Hereditarily Finite Representations of Natural Numbers and Self-Delimiting Codes

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
Research supported in part by NSF grant 1018172

MSFP'2010, Baltimore

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 $\mathbb{N} \rightarrow [\mathbb{N}]$
- 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
```

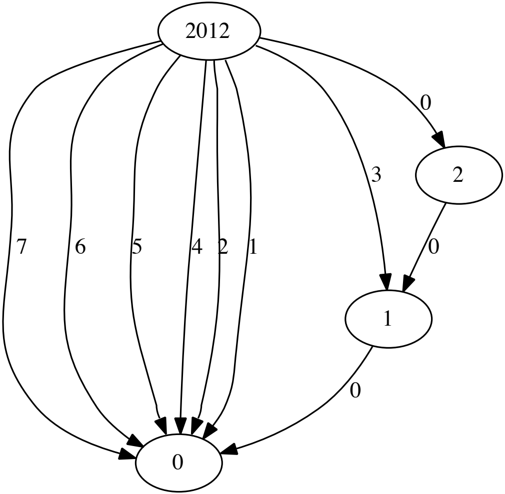

Figure: 2012 as a HFF

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
```

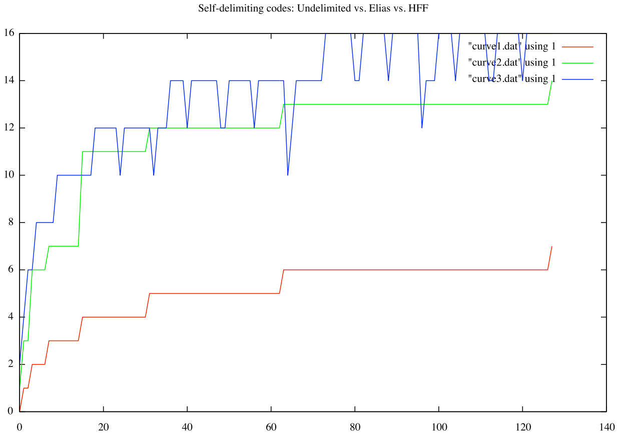

Figure: Code sizes up to 2^7: red=Undelimited, yellow=Elias, blue=hff_pars

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]
```

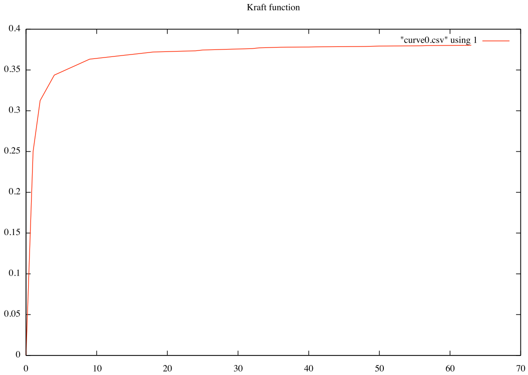

Figure: Kraft sum for balanced parenthesis codes

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 :-)

```
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 in the paper is at: `http://logic.cse.unt.edu/tarau/research/2010/selfdelim.hs`.