A Size-proportionate Bijective Encoding of Lambda Terms as Catalan Objects endowed with Arithmetic Operations

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PADL'2016

Research supported by NSF grant **1423324**

Outline

1. A compressed de Bruijn representation of lambda terms
2. Ranking and unranking: a Catalan embedding of compressed de Bruijn terms
3. Normalization with tree-based arithmetic operations
4. Combinatorial generation mechanisms for Catalan objects and lambda terms
5. Random generation of lambda terms
6. A discussion of related work
7. Conclusions

De Bruijn Indices

- *de Bruijn Indices* provide a name-free representation of lambda terms
- terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation
  - variables following lambda abstractions are omitted
  - their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* on the way up to the root of the term
- de Bruijn terms are *parameterized by the type* `a` *of the indices used by* `Vb` (because our "integers" will be a moving target!)

```
data B a = Vb a | Lb (B a) | Ab (B a) (B a) deriving (Eq,Show,Read)
```

- `Vb` marks variables, `Lb` lambda binders, `Ab` applications
- note: we start counting up from 0
- closed terms: every variable occurrence belongs to a binder
- open terms: otherwise

Compressing λ-terms

Figure: Random λ-terms have long necks: $\lambda(\lambda(\lambda(\lambda(\lambda(\ldots\ \ldots(\ldots$

Figure: But they can be compressed!

λ-term $\Rightarrow$ compressed λ-term

Compressed de Bruijn terms as labeled binary trees

- iterated λs (represented as a block of constructors `Lb` in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them
- $\Rightarrow$ it makes sense to represent that number more efficiently in the usual binary notation!
- in de Bruijn notation, blocks of λs can wrap either applications or variable occurrences represented as indices

```
data X a = Vx a a | Ax a (X a) (X a) deriving (Eq,Show,Read)
```

- $\Rightarrow$ we need only two constructors:
  - `Vx` indicating in a term `Vb k n` that we have `k` λs wrapped around the de Bruijn index `Vb n`
  - `Ax` indicating in a term `Vx k x y` that `k` λs are wrapped around the application `Ab x y`
- we call the terms built this way with the constructors `Vx` and `Ax` *compressed de Bruijn terms*

From de Bruijn to compressed

- `b2x`: by case analysis on `Vb, Ab, Lb`
- it counts the binders `Lb` as it descends toward the leaves of the tree

```
b2x :: (Cat a) => B a -> X a
b2x (Vb x) = Vx e x
b2x (Ab x y) = Ax e (b2x x) (b2x y)
b2x (Lb x) = f e x where
  f k (Ab x y) = Ax (s k) (b2x x) (b2x y)
  f k (Vb x) = Vx (s k) x
  f k (Lb x) = f (s k) x
```

- all arithmetic computations in this paper are performed in terms of instances of the type class `Cat` - our natural numbers are arbitrary members of the Catalan family of combinatorial objects!

From compressed to de Bruijn

The function `x2b` converts from the compressed to the usual de Bruijn representation. It reverses the effect of `b2x` by expanding the `k` in `V k n` and `A k x y` into `k Lb` binders (no binders when `k=0`). The function `iterLam` performs this operation in both cases, and the predecessor function `s'` computes the decrements at each step.

```
x2b :: (Cat a) => X a -> B a
x2b (Vx k x)   = iterLam k (Vb x)
x2b (Ax k x y) = iterLam k (Ab (x2b x) (x2b y))

iterLam :: Cat a => a -> B a -> B a
iterLam k x | e_ k = x
iterLam k x = iterLam (s' k) (Lb x)
```

Proposition

The functions `b2x` *and* `x2b`*, having as domains and range open terms, are inverses.*

Ranking and unranking: a Catalan embedding of compressed de Bruijn terms

- compressed de Bruijn terms are in fact labeled binary trees
- their labels are already expressed as instances of `Cat`
- $\Rightarrow$ we will derive an encoding of the compressed de Bruijn terms into objects of type `Cat`
- we expect the bijection to unlabeled binary trees to be quite simple
- the binary tree instance of type `Cat` will be size-proportionate with the encoded term
- the intuition behind the algorithm:
  - leaf nodes of the lambda term are encoded into leaves or small trees close to the leaves
  - application nodes are encoded into internal nodes of the binary tree

Ranking compressed de Bruijn terms

```
x2t :: Cat a => X a -> a

x2t (Vx k n) | e_ k && e_ n = n
x2t (Vx k n) = c (s' (s' (c (n,k))),e)
x2t (Ax k a b) = c (k,q) where q = c (x2t a,x2t b)
```

- leaves `Vx k x` are encoded either as
  - empty leaves of the binary tree `e`
  - or as subtrees with the right branch an empty leaf `e`
- to ensure the encoding is bijective, we will need to decrement the result of the constructor `c` twice in the second rule, with the predecessor function `s'`
- we ensure that this case leaves no gaps in the range of the function `x2t`
- for application nodes `Ax k a b` we recurse on nodes `a` and `b`
- we put the branches together with the constructor `c`
- when `c=C` (binary trees) or `c=M` (multiway trees), we obtain a tree of a size proportionate to the compressed de Bruijn term

Unranking compressed de Bruijn terms

```
t2x :: Cat a => a -> X a
t2x x | e_ x = Vx x x
t2x z = f y where
  (x,y) = c' z
  f y | e_ y = Vx k n where (n,k) = c' (s (s x))
  f y | c_ y = Ax x (t2x a) (t2x b) where (a,b) = c' y
```

- case analysis on the right branch of the binary tree will tell if it is a leaf node or an internal node of the lambda tree
- if it is a leaf, the increment in `x2t`, needed for bijectivity, is reversed
- we apply the successor function `s` twice before applying the deconstructor `c'`

Proposition

The functions `t2x` *and* `x2t`*, converting between open compressed de Bruijn terms and corresponding instances of* `Cat`*, are inverses.*

Normalization with tree-based arithmetic operations

- we will adapt here an evaluation mechanism for the (Turing-complete) language of closed lambda terms, called *normal order reduction*
- a mapping between de Bruijn terms and *a new data type that mimics standard lambda terms, except for representing binders as functions in the underlying implementation language*
- our transformation will be used both ways to evaluate and then return the result as a de Bruijn term
- we are inspired by a "folklore" technique widely used in FP - sometime called HOAS (higher order abstract syntax)
- however, we will organize all the computations to happen in terms of our `Cat`-based arithmetic

Representing lambdas as Haskell functions

```
data H a = Vh a | Lh (H a -> H a) | Ah (H a) (H a)
```

- the data type `H` represents leaves `Vh` of the lambda tree and applications `Ah`
- however, lambda binders `Lh`, meant to be substituted with terms during β-reduction steps, are represented as functions from the domain `H` to itself

Normalization

```
nf :: H a -> H a
nf (Vh a) = Vh a
nf (Lh f)   = Lh (nf . f)
nf (Ah f a) = h (nf f) (nf a) where
  h :: H a -> H a -> H a
  h (Lh g) x = g x
  h g x = Ah g x
```

- `nf` implements normalization of a term of type `H`, derived from a closed de Bruijn term
- at each `Lh f` step, `nf` traverses it and it is composed with `f`
- at each `Ah f a` step, if the left branch is a lambda, it is applied to the reduced form of the right branch as implemented by the helper function `h`
- otherwise, the application node is left unchanged
- the result of implementing lambdas as functions is that we not only borrow substitutions from the underlying Haskell system but also the underlying (normal) order of evaluation

Conversion to/from closed de Bruijn terms

```
h2b :: Cat a ⇒ H a -> B a
h2b t = h e t where
  h d  (Lh f)  = Lb  (h d' (f (Vh d'))) where d' = s d
  h d  (Ah a b) = Ab (h d a) (h d b)
  h d  (Vh d') = Vb (sub d d')
```

all computations are performed with type class `Cat`

```
b2h :: Cat a ⇒ B a -> H a
b2h t = h t [] where
  h :: Cat a ⇒  B a  -> [H a] -> H a
  h (Lb a) xs  = Lh (\x -> h a (x:xs))
  h (Ab a b) xs = Ah (h a xs) (h b xs)
  h (Vb i) xs  = at i xs
```

example: h2b inverts b2h (but "exotic" terms might exist on the H side)

```
> (h2b . b2h) (Lb (Lb (Lb (Ab (Ab (Vb 0) (Vb 2)) (Vb 1)))))
Lb (Lb (Lb (Ab (Ab (Vb 0) (Vb 2)) (Vb 1))))
```

Lending the reducer between representations

We obtain a normal order reducer for de Bruijn terms by wrapping up `nf` with the transformers `b2h` and `h2b`.

```
evalB :: (Cat a) => B a -> B a
evalB x  | isClosedB x = (h2b .nf . b2h) x
evalB _ = Vb e
```

We can then lend the evaluator also to compressed de Bruijn terms.

```
evalX :: (Cat a) => X a -> X a
evalX x = (b2x . evalB . x2b) x
```

Example:
reduction to the identity $I = \lambda x_0.x_0$ of
$SKK = ((\lambda x_0.\lambda x_1.\lambda x_2.((x_0\ x_2)\ (x_1\ x_2)))\ \lambda x_3.\lambda x_4.x_3)\ \lambda x_5.\lambda x_6.x_5)$ in compressed de Bruijn notation

```
> evalX (Ax 0 (Ax 0 (Ax 3 (Ax 0 (Vx 0 2) (Vx 0 0))
                          (Ax 0 (Vx 0 1) (Vx 0 0))) (Vx 2 1)) (Vx 2 1))
Vx 1 0
```

Catalan objects as lambda terms

Given the bijection between instances of the Catalan family, we can go one step further and extend the evaluator to binary trees and ordinary natural numbers, seen as lambda terms.

```
evalT :: T->T
evalT = x2t . evalX . t2x

evalN :: N->N
evalN = x2t . evalX . t2x
```

Evaluation of binary trees and natural numbers seen as lambda terms.

```
> evalT (C (C E (C E E)) (C (C E E) E))
C (C (C E E) E) E
> filter (>0) (map evalN [0..31])
[1,4,8,1,11,1,15,16,15,20,23,15,28,31]
```

As evaluation happens in a Turing-complete language, these functions are not total. For instance, `evalN 318`, corresponding to the lambda term $\omega = (\lambda x.(x\ x))(\lambda x.(x\ x))$, is non-terminating.

Generation of Catalan objects and lambda terms

- given the size-proportionate bijection between open lambda terms and Catalan objects, we can use generators for the later to generate the former
- a generator for Catalan objects:

```
genCat :: Cat t => t -> [t]
genCat n | e_ n = [n]
genCat n | c_ n = [ c (x,y) |
  k←nums (s' n), x←genCat k, y←genCat (s' (sub n k))]

> mapM_ print (genCat (t 3))
C E (C E (C E E))
C E (C (C E E) E)
C (C E E) (C E E)
C (C E (C E E)) E
C (C (C E E) E) E

> genCat 3
[5,6,4,7,15]
```

A generator for open lambda terms

```
genCatX :: Cat a => a -> [X a]
genCatX  = filter isClosedX . map t2x . genCat

genCatB :: Cat a => a -> [B a]
genCatB  = filter isClosedB . map t2b . genCat
```

generation of closed compressed de Bruijn terms decoded from binary trees with 3 internal nodes

```
> mapM_ print (genCatX 4)
Ax 0 (Vx 1 0) (Vx 1 0)
Ax 1 (Vx 0 0) (Vx 1 0)
Ax 1 (Vx 1 0) (Vx 0 0)
Ax 2 (Vx 0 0) (Vx 0 0)
Ax 3 (Vx 0 0) (Vx 0 0)
Vx 3 0
Vx 4 0
Vx 8 0
```

Generation of lambda terms via unranking

- an unranking algorithm is usable for generation of large (random) open terms
- generating open terms in compressed de Bruijn form
- by iterating over an initial segment of $\mathbb{N}$ with the function `t2x`

```
genOpenX :: Cat a => a -> [X a]
genOpenX l = map t2x (nums l)
```

- `genClosedX` generates closed terms by filtering the results of `genOpenX` with the predicate `isClosedX`

```
genClosedX l = filter isClosedX (genOpenX l)
```

- given the asymptotically low density of closed terms in the set of open ones this does not scale up very well

Random generation of lambda terms

- as the ranking bijection of the compressed de Bruijn lambda terms maps them to Catalan objects, we can use unranking of uniformly generated random binary trees to generate random terms
- generating random binary trees
- we rely on the Haskell library `Math.Combinat.Trees` to generate binary trees uniformly using a variant of Rémy's algorithm
- we use Haskell's built-in random generator from package `System.Random`
- this allows generation of random lambda terms corresponding to super-exponentially sized numbers of type `N`, but size-proportionate when natural numbers are represented by the binary trees of type `T`

Generating random Catalan objects

```
ranCat :: (Cat t, RandomGen g) => (N ->t) -> Int -> g -> (t, g)
ranCat tf size g = (bt2c bt,g') where
  (bt,g') = randomBinaryTree size g
  bt2c (Leaf ()) = tf 0
  bt2c (Branch l r) = c (bt2c l,bt2c r)
```

- `ranCat` is parametrized by the function `tf` that picks a type for a leaf among the instances of `Cat`, to be propagated as the type of tree, as well as the size of the tree and the random generator `g`
- `ranCat1` allows getting a random tree of a given size and type, by giving a seed that initializes the random generator `g`

```
ranCat1 tf size seed = fst (ranCat tf size (mkStdGen seed))
```

Generating random compressed de Bruijn terms

- we use the bijection `t2x` from Catalan objects to open compressed de Bruijn trees
- it is parameterized by the function `tf` that picks the type of the instance of `Cat` to be used
- `ranOpenX` generates random terms and `ranClosedX` filters the generated terms until a closed one is found

```
ranOpenX tf size g = (t2x r,g') where(r,g') = ranCat tf size g

ranClosedX tf size g =
  if isClosedX x then x else ranClosedX tf size g' where
    (a,g') = ranCat tf size g
    x = t2x a
```

- `ranOpen1X` and `ranClosed1X` use a `seed` to build a `RandomGen`

```
ranOpen1X tf size seed = t2x (ranCat1 tf size seed)
ranClosed1X tf size seed = ranClosedX tf size g where
  g = mkStdGen seed
```

Generation of some random lambda terms (including very large ones) in compressed de Bruijn form

```
> ranClosed1X n 3 9
Ax 1 (Vx 0 0) (Vx 0 0)

> ranClosed1X t 3 9
Ax (C E E) (Vx E E) (Vx E E)

> n (sizeX (ranClosed1X t 100 9))
96

>  n (sizeX (ranOpen1X t 50000 42))
50001
```

Related work

- combinatorics of lambda terms, including enumeration, random generation and asymptotic behavior has seen an increased interest recently partly motivated by applications to software testing
- ranking and unranking of lambda terms can be seen as a building block for bijective serialization of practical data types
- compressed de Bruijn terms are used in our CICM/Calculemus paper in combination with the generalized Cantor bijection between $\mathbb{N}^k \to \mathbb{N}$ to provide a bijective Gödel numbering scheme
- that encoding is size-proportional with numbers in the usual notation but the $\mathbb{N} \to \mathbb{N}^k$ side of this bijection is only computable using a binary search algorithm, so it is limited to relatively small open terms

Conclusions

- our computations for encoding, decoding, compressing and normalizing lambda terms have used a type class defining generic arithmetic operations on members of the Catalan family of combinatorial objects
- some interesting synergies have been triggered by this:
  - a size-proportionate bijective encoding of compressed De Bruijn terms
  - we have *radically* changed the representation of "natural numbers" to operate through a binary tree view of Catalan objects
  - this unusual target for ranking/unranking of lambda terms has facilitated (via random generation algorithm for binary trees) generation of (possibly very large) random (open) lambda terms
- our size-proportional encodings as arithmetic-endowed Catalan objects can be easily adapted to:
  - expression trees
  - recursive data types
  - directory structures
  - parse trees for programming and natural languages
  - phylogenetic trees