tables, we derive an encoding for Ordered Binary Decision Trees (OBDTs)
- boolean evaluation of a OBDT mimics its structural conversion to a natural number through recursive application of a matching pairing function
- also: we derive *ranking* and *unranking* functions for OBDTs, generalize to arbitrary variable order and multi-terminal OBDTs
- literate Haskell program, code at `http://logic.csci.unt.edu/tarau/research/2009/fOBDT.hs`

Pairing functions

“pairing function”: a bijection $J : Nat \times Nat \rightarrow Nat$

$\texttt{K(J(x,y))} = \texttt{x,}$
$\texttt{L(J(x,y))} = \texttt{y}$
$\texttt{J(K(z), L(z))} = \texttt{z}$

examples:

- Cantor’s pairing function: geometrically inspired (100++ years ago - possibly also known to Cauchy - early 19-th century)
- the Pepis-Kalmar Pairing Function (1938):

$$f(x,y) = 2^x(2y+1) - 1 \tag{3}$$

a pairing/unpairing function based on boolean operations

```
type Nat = Integer
type Nat2 = (Nat,Nat)

bitpair ::  Nat2 → Nat
bitunpair :: Nat → Nat2

bitpair (x,y) = inflate x .|. inflate' y
bitunpair z = (deflate z, deflate' z)
```

inflate : abcde-> a0b0c0d0e

inflate': abcde-> 0a0b0c0d0e

inflate/deflate in terms of boolean operations

```
inflate 0 = 0
inflate n = (twice . twice . inflate . half) n .|. parity n

deflate 0 = 0
deflate n = (twice . deflate . half . half) n .|. parity n

deflate' = half . deflate . twice
inflate' = twice . inflate

half n = shiftR n 1 :: Nat
twice n = shiftL n 1 :: Nat
parity n = n .&. 1 :: Nat
```

bitpair/bitunpair: an example

the transformation of the bitlists – with bitstrings aligned:

```
*OBDT> bitunpair 2012
(62,26)

-- 2012:[0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1]
--   62:[0,    1,    1,    1,    1,    1]
--   26:[   0,    1,    0,    1,    1   ]
```

Note that we represent numbers with bits in reverse order.
Also, some simple algebraic properties:

```
bitpair (x,0) =       inflate x
bitpair (0,x) = 2 * (inflate x)
bitpair (x,x) = 3 * (inflate x)
```

Visualizing the pairing/unpairing functions

- Given that unpairing functions are bijections from $N \to N \times N$ they will progressively cover all points having natural number coordinates in the plan.
- Pairing can be seen as a function z=f(x,y), thus it can be displayed as a 3D surface.
- Recursive application – the unpairing tree can be represented as a DAG – by merging shared nodes.

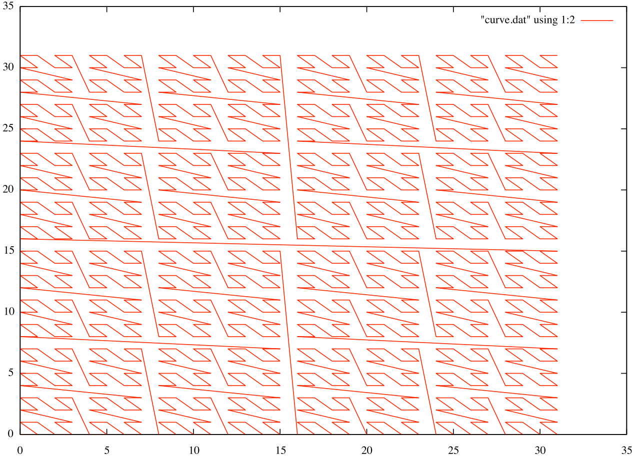


Figure: 2D curve connecting values of `bitunpair n` for $n \in [0..2^{10}-1]$

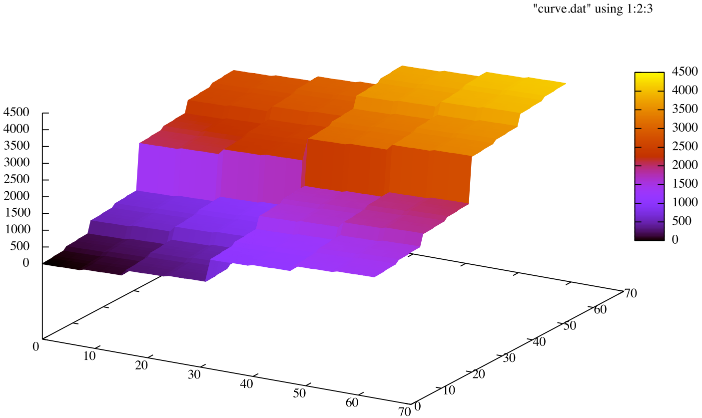


Figure: Values of bitpair x y with x,y in [0..63]

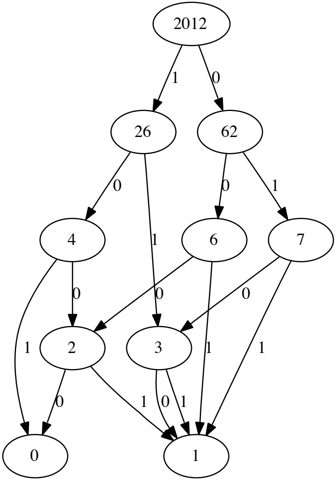


Figure: Graph obtained by recursive application of `bitunpair` for 2012

Unpairing Trees: seen as OBDTs

```
data OBDT a = OBDT a (BT a)
data BT a = B0 | B1 | D a (BT a) (BT a)

unfold_obdt :: Nat2 → OBDT Nat
unfold_obdt (n,tt) | tt < 2^2^n = OBDT n bt where
  bt = unfold_with bitunpair n tt

  unfold_with _ n 0 | n<1 = B0
  unfold_with _ n 1 | n<1 = B1
  unfold_with f n tt =
    D k (unfold_with f k tt1) (unfold_with f k tt2) where
      k=pred n
      (tt1,tt2)=f tt
```

Folding back Trees to Natural Numbers

```
fold_obdt :: OBDT Nat → Nat2
fold_obdt (OBDT n bt) = (n,fold_with bitpair bt) where
  fold_with rf B0 = 0
  fold_with rf B1 = 1
  fold_with rf (D _ l r) = rf (fold_with rf l,fold_with rf r)
```

This is a purely structural operation - no boolean evaluation involved!

```
*OBDT>unfold_obdt (3,42)
  OBDT 3 (D 2 (D 1 (D 0 B0 B0)
                   (D 0 B0 B0))
              (D 1 (D 0 B1 B1)
                   (D 0 B1 B0)))
*OBDT>fold_obdt it
  42
```

Truth tables as natural numbers

```
    x y z → f x y z
(0,[0,0,0])→0
(1,[0,0,1])→1
(2,[0,1,0])→0
(3,[0,1,1])→1
(4,[1,0,0])→0
(5,[1,0,1])→1
(6,[1,1,0])→1
(7,[1,1,1])→0
               ::
{1,3,5,6}:: 106 = 2^1 + 2^3 + 2^5 + 2^6 = 2 + 8 + 32 + 64
01010110 (right to left)
```

Computing all Values of a Boolean Function with Bitvector Operations (Knuth 2009 - Bitwise Tricks and Techniques)

Proposition

Let v_k be a variable for $0 \leq k < n$ where n is the number of distinct variables in a boolean expression. Then column k in the matrix representation of the inputs in the the truth table represents, as a bitstring, the natural number:

$$v_k = (2^{2^n} - 1)/(2^{2^{n-k-1}} + 1) \tag{4}$$

For instance, if $n = 2$, the formula computes $v_0 = 3 = [0,0,1,1]$ and $v_1 = 5 = [0,1,0,1]$.

we can express v_n with boolean operations + `bitpair`!

The function `vn`, working with arbitrary length bitstrings are used to evaluate the [0..n-1] *projection variables* v_k representing encodings of columns of a truth table, while `vm` maps the constant `1` to the bitstring of length 2^n, `111..1`:

```
vn 1 0 = 1
vn n q | q == pred n = bitpair (vn n 0,0)
vn n q  | q≥0 && q < n′ = bitpair (q′,q′) where
  n′ = pred n
  q′ = vn n′ q

vm n = vn (succ n) 0
```

OBDTs

- an ordered binary decision diagram (OBDT) is a rooted ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to `0` (left branch) and `1` (right branch).
- deriving a OBDT of a boolean function f: repeated Shannon expansion

$$f(x) = (\bar{x} \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \tag{5}$$

with a more familiar notation:

$$f(x) = \mathit{if}\ x\ \mathit{then}\ f[x \leftarrow 1]\ \mathit{else}\ f[x \leftarrow 0] \tag{6}$$

Boolean Evaluation of OBDTs

- OBDTs ⇒ ROBDDs by sharing nodes + dropping identical branches
- `fold_obdt` might give a different result as it computes different pairing operations!
- however, we obtain a truth table if we evaluate the OBDT tree as a boolean function – it would be nice if we could relate this to the original truth table from which we unfolded the OBDT!

```
eval_obdt (OBDT n bt) = eval_with_mask (vm n) n bt where
  eval_with_mask m _ B0 = 0
  eval_with_mask m _ B1 = m
  eval_with_mask m n (D x l r) = ite_ (vn n x)
    (eval_with_mask m n l)  (eval_with_mask m n r)

ite_ x t e = ((t `xor` e).&.x) `xor` e
```

The Equivalence of boolean evaluation and recursive pairing

SURPRISINGLY, it turns out that:

- boolean evaluation `eval_obdt` faithfully emulates `fold_obdt`
- and it also works on reduced OBDTs, ROBDDs, BDDs as they represent the same boolean formula

```
*OBDT> unfold_obdt (3,42)
OBDT 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
            (D 1 (D 0 B1 B1) (D 0 B1 B0)))
*OBDT> eval_obdt it
42
```

The Equivalence

Proposition

The complete binary tree of depth n, obtained by recursive applications of `bitunpair` *on a truth table computes an (unreduced) OBDT, that, when evaluated (reduced or not) returns the truth table, i.e.*

$$\texttt{fold\_obdt} \circ \texttt{unfold\_obdt} \equiv id \tag{7}$$

$$\texttt{eval\_obdt} \circ \texttt{unfold\_obdt} \equiv id \tag{8}$$

Ranking and Unranking of OBDTs

Ranking/unranking: bijection to/from *Nat*

- one more step is needed to extend the mapping between *OBDTs* with *N* variables to a bijective mapping from/to *Nat*:
- we will have to “shift toward infinity” the starting point of each new block of OBDTs in *Nat* as OBDTs of larger and larger sizes are enumerated
- we need to know by how much - so we compute the sum of the counts of boolean functions with up to *N* variables.

Ranking/unranking of OBDTs

```
bsum 0 = 0
bsum n | n>0 = bsum1 (n-1) where
  bsum1 0 = 2
  bsum1 n | n>0 = bsum1 (n-1)+ 2^2^n
```

```
*OBDT> map bsum [0..6]
  [0,2,6,22,278,65814,4295033110]
```

A060803 in the Online Encyclopedia of Integer Sequences

```
*OBDT> nat2bdd 42
OBDT 3 (D 2 (D 1 (D 0 B0 B1) (D 0 B1 B0))
            (D 1 (D 0 B0 B0) (D 0 B0 B0)))
*OBDT> bdd2nat it
42
```

Generalizations

Given a permutation of n variables represented as natural numbers in $[0..n-1]$ and a truth table $tt \in [0..2^{2^n}-1]$ we can define:

```
to_obdt vs tt | 0≤tt && tt ≤ m =
  OBDT n (to_obdt_mn vs tt m n) where
    n=genericLength vs
    m=vm n

to_obdt_mn []       0 _ _ = B0
to_obdt_mn []       _ _ _ = B1
to_obdt_mn (v:vs) tt m n = D v l r where
  cond=vn n v
  f0= (m `xor` cond) .&. tt
  f1= cond .&. tt
  l=to_obdt_mn vs f1 m n
```

Applications

- possible applications to (RO)BDDs: circuit synthesis/verification
- BDD minimization using our generalization to arbitrary variable order
- combinatorial enumeration and random generation of circuits
- succinct data representations derived from our OBDT encodings
- an interesting “mutation": use integers/bitstrings as genotypes, OBDTs as phenotypes in Genetic Algorithms

Conclusion

- NEW: the connection of pairing/unpairing functions and boolean evaluation of OBDTs
- synergy between concepts borrowed from *foundation of mathematics, combinatorics, boolean logic, circuits*
- Haskell as sandbox for experimental mathematics: type inference helps clarifying complex dependencies between concepts quite nicely - moving to a functional subset of Mathematica, after that, is routine.

PART IV. Hereditarily Finite Representations of Natural Numbers and Self-Delimiting Codes

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

Outline

$\Rightarrow$ Previous work: a framework to provide isomorphisms between fundamental data types (PPDP'2009, PPDP'2010, Calculemus'2009, Calculemus'2010)

- Gödel Numberings $\Rightarrow$ Ranking/Unranking bijections to/from $\mathbb{N}$
- Hereditarily Finite Functions (**HFF**): obtained by recursive application of a bijection $\mathbb{N} \to [\mathbb{N}]$
- as an application of the framework, we derive self-delimiting codes as isomorphic representations of HFF and parenthesis languages
- a quick look at encoding for S,K combinator trees and Goedel System **T** types

This is Arithmetically Destructured Functional Programming!

$\Rightarrow$ some destructuring is needed to reveal the structure ...

Figure: some destructuring is needed to reveal the structure ...

Uncovering the list structure “hiding" inside a natural number

```
type N = Integer

cons :: N→N→N
cons x y  = (2^x)*(2*y+1)

hd,tl :: N→N

hd n | n>0 = if odd n then 0 else 1+hd  (n `div` 2)

tl n = n `div` 2^((hd n)+1)

*SelfDelim> (hd 2012, tl 2012)
(2,251)
*SelfDelim> cons 2 251
2012
```

A bijection between finite functions/sequences and $\mathbb{N}$

```
nat2fun :: N→[N]
nat2fun 0 = []
nat2fun n = hd n : nat2fun (tl n)

fun2nat :: [N]→N
fun2nat [] = 0
fun2nat (x:xs) = cons x (fun2nat xs)
```

Proposition

`fun2nat` *is a bijection from finite sequences of natural numbers to natural numbers and* `nat2fun` *is its inverse.*

The Groupoid of Isomorphisms

```
data Iso a b = Iso (a→b) (b→a)
from (Iso f _) = f
to (Iso _ g) = g

compose :: Iso a b → Iso b c → Iso a c
compose (Iso f g) (Iso f′ g′) = Iso (f′ . f) (g . g′)

itself = Iso id id
invert (Iso f g) = Iso g f
```

Proposition

`Iso` *is a groupoid: when defined,* `compose` *is associative,* `itself` *is an identity element,* `invert` *computes the inverse of an isomorphism.*

Choosing a Hub

```
type Hub = [N]
```

We can now define an *Encoder* as an isomorphism connecting an object to *Hub*

```
type Encoder a = Iso a Hub
```

the combinators *with* and *as* provide an *embedded transformation language* for routing isomorphisms through two *Encoders*:

```
with :: Encoder a→Encoder b→Iso a b
with this that = compose this (invert that)

as :: Encoder a → Encoder b → b → a
as that this thing = to (with that this) thing
```

The bijection from $\mathbb{N}$ to $[\mathbb{N}]$ as an Encoder

We can define the `Encoder`

```
nat :: Encoder N
nat = Iso nat2fun fun2nat
```

working as follows

```
*SelfDelim> as fun nat 2012
[2,0,0,1,0,0,0,0]
*SelfDelim> as nat fun [2,0,0,1,0,0,0,0]
2012
```

Bijective base-2 natural numbers

Definition

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

- using a list notation for bitstrings we have:
  $0 = [], 1 = [0], 2 = [1], 3 = [0,0], 4 = [1,0], 5 = [0,1], 6 = [1,1]$
- a bijection between $\mathbb{N}$ and $\{0,1\}^*$
- no bit left behind :-)
- $\Rightarrow$ maximum information density for *undelimited* sequences

Mapping Natural Numbers to Bijective base-2 Bitstrings

```
bits :: Encoder [N]
bits = compose (Iso bits2nat nat2bits) nat

nat2bits = init . (to_base 2) . succ

bits2nat bs = pred (from_base 2 (bs ++ [1]))
```

```
*SelfDelim> as bits nat 2012
[1,0,1,1,1,0,1,1,1,1]

*SelfDelim> as nat bits it
2012
```

Generic unranking and ranking hylomorphisms

- The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*.
- The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.
- *unranking anamorphism* (*unfold* operation): generates an object from a simpler representation - for instance the seed for a random tree generator
- *ranking catamorphism* (a *fold* operation): associates to an object a simpler representation - for instance the sum of values of the leaves in a tree
- together they form a mixed transformation (*hylomorphism*)

Ranking/unranking hereditarily finite datatypes

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

The two sides of our hylomorphism are parameterized by two transformations `f` and `g` forming an isomorphism `Iso f g`:

```
unrank f n = H (unranks f (f n))
unranks f ns = map (unrank f) ns

rank g (H ts) = g (ranks g ts)
ranks g ts = map (rank g) ts
```

"structured recursion": propagate a simpler operation guided by the structure of the data type obtained as:

```
tsize = rank (λx→1 + (sum x))
```

Extending isomorphisms with hylomorphisms

We can now combine an anamorphism+catamorphism pair into an isomorphism `hylo` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

```
hylo :: Iso b [b] → Iso T b
hylo (Iso f g) = Iso (rank g) (unrank f)
```

Encoding Hereditarily Finite Functions

```
hff :: Encoder T
hff = compose (hylo nat) nat

*SelfDelim> as hff nat 2012
H [H [H [H []]],H [],H [],H [H []],H [],H [],H [],H []]

*SelfDelim> as nat hff it
2012
```

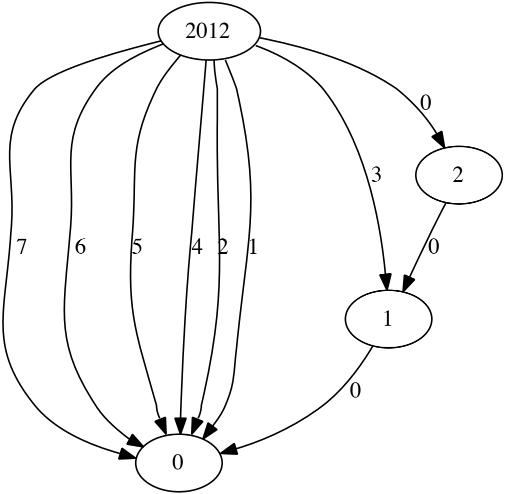


Figure: 2012 as a HFF

Self-Delimiting Codes

- a precise estimate of the actual size of various bitstring representations requires also counting the overhead for “delimiting” their components as this would model accurately the actual effort to transmit them over a channel or combine them in composite data structures
- an asymptotically optimal mechanism for this is the use of a *universal self-delimiting code* for instance, the *Elias omega code*
- To implement it, the encoder proceeds by recursively encoding the length of the string, the length of the length of the string etc.

Elias Omega Code

```
elias :: Encoder [N]
elias = compose (Iso (fst . from_elias) to_elias) nat
```

working as follows:

```
*SelfDelim> as elias nat 2012
[1,1,1,0,1,0,1,1,1,1,1,0,1,1,1,0,1,0]
*SelfDelim> as nat elias it
2012
```

Parenthesis Language Encodings

```
hff_pars :: Encoder [N]
hff_pars = compose (Iso f g) hff where
  f=parse_pars 0 1
  g=collect_pars 0 1

hff_pars' :: Encoder String
hff_pars' = compose (Iso f g) hff where
    f=parse_pars '(' ')'
    g=collect_pars '(' ')'

*SelfDelim> as hff_pars' nat 2012
"(((()))()()(())()()()())"
*SelfDelim> as nat hff_pars' it
2012
```

Parenthesis Language Encoding of Hereditarily Finite Types as a Self-Delimiting code

Proposition

The `hff_pars` *encoding is a self-delimiting code.*
If `n` *is a natural number, then* `hd n` *equals the code of the first parenthesized subexpression of the code of n and* `tl n` *equals the code of the expression obtained by removing it from the code for* `n`*, both of which represent self-delimiting codes.*

Recursive self-delimiting

A“fractal like" property:

```
*SelfDelim> as hff_pars nat 2012
[0, 0,0,0,1,1,1, 0,1,0,1,0,0,1,1,0,1,0,1,0,1,0,1, 1]
    ^^^ hd ^^^^
*SelfDelim> as hff_pars nat (hd 2012)
[0,0,0,1,1,1]  -- i.e. 2

*SelfDelim> as hff_pars nat 2
[0, 0,0,1,1, 1] -- i.e. 1
    ^^hd^^
*SelfDelim> as hff_pars nat (hd 1)
[0,1] -- i.e. 0
```

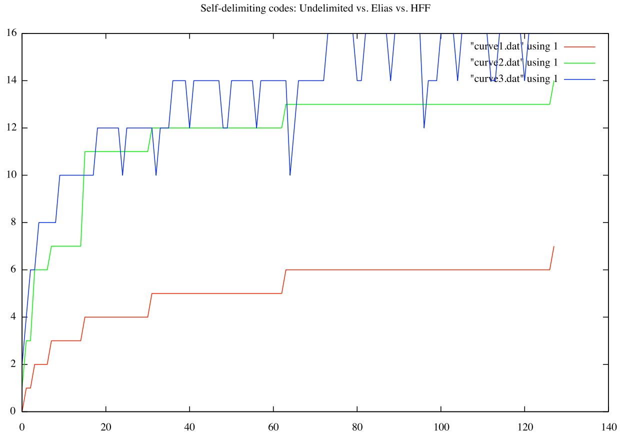


Figure: Code sizes up to 2^7: red=Undelimited, yellow=Elias, blue=hff_pars

Comparing Codes

The downward spikes in the blue upper curve shows the small regions where the balanced parenthesis HFF representation temporarily wins over Elias code.

- self-delimiting is not free - extra bits
- recursive self-delimiting is less compact than optimal one-level delimiting (Elias code), but not always
- recursive self-delimiting can be more compact for combinations of (a few) powers of 2 - sparseness
- an application: variants of recursive self-delimiting can be used to encode succinctly multi-level structured data - for instance XML files

Kraft's inequality for recursive self-delimiting code

```
kraft_sum m = sum (map kraft_term [0..m-1])

kraft_term n = 1 / (2 ** l) where l = parsize n

parsize = genericLength . (as hff_pars nat)

kraft_check m = kraft_sum m ≤ 1
```

```
*SelfDelim> map kraft_sum [10,100,1000,10000,100000,
                            200000,500000]
[0.3642,0.3829,0.3903,0.3939,0.3961,0.3967,0.3972]
```

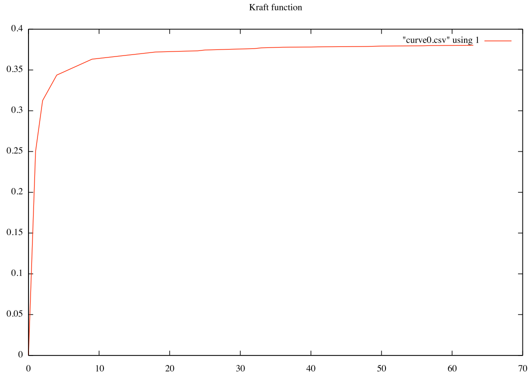


Figure: Kraft sum for balanced parenthesis codes

Self-Delimiting Codes for S,K Combinator Expressions

```
data Combs = K|S|A Combs Combs deriving (Eq,Read,Show)

encodeSK K=0
encodeSK S=1
encodeSK (A x y) = cons (encodeSK x) (encodeSK y)

decodeSK 0 = K
decodeSK 1 = S
decodeSK n = A (decodeSK (hd n)) (decodeSK (tl n))

*SelfDelim> map decodeSK [0..7]
[K, S, A S K, A K S, A (A S K) K,
   A K (A S K), A S S, A K (A K S)]
*SelfDelim> map encodeSK it
[0,1,2,3,4,5,6,7]
```

Deriving an Encoder for S,K trees

```
skTree :: Encoder Combs
skTree = compose (Iso encodeSK decodeSK) nat
```

the encoding of the `I=S K K` combinator

```
iComb = A (A S K) K
```

```
*SelfDelim> as nat skTree iComb
4
*SelfDelim> as hff_pars skTree iComb
[0,0,0,0,1,1,1,1]
*SelfDelim> as hff_pars' skTree iComb
"(((())))"
```

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

- Gödel System **T** types: a minimalist ancestor of modern type systems
- Binary trees are members of the *Catalan family* $\Rightarrow$ isomorphic with hereditarily finite functions and parenthesis languages
- types and arithmetic operations on natural numbers - buy one, get one free :-)

```haskell
infixr 5 :->
data G = E|G :-> G deriving (Eq, Read, Show)
```

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

```
s E = E:->E
s (E:->y) = s x:->y' where x:->y' = s y
s (x:->y) = E:->(p x:->y)

p (E:->E) = E
p (E:->(x:->y)) = s x:->y
p (x:->y) = E:->p (p x:->y)
```

An interesting consequence:

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

Defining the System **T** Recursor

```
rec :: (G → G → G) → G → G → G

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

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

recAdd = itr s

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

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

Arithmetic Operations with System **T** Types

```
*SelfDelim> [s E, s (s E), s (s (s E)), s (s (s (s E)))]
[E :-> E, (E :-> E) :-> E, E :-> (E :-> E),((E :-> E) :-> E) :-> E]

*SelfDelim> recAdd (s (s (s E))) (s (s (s E)))
(E :-> E) :-> (E :-> E)

*SelfDelim> recMul (s (s (s E))) (s (s (s E)))
E :-> (((E :-> E) :-> E) :-> E)

*SelfDelim> recPow (s (s E)) (s (s (s E)))
(E :-> (E :-> E)) :-> E
```

Conclusion

- we have shown how some interesting encodings can be derived from isomorphisms between fundamental data types
- work in progress: a framework providing a uniform construction mechanism for key concepts of finite mathematics: finite functions, sets, trees, graphs, digraphs, DAGs etc.
- future work: plans for connecting this framework to Joyal's combinatorial species
- the code shown here is at: `http://logic.cse.unt.edu/tarau/research/2010/selfdelim.hs`.

PART V. A Unified Formal Description of Arithmetic and Set Theoretical Data Types with Type Classes

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CSCE 6933 Topics in Computational Mathematics, Fall 2010

Motivation

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

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

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

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

Outline

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

Bijective base-2 natural numbers

Definition

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

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

Arithmetic operations using bijective base-2 numbers

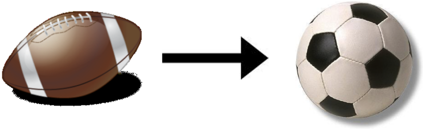


Figure: A thought on reinventing the wheel: some wheels are just better :-)

5 primitive BitStack / bijective base 2 operations

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

withO xs = O xs           -- op 2

withI xs = I xs           -- op 3

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

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

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

Emulating Peano Arithmetic with Bijective Base-2 Arithmetic

```
zero = empty
one = withO empty

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

Proposition

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

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

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

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

Derived Polymath Operations

```
e_ :: n→Bool
e_ x = x==e

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

u :: n
u = o e

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

Successor s and predecessor p functions

```
s :: n→n

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

p :: n→n

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

Generic Inductive Proofs of Program Properties

Proposition

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

- The inductive proof of this property uses the definitions directly.
- Clearly, p (s e) = p u = e (using the first pattern in s and p).
- Assume p (s x) = x.
  - Then p (s (o x)) = p (i x) = o x.
  - Also p (s (i x)) = p (o (s x)) = i (p (s x)) = i x.
- This proves $\forall\ x\ p(s\ x) = x$.

The induction on the second part of the proposition is similar.
Likely to be easy to implement in Coq with the (new) type classes.

A polymorphic converter between Polymath instances

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

```
view :: (Polymath a,Polymath b)⇒a→b

view x | e_ x = e
view x | o_ x = o (view (r x))
view x | i_ x = i (view (r x))

views xs = map view xs
```

The reference instance: Peano arithmetic

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

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

instance Polymath Peano where
  e = Zero

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

Instance Peano (continued)

```
o x = Succ (o' x) where
  o' Zero = Zero
  o' (Succ x) = Succ (Succ (o' x))

i x = Succ (o x)

r (Succ Zero) = Zero
r (Succ (Succ Zero)) = Zero
r (Succ (Succ x)) = Succ (r x)
```

Representing Hereditarily Finite Sets (HFS)

Hereditarily finite sets are built inductively from the empty set by adding finite unions of existing sets at each stage. We represent them as a rooted ordered tree datatype `S`

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

where the “empty leaf" `S []` denotes the empty set.

Definition (Ackermann mapping)

Objects of type `S` *are subject to the constraint that the function* `f` *associating a natural number to a hereditarily finite set x of type* `S`*, given by the formula*

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

is a bijection.

A HFS and its successor

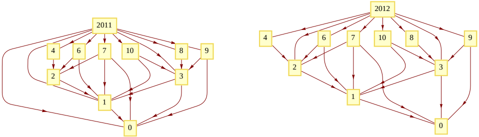


Figure: HFS: *2011* and *2012*

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

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

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

HFS as a Polymath instance

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

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

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

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

  i = s' . o

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

s’ and p’ are implementations of s and p

Proposition

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

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

More examples of HFS operations

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

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

Polymorphic Ordering: shared by sets, Peano numbers etc.

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

  lcmp x y | e_ x && e_ y = EQ
  lcmp x y | e_ x && not(e_ y) = LT
  lcmp x y | not(e_ x) && e_ y = GT
  lcmp x y = lcmp (r x) (r y)
```

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

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

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

```
lt,gt,eq :: n→n→Bool
lt x y = LT==cmp x y
gt x y = GT==cmp x y
eq x y = EQ==cmp x y

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

--- polySubtract:: n→n→n ....
```

Galois Connections with i, o, r

Definition

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

Definition

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

Galois Connections induced by Polymath primitives

Proposition

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

$$\forall a \, \forall b \; a \le b \Rightarrow o \; a \le o \; b \tag{9}$$

$$\forall a \, \forall b \; a \le b \Rightarrow i \; a \le i \; b \tag{10}$$

$$\forall a \, \forall b \; a \le b \Rightarrow r \; a \le r \; b \tag{11}$$

$$\forall a \, \forall b \; o \; a \le b \Leftrightarrow a \le r \; b \tag{12}$$

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

Derived properties holding for all Polymath instances

$$r \circ o \equiv r \circ i \equiv \lambda x.x \tag{13}$$

$$s \circ o \equiv i \equiv p \circ o \circ s \tag{14}$$

$$o \circ s \equiv s \circ i \equiv s \circ s \circ o \tag{15}$$

$$o \equiv p \circ i \equiv s \circ i \circ p \tag{16}$$

$$p \circ s \equiv \lambda x.x \tag{17}$$

$$\forall x((x \neq e) \Rightarrow s(p\ x) \equiv x) \tag{18}$$

Set Operations

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

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

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

Examples

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

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

Powerset, set membership, augmentSet

```
powset :: n→n
powset x = as_nat_set
  (map as_nat_set (subsets (as_set_nat x))) where
    subsets [] = [[]]
    subsets (x:xs) = [zs|ys←subsets xs,zs←[ys,(x:ys)]]

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

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

Ordinals

- The n-th *von Neumann ordinal* is the set $\{0, 1, \ldots, n-1\}$
- used to emulate natural numbers in finite set theory.
- It is implemented by the function `nthOrdinal`:

```
nthOrdinal :: n→n
nthOrdinal x | e_ x = e
nthOrdinal n = augmentSet (nthOrdinal (p n))
```

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

A practical outcome: representing some very large numbers

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

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

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

Deriving Digraphs, DAGs, Undirected Graphs

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

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

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

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

instance Polymath T where
  e = T

  o_ (T :-> x ) = True
  o_ _          = False

  o x = T :-> x
  i = s . o

  r (T:->y) = y
  r (x:->y) = p (p x:->y)
```

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

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

p (T:->T) = T
p (T:->(x:->y)) = s x:->y
p (x:->y) = T:->p (p x:->y)
```

An interesting consequence:

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

Examples for System **T** arithmetics

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

Defining the System **T** Recursor

```
rec :: (T → T → T) → T → T → T

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

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

recAdd = itr s

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

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

Conclusion

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

Future work/Open problems

General open problem:

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

(Manually) solved instances described so far:

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

Data types: a “parallel universes" view

- everything is just a *view* of the same fundamental entity: information
- being able to shift from one universe to another reveals interesting algorithms
- representational synergies can simplify the baroque ontology of computer science fields

foundational interconnectedness: Leibniz, in *La Monadologie, 1714*:

> Now this interconnectedness, or this accommodation of all created things to each, and of each to all the rest, means that each simple substance has relations to all the others, which it expresses.
> Consequently, it is a permanent living mirror of the universe.