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, Denton, USA
paul.tarau@unt.edu

Abstract. We describe a size-proportionate bijection between lambda terms in a compressed de Bruijn notation and the Catalan family of combinatorial objects implemented as a Haskell type class, that has as instances binary trees and multiway-trees with empty leaves, as well as standard bitstring-represented natural numbers.

By building on previous work that defines arithmetic operations on instances of this family, we extend lambda calculus with efficient arithmetic operations.

At the same time, operations like normalization of lambda terms are made available to members of the Catalan family of combinatorial objects.

As a practical application to software testing we derive a mechanism for generating large random lambda terms from Rémy's algorithm for efficient generation of random binary trees.

Keywords: Lambda calculus · compressed de Bruijn terms · Tree-based numbering systems · Ranking and unranking of lambda terms · Normalization with higher order abstract syntax · Random generation of large lambda terms

1 Introduction

Bijective encodings of tree-like structures go back to Ackermann's bijection between natural numbers and hereditarily finite sets [1]. They are relatively easy to design if one does not care about one side of the bijection exponentially exploding in size, as it is the case, for instance, with Ackermann's bijection.

With significant effort, such size-proportionate bijections between term algebras and the set of natural numbers represented with the usual binary notation are defined in [21], using ranking of balanced parentheses languages and a generalization of Cantor's pairing function [5,16] to tuples. However, the binary search and complex computations involved in the ranking algorithms limit the encoding described in [21] to relatively small terms and numbers not larger than about 2000 bits.

M. Gavanelli and J. Reppy (Eds.): PADL 2016, LNCS 9585, pp. 99–116, 2016.
DOI: 10.1007/978-3-319-28228-2_7

A more revolutionary approach has been sketched out in [23]. Instead of trying to adjust the bijective Gödel numbering scheme to be size-proportionate as a bijection to bitstring-represented numbers, [23] replaces its target: natural numbers are represented as binary trees. This paper generalizes that approach to an arbitrary member of the Catalan family of combinatorial objects [20], on which the usual arithmetic operations are defined. At the same time, it lifts a limitation on the size-proportionate encoding of [23] where that property is lost unless de Bruijn indices fit in a fixed size word. The generalization of binary-tree arithmetic to Catalan objects, on which we rely in this paper, is described extensively in the unpublished arxiv draft [22], a subset of which is covered in [25].

Following [22], a Haskell type class will be used to abstract away the number representation. This has the benefit of having arithmetic operations implemented by any instance of the Catalan family of combinatorial objects. It will also enable trying out our algorithms on the usual "human friendly" natural numbers, which can be seen, via a bijective transformation, as such an instance.

The arithmetic operations performed with the Catalan family based numbering system can work with numbers comparable in size with Knuth's "arrow-up" notation. These computations have a worst case and average case complexity that is comparable with the traditional binary numbers, while their best case complexity outperforms binary numbers by an arbitrary tower of exponents factor.

More importantly, encodings of lambda terms, that can be seen as a tree-to-tree transformation, are naturally size-proportionate.

Our bijective encoding to tree-based number systems will provide the means to derive an algorithm for the generation of random lambda terms from well-known random generation algorithms for binary trees (Rémy's algorithm). Random lambda terms (and in particular, the very large ones our encoding enables) can be useful for testing tools where they play the role of an intermediate language, like compilers for functional languages and proof assistants.

By adding a normal order reducer of our lambda terms we provide a uniform representation for computations with lambda terms, and efficient arithmetic operations - two essential building blocks of modern functional languages.

Together, these have applications to implementation of domain specific languages, compiler stages and proof assistants relying on lambda terms as their intermediate language.

The paper is organized as follows. Section 2 introduces the compressed de Bruijn terms and bijective transformations from them to standard lambda terms. Section 3 describes mappings from lambda terms to Catalan families of combinatorial objects. These mappings lead to size-proportionate ranking and unranking algorithms for lambda terms. Section 4 gives an algorithm for normal order reduction of lambda terms that also extends to their Catalan encodings. Section 5 relates combinatorial generation of Catalan objects and that of lambda terms. Section 6 introduces algorithms for generation of random lambda terms. Section 7 discusses related work. Section 8 concludes the paper.

The paper is organized as literate Haskell program. The code in the paper is available at http://www.cse.unt.edu/~tarau/research/2015/XDB.hs, tested with GHC 7.10.2. It defines a module that includes code from [25] (file GCcat.hs, a superset of which is available from the arXiv draft [22]), which defines a type class for arithmetic operations with Catalan objects, generically. It also includes Haskell library packages needed for the generation of random binary trees.

```
module XDB where
import GCat
import System.Random
import Math.Combinat.Trees
```

To achieve a size-proportionate bijective Gödel numbering scheme, all our arithmetic computations will be performed with members of the type class `Cat` which provides a generic implementation in terms of members of the Catalan family of combinatorial objects [22], in particular binary or multiway trees with empty leaves.

2 A Compressed Representation of de Bruijn Terms

We will summarize here a compressed representation for lambda terms in de Bruijn notation introduced as Prolog program in [24], that will facilitate defining a bijection to tree-represented natural numbers.

2.1 De Bruijn Indices

De Bruijn indices [4,13] provide a *name-free* representation of lambda terms. All terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation. Variables following lambda abstractions are omitted and their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* is found on the way up to the root of the term.

We represent them using the constructor `Ab` for application, `Lb` for lambda abstractions (that we will call shortly *binders*) and `Vb` for marking the integers corresponding to the de Bruijn indices. This gives the Haskell data type `B a` as the definition of the de Bruijn terms *parameterized by the type* `a` *of the indices used by* `Vb`.

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

For instance, when the parameter `a` is specialized to ordinary integers, the `S` combinator $\lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z)$ is represented as `Lb (Lb (Lb (Ab (Ab (Vb 2) (Vb 0)) (Ab (Vb 1) (Vb 0)))))`, corresponding to the fact that `Vb 2` is bound by the outermost lambda (three steps away, counting from `0`) and the occurrences of `Vb 0` are bound each by the closest lambda on the way to the root, represented by the third constructor `Lb`.

2.2 Compressed de Bruijn Terms

Iterated lambdas (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. So it makes sense to represent that number in a more efficient numbering system. Note that in de Bruijn notation blocks of lambdas can wrap either applications or variable occurrences represented as indices. This suggests using just two constructors: `Vx` indicating in a term `Vx k n` that we have `k` lambdas wrapped around the de Bruijn index `Vb n` and `Ax`, indicating in a term `Ax k x y` that `k` lambdas 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*. They are specified by The Haskell data type `X`.

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

For instance, the `S` combinator `Lb (Lb (Lb (Ab (Ab (Vb 2) (Vb 0)) (Ab (Vb 1) (Vb 0)))))` in de Bruijn notation, will be represented as `Ax 3 (Ax 0 (Vx 0 2) (Vx 0 0)) (Ax 0 (Vx 0 1) (Vx 0 0))`, with the outermost constructor `Ax` encoding the three `Lb` binders and `k=0` elsewhere indicating the presence of no lambda binder in (front of) applications `Ax k` or indices `Vx k`. Note also that lambda binders counted by `k` in a leaf term `Vx k n` can bind at most one variable as no application splits the tree below them.

Open and Closed Terms. Lambda terms might contain *free variables* not associated to any binders. Such terms a called *open*. Any syntactically well formed term of types `B` and `X` is an open term. A *closed* term is such that each variable occurrence is associated to a binder. Closed terms can be easily identified by ensuring that the lambda binders on a given path from root outnumber the de Bruijn index of a variable occurrence ending the path.

To facilitate size-proportionate encodings of lambda terms, arithmetic operations in this paper will be performed in terms of tree instances of the type class `Cat`*, described in the companion paper* [25]*, a superset of which is available as* [22]. The function `isClosedX` checks that a compressed de Bruijn term is closed by trying to find a lambda binding every index on the way up to the root of the lambda tree. The addition operation `add` and successor function `s`, defined for instances of the type class `Cat` (see [22,25]), will be used to count binders and the comparison operation `cmp` (see [22]) will ensure that binders on the way down from the root outnumber index values at the leaves of the lambda tree.

```
isClosedX :: Cat a ⇒ X a → Bool
isClosedX t = f t e where
  f (Vx k n) d = LT==cmp n (add d k)
  f (Ax k x y) d = f x d' && f y d' where d' = add d k
```

Example 1. `isClosedX` *on the* `K` *combinator* $\lambda x_0.(\lambda x_1.\ x_0)$*, written* `(Vx 2 1)` *as a compressed de Bruijn term, and a similar small open term. Note the use of both* `Cat` *instances* `N` *and* `T` *parameterizing our (compressed) de Bruijn terms.*

```
*XDB> isClosedX (Vx 2 1)
True
*XDB> isClosedX (Vx 2 2)
False
*XDB> isClosedX (Vx (C E (C E E)) (C E E))
True
*XDB> isClosedX (Vx (C E (C E E)) (C E (C E E)))
False
```

2.3 Converting from de Bruijn to Compressed de Bruijn Terms

The function `b2x` converts from the usual de Bruijn representation to the compressed one. It proceeds by case analysis on `Vb, Ab, Lb` and counts the binders `Lb` as it descends toward the leaves of the tree. Its steps are controlled by the successor function `s` that increments the counts when crossing a binder.

```
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
```

2.4 Converting from Compressed de Bruijn to de Bruijn Terms

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

Example 2. *The conversion between types* `B` *and* `X` *of the combinator* $Y = \lambda x_0.(\lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$.

```
*XDB> b2x (Lb (Ab (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))
          (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))))
Ax 1 (Ax 1 (Vx 0 1) (Ax 0 (Vx 0 0) (Vx 0 0)))
```

```
          (Ax 1 (Vx 0 1) (Ax 0 (Vx 0 0) (Vx 0 0)))
*XDB> x2b it
Lb (Ab (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))
          (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0)))))
```

This bijection allows borrowing algorithms between the two representations. The function isClosedB tests if a term in de Bruijn notation is closed.

```
isClosedB :: Cat a ⇒ B a → Bool
isClosedB = isClosedX . b2x
```

3 Ranking and Unranking as a Catalan Embedding of Compressed de Bruijn Terms

We will derive an encoding of the compressed de Bruijn terms into objects of type Cat, such that the binary tree instance of type Cat is size-proportionate with the encoded term.

The intuition behind the algorithm is that we want leaf nodes of the lambda term to encode into leaves or small trees close to the leaves and application nodes to encode into internal nodes of the binary tree, as much as possible.

3.1 Ranking Compressed de Bruijn Terms

The function x2t implements such an encoding.

```
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)
```

Note that leaves Vx k x are encoded either as empty leaves of the binary tree or as subtrees with the right branch an empty leaf. 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' to 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 and then we put the branches together with the constructor c. When c=C or c=M, this results in a tree of a size proportionate to the compressed de Bruijn term.

3.2 Unranking Compressed de Bruijn Terms

The decoding function t2x reverses the steps of the encoder x2t.

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, in which case the increment in x2t, needed for bijectivity, is reversed by applying the successor function s twice before applying the deconstructor c'.

```
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
```

Proposition 2. *The functions* `t2x` *and* `x2t`*, converting between open compressed de Bruijn terms and corresponding instances of* `Cat`*, are inverses.*

Example 3. *The work of* `t2x` *and* `x2t` *on* `Cat` *instance* `N`*.*

```
*XDB> t2x 1234
Ax 0 (Vx 0 0) (Ax 1 (Vx 0 0) (Ax 0 (Vx 1 0) (Ax 1 (Vx 0 0) (Vx 0 0))))
*XDB> x2t it
1234
*XDB> t it
C E (C E (C (C E E) (C E (C E (C (C E E) (C (C E E) (C E E)))))))
*XDB> map (x2t.t2x) [0..15]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Note however that when using the instance `N` of `Cat` which implies the usual binary number representation, the encoding is, as expected, not size proportionate.

This precludes the use of the usual random number generators returning integers in binary notation to generate very large random lambda terms. We will circumvent this problem by using instead an algorithm that (uniformly) generates random binary trees (see Sect. 6).

We define the unranking function `t2b` and the ranking function `b2t` for de Bruijn terms, as follows.

```
t2b :: Cat a ⇒ a → B a
t2b = x2b . t2x

b2t :: Cat a ⇒ B a → a
b2t = x2t . b2x
```

Proposition 3. *The functions* `t2b` *and* `b2t` *converting between open de Bruijn terms and corresponding instances of* `Cat`*, are inverses.*

Example 4. *The encoding and decoding of the de Bruijn form of the pairing combinator* $\lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_2\ x_0)\ x_1)$ *to ordinary binary numbers and binary trees.*

```
*XDB> b2t (Lb (Lb (Lb (Ab (Ab (Vb 0) (Vb 2)) (Vb 1)))))
1389505070847794345082851820104254894239239815
             987686768473491008094957555679247
*XDB> t it
C (C (C E E) E) (C (C E (C E (C (C E (C E (C E E))) E))) (C (C E E) E))
*XDB> t2b it
```

```
Lb (Lb (Lb (Ab (Ab (Vb E) (Vb (C E (C E E)))) (Vb (C E E)))))
*XDB> b2t it
C (C (C E E) E) (C (C E (C E (C (C E (C E (C E E))) E))) (C (C E E) E))
*XDB> n it
1389505070847794345082851820104254894239239815
             98768676847349100809495755567 9247
*XDB> t2b it
Lb (Lb (Lb (Ab (Ab (Vb 0) (Vb 2)) (Vb 1))))
```

To facilitate comparison, it is useful to define the functions `sizeT` that returns the number of internal nodes of the binary tree view of a Catalan object and `sizeX` returning the size of a lambda term in compressed de Bruijn form, in which numeric components `k` and `n` are also measured with `sizeT`.

```
sizeT :: Cat t ⇒ t → t
sizeT x | e_ x = x
sizeT x = s (add (sizeT a) (sizeT b)) where (a,b) = c' x

sizeX :: Cat a ⇒ X a → a
sizeX (Vx k n) = add (sizeT k) (sizeT n)
sizeX (Ax k a b) = s (add (sizeT k) (add (sizeX a) (sizeX b)))
```

Example 5. *The sum of the two sizes on an initial segment of* $\mathbb{N}$ *illustrates the fact that the bijection* `t2x` *is indeed size-proportionate.*

```
*Main> sum (map (sizeT) [0..10000])
114973
*Main> sum (map (sizeX.t2x) [0..10000])
75288
```

Proposition 4. *The average time complexity of* `t2x` *and* `x2t` *is* $O(n)$ *for input size* n *and their worst case time complexity is* $O(n\ log^*(n))$ *when working on instance* `T` *(binary trees).*

Proof. It follows from the fact that the average complexity of `c`, `c'` `s` and `s'` is constant time, the worst case complexity of `s` and `s'` is $O(log^*(n))$ and that $O(n)$ of these are performed by `t2x` and `x2t`.

3.3 Conversion to/from a Canonical Representation of Lambda Terms with Integer Variable Names

We represent standard lambda terms [2] by using the constructors `Ls` for lambda abstractions, `As` for applications and `Vs` for variable occurrences.

```
data S a = Vs a | Ls a (S a) | As (S a) (S a) deriving (Eq,Show,Read)
```

The function `b2s` converts from the de Bruijn representation to lambda terms whose canonical names are provided by natural numbers. We will call them terms in *standard notation*.

```
b2s :: Cat a ⇒ B a → S a
b2s a = f a e [] where
  f :: (Cat a) ⇒ B a → a → [a] → S a
  f (Vb i) _ vs =  Vs (at i vs)
  f (Lb a) v vs = Ls v (f a (s v) (v:vs))
  f (Ab a b) v vs = As (f a v vs) (f b v vs)

at i (x:_) | e_ i = x
at i (_:xs) = at (s' i) xs
```

Note the use of the helper function `at` that associates to an index `i` a variable in position `i` on the list `vs`. As we initialize in `b2s` when calling helper function `f` the list of index variables to `[]`, we enforce that only closed terms (having no free variables) are accepted.

The inverse transformation is defined by the function `s2b`.

```
s2b  :: Cat a ⇒ S a → B a
s2b x = f x [] where
  f :: Cat a ⇒ S a → [a] → B a
  f (Vs x) vs = Vb  (at x vs)
  f (As x y) vs = Ab (f x vs) (f y vs)
  f (Ls v y) vs = Lb a where a = f y (v:vs)
```

Note again the use of `at`, this time to locate the index `i` on the list of variables `vs`. By initializing `vs` with `[]` in the call to helper function `f`, we enforce that only closed terms are accepted.

Proposition 5. *The functions* `s2b` *and* `b2s`*, converting between closed de Bruijn terms and closed standard terms, are inverses.*

Example 6. *The bijection defined by the functions* `s2b` *and* `b2s` *on the term* $\lambda x_0.(\lambda x_1.(x_0\ (x_1\ x_1))\ \lambda x_2.(x_0\ (x_2\ x_2)))$.

```
*XDB> b2s (Lb (Ab (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))
          (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))))
Ls 0 (As (Ls 1 (As (Vs 0) (As (Vs 1) (Vs 1))))
          (Ls 1 (As (Vs 0) (As (Vs 1) (Vs 1)))))
*XDB> s2b it
Lb (Ab (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0))))
           (Lb (Ab (Vb 1) (Ab (Vb 0) (Vb 0)))))
```

4 Normalization with Tree-Based Arithmetic Operations

We will now describe an evaluation mechanism for the (Turing-complete) language of closed lambda terms, called *normal order reduction* [19]. 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*, will be used both ways to evaluate and then return the result as a de Bruijn term.

4.1 Representing Lambdas as Functions in the Implementation Language

The data type `H` represents leaves `Vh` of the lambda tree and applications `Ah` the same way as the standard lambda terms of type `S`. However, Lambda binders, meant to be substituted with terms during β-reduction steps are represented as functions from the domain `H` to itself.

```
data H a = Vh a | Lh (H a → H a) | Ah (H a) (H a)
```

4.2 A HOAS-style Normal Order Reducer

Normal order evaluation [19] ensures that if a normal form exists, it is found after a finite number of steps. In lambda-calculus based functional languages computing a normal form, normalization can be achieved through a HOAS (Higher-Order Abstract Syntax) mechanism, that borrows the substitution operation from the underlying "meta-language". To this end, lambdas are implemented as functions which get executed (usually lazily) when substitutions occur. We refer to [18] for the original description of this mechanism, widely used these days for implementing embedded domain specific languages and proof assistants in languages like Haskell or ML.

The function `nf` implements normalization of a term of type `H`, derived from a closed de Bruijn term. At each normalization step, when encountering a binder of the form `Lh f`, the normalizer `nf` traverses it and it is composed with `f`. At each application step `Ah f a`, 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.

```
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
```

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.

4.3 Closed Terms to/from HOAS

To implement conversion from the type `H` to the type `B` the function `h2b` traverses the application nodes. As in the case of our other transformers, the (simple) numerical computations involved in the transformations will be performed using the arithmetic on Catalan objects of type `Cat`.

```
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’)
```

```
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 (λx → h a (x:xs))
  h (Ab a b) xs = Ah (h a xs) (h b xs)
  h (Vb i) xs  = at i xs
```

Example 7. *Testing that* `h2b` *is a left inverse of* `h2b`*.*

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

While so called “exotic terms” are possible in the data type `H` to which no terms of type `B` correspond, the terms brought to the `H` side by `b2h` and back by `h2b` are identical.

4.4 Evaluating Closed Lambda Terms

As our normal order reduction is borrowed via a HOAS mechanism from the underlying Haskell system, evaluation is restricted to closed terms. Instead of getting help form a `Maybe` type, it is simpler to define its result as the trivial open term `Vb e` for all open terms.

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 8. *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.*

```
*XDB> 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
```

4.5 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.

```
evalT :: T→T
evalT = x2t . evalX . t2x
```

As we have also made the usual natural numbers members of the Catalan family, we can define normal order reduction of such "arithmetized" lambda terms as the arithmetic function `evalN`.

```
evalN :: N→N
evalN = x2t . evalX . t2x
```

Example 9. *Evaluation of binary trees and natural numbers seen as lambda terms.*

```
*XDB> evalT (C (C E (C E E)) (C (C E E) E))
C (C (C E E) E) E
*XDB> 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.

5 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.

5.1 A Generator for Catalan Objects

The function `genCat` implements a simple generator for Catalan objects with a fixed number of internal nodes. Note that computations are expressed in terms of the arithmetic operations on type `Cat`. It uses the function `nums` (see [22]) that generates an initial segment of the set of natural numbers as a Haskell list.

```
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))]
```

Example 10. *Generation of Catalan object with 3 internal nodes and their natural number encodings.*

```
*XDB> 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

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

Given that closed terms have interesting uses in random testing [8], we derive generators for them in compressed de Bruijn and de Bruijn form.

```
genCatX :: Cat a ⇒ a → [X a]
genCatX  = filter isClosedX . map t2x . genCat

genCatB :: Cat a ⇒ a → [B a]
genCatB  = filter isClosedB . map t2b . genCat
```

Example 11. *Generation of closed compressed de Bruijn terms decoded from binary trees with 3 internal nodes.*

```
*XDB> 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
```

5.2 Generation of Lambda Terms via Unranking

While direct enumeration of terms constrained by number of nodes or depth is straightforward in Haskell an unranking algorithm is also usable for generation of large terms, including generation of very large random terms.

Generating Open Terms in Compressed de Bruijn Form. Open terms are generated simply 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)
```

Reusing unranking-based open term generators for more constrained families of lambda terms works when their asymptotic density is relatively high. Fortunately we know from the extensive quantitative analysis available in the literature [6,7,11] when this is the case.

The function genClosedX generates closed terms by filtering the results of genOpenX with the predicate isClosedX.

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

Example 12. *Generation of closed compressed de Bruijn terms. Note the more than* 50 % *closed terms among the first* 10000 *open terms.*

```
*XDB> genClosedX 8
[Vx 1 0,Ax 1 (Vx 0 0) (Vx 0 0),Ax 2 (Vx 0 0) (Vx 0 0)]

*XDB> map x2t (genClosedX 30)
[1,4,8,9,11,12,15,16,19,20,23,24,28]

*XDB> length (genClosedX (t 10000))
5375
```

6 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.

6.1 Generating Random Binary Trees

We will rely on the Haskell library Math.Combinat.Trees to generate binary trees uniformly, using a variant of Rémy's algorithm described in [14], as well as Haskell's built-in random generator from package System.Random. This will allow 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.

The function 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.

```
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)
```

The function 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))
```

6.2 Generating Random Compressed de Bruijn Terms

We will use the bijection t2x from Catalan objects to open compressed de Bruijn trees, parameterized by the function tf that picks the type of the instance of Cat to be used.

The function ranOpenX generates random terms in a way similar to the function ranCat.

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

The function ranOpen1X generates random terms given a seed for the random generator.

```
ranOpen1X tf size seed = t2x (ranCat1 tf size seed)
```

The function ranClosedX filters the generated terms until a closed one is found.

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

The function ranClosed1X works in a similar way, except for providing a seed instead of a random generator.

```
ranClosed1X tf size seed = ranClosedX tf size g where g = mkStdGen seed
```

Example 13. *Generation of some random lambda terms (including very large ones) in compressed de Bruijn form.*

```
*XDB> ranClosed1X n 3 9
Ax 1 (Vx 0 0) (Vx 0 0)
*XDB> ranClosed1X t 3 9
Ax (C E E) (Vx E E) (Vx E E)
*XDB> n (sizeX (ranClosed1X t 100 9))
96
*XDB>  n (sizeX (ranOpen1X t 50000 42))
50001
```

7 Related Work

Originally introduced in [4], the de Bruijn notation makes terms equivalent up to α-conversion and facilitates their normalization [13]. As indices replace variable names by their stack-order relative positioning to their binders, they are already more compact than standard lambda terms. However as iteration of their lambda binders can be seen as a form of unary Peano arithmetic, it made sense to further compress them by counting the binders more efficiently. This mechanism, is first described in [24] where in combination with the generalized Cantor bijection between $\mathbb{N}^k \to \mathbb{N}$ it is used to provide a bijective Gödel numbering scheme. However, as the $\mathbb{N} \to \mathbb{N}^k$ side of this bijection is only computable using a binary search algorithm, it is limited to relatively small terms, by contrast to the one described in this paper that works in time proportional to the size of both the terms and their tree-based number encodings.

In [23] a binary tree-arithmetic encoding is introduced, that can be seen as an instance of the generic Catalan-object arithmetic used in this paper. However, as it describes computations in terms of ordinary arithmetic, it is size-proportionate only under the assumption that variables fit in word-represented integers.

Combinatorics of lambda terms, including enumeration, random generation and asymptotic behavior has seen an increased interest recently (see for instance [3,6,7,10,11]), partly motivated by applications to software testing [8,17] given the widespread use of lambda terms as an intermediate language in compilers for functional languages and proof assistants.

Ranking and unranking of lambda terms can be seen as a building block for bijective serialization of practical data types [15,26] as well as for Gödel numbering schemes of theoretical relevance. In fact, ranking functions for sequences can be traced back to Gödel numberings [9,12] associated to formulas.

8 Conclusions and Future Work

We have provided a fresh look at several aspects of the representation and encoding of lambda terms with focus on their de Bruijn form and a compressed variant of it. Our computations have used a type class defining generic arithmetic operations on members of the Catalan family of combinatorial objects, described in detail in [25]. They have served to implement bijections between representations of terms and conversion to/from HOAS-like form used for normalization of lambda terms. Some interesting synergies have been triggered by this combination of apparently heterogeneous techniques:

- we have provided a simple size-proportionate bijective encoding of compressed De Bruijn terms to our tree-based "natural numbers"
- the same "natural numbers" (actually operated on through a binary tree perspective on Catalan objects), have served to do routine arithmetic operations, with average complexity comparable to the usual binary numbers
- the use of tree-based numbers, a target for ranking/unranking of lambda terms and a uniform random generation algorithm for binary trees, have enabled generation of (possibly very large) random open lambda terms

Note also that our size-proportional encodings as arithmetic-endowed Catalan objects can be easily adapted to other tree representations widely used in computer science and computational sciences, like expression trees, recursive data types, directory structures, parse trees for programming and natural languages, phylogenetic trees, etc.

We have not approached yet some of the remaining hard problems related to uniform random generation of more "realistic" lambda terms appearing in compilers and proof assistants e.g. well-typed and closed terms, for which no linear-time algorithms are known. The techniques, involving binary search, based on ordinary binary numbers for either term algebras in [21] or closed lambda terms in [11] will require more work to adapt to possibly linear algorithms based on our size-proportionate encodings in a tree-based numbering system. Also, an empirical study of the shapes and distribution of frequent lambda term patterns appearing in written and generated code is likely to be useful to fine-tune ranking/unranking algorithms better suited for random generation of such terms.

Acknowledgement. This research has been supported by NSF grant 1423324. We thanks the reviewers of PADL'16 for their constructive suggestions and comments.

References

1. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen **114**, 305–315 (1937)
2. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics, vol. 103, 2, revised edn. Elsevier, North Holland (1984)
3. Bodini, O., Gardy, D., Gittenberger, B.: Lambda-terms of bounded unary height. In: ANALCO, pp. 23–32. SIAM (2011)
4. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae **34**, 381–392 (1972)
5. Cegielski, P., Richard, D.: On arithmetical first-order theories allowing encoding and decoding of lists. Theor. Comput. Sci. **222**(1–2), 55–75 (1999)
6. David, R., Grygiel, K., Kozik, J., Raffalli, C., Theyssier, G., Zaionc, M.: Asymptotically almost all λ-terms are strongly normalizing (2010). Preprint: arXiv:math.LO/0903.5505v3
7. David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., Zaionc, M.: Some properties of random lambda terms. Logical Methods Comput. Sci., 9(1) (2009)
8. Fetscher, B., Claessen, K., Pałka, M., Hughes, J., Findler, R.B.: Making random judgments: automatically generating well-typed terms from the definition of a type-system. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 383–405. Springer, Heidelberg (2015)
9. Gödel, K.: Über formal unentscheidbare Sätze der principia mathematica und verwandter systeme I. Monatshefte für Mathematik und Physik **38**, 173–198 (1931)
10. Grygiel, K., Idziak, P.M., Zaionc, M.: How big is BCI fragment of BCK logic. J. Log. Comput. **23**(3), 673–691 (2013)
11. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5), 594–628 (2013)
12. Hartmanis, J., Baker, T.P.: On simple Goedel numberings and translations. In: Loeckx, J. (ed.) ICALP. Lecture Notes in Computer Science, vol. 14, pp. 301–316. Springer, Berlin Heidelberg (1974)
13. Kamareddine, F.: Reviewing the classical and the de Bruijn notation for calculus and pure type systems. J. Logic Comput. **11**(3), 363–394 (2001)
14. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees-History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional, Boston (2006)
15. Kobayashi, N., Matsuda, K., Shinohara, A.: Functional programs as compressed data. In: ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, ACM Press, January 2012

16. Lisi, M.: Some remarks on the Cantor pairing function. Le Matematiche **62**(1), 55–65 (2007). http://www.dmi.unict.it/ojs/index.php/lematematiche/article/view/14
17. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, pp. 91–97. ACM, New York, NY, USA (2011)
18. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI 1988, pp. 199–208. ACM, New York, NY, USA (1988)
19. Sestoft, P.: Demonstrating lambda calculus reduction. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 420–435. Springer, Heidelberg (2002)
20. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publishing Co., Belmont (1986)
21. Tarau, P.: Compact serialization of prolog terms (with catalan skeletons, cantor tupling and gödel numberings). Theor. Pract. Logic Program. **13**(4–5), 847–861 (2013)
22. Tarau, P.: A generic numbering system based on catalan families of combinatorial objects. CoRR abs/1406.1796 (2014)
23. Tarau, P.: On a uniform representation of combinators, arithmetic, Lambda terms and types. In: Albert, E. (ed.) PPDP 2015: Proceedings of the 17th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, pp. 244–255. ACM, New York, NY, USA, July 2015
24. Tarau, P.: Ranking/unranking of Lambda terms with compressed de Bruijn Indices. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS, vol. 9150, pp. 118–133. Springer, Heidelberg (2015)
25. Tarau, P.: Computing with catalan families, generically. In: Gavanelli, M., Reppy, J. (eds.) PADL'16. LNCS. Springer, St. Petersburg, Florida, USA (2016)
26. Vytiniotis, D., Kennedy, A.: Functional pearl: every bit counts. ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming, ACM Press, September 2010