Ranking/unranking of lambda terms with compressed de Bruijn indices

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CICM/Calculemus'2015

Research supported by NSF grant **1423324.**

Motivation

- λ-calculus is possibly the most heavily researched computational mechanism, but it has a nice property: the deeper you dig, the more interesting it becomes $:-)$
- λ-terms provide a foundation to modern functional languages, type theory and proof assistants
- they are now part of mainstream programming languages including Java, C# and Apple's Swift – (even C++ 11)
- Prolog's backtracking, sound unification and Definite Clause Grammars make it an "all-in-one" meta-language for generation, type inference, symbolic computation
- our bigger goal $\Rightarrow$ build a *declarative playground* for λ-terms, types and combinators

Outline

De Bruijn Indices

Our metalanguage: a subset of Prolog, with occasional use of some built-ins, Horn clauses of the form $a_0 :- a_1, a_2 \ldots a_n.$

- a lambda term: $\lambda a.(\lambda b.(a\ (b\ b))\ \lambda c.(a\ (c\ c))) \Rightarrow$
- in Prolog: l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))
- *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
- term with canonical names: l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C))))) $\Rightarrow$
- de Bruijn term: l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0)))))))
- note: we start counting up from 0
- closed terms: every variable occurrence belongs to a binder
- open terms: otherwise

From de Bruijn terms to terms with canonical names

The predicate `b2l` converts from the de Bruijn representation to lambda terms whose canonical names are provided by logic variables. We will call them terms in *standard notation*.

```
b2l(A,T):-b2l(A,T,_Vs).

b2l(v(I),V,Vs):-nth0(I,Vs,V).          % find binder
b2l(a(A,B),a(X,Y),Vs):-b2l(A,X,Vs),b2l(B,Y,Vs).
b2l(l(A),l(V,Y),Vs):-b2l(A,Y,[V|Vs]).  % define binder
```

the inverse transformation is `l2b` $\rightarrow$ in the paper

```
?- LT=l(A,l(B,l(C,a(a(A,C),a(B,C))))),
   l2b(LT,BT),
   b2l(BT,LT1),
   LT=LT1.
LT = LT1,
LT1 = l(A, l(B, l(C, a(a(A, C), a(B, C))))),
BT = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))).
```

Compressing λ-terms

Figure: Random λ-terms have long necks: l(l(l(l(l(... . . .a(. . .

Figure: But they can be compressed!

λ-term $\Rightarrow$ compressed λ-term

Compressed de Bruijn terms

- iterated λs (represented as a block of constructors `l/1` 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
- $\Rightarrow$ we need only two constructors:
  - `v/2` indicating in a term `v(K,N)` that we have `K` λs wrapped around the de Bruijn index `v(N)`
  - `a/3` indicating in a term `a(K,X,Y)` that `K` λs are wrapped around the application `a(X,Y)`
- we call the terms built this way with the constructors `v/2` and `a/3` *compressed de Bruijn terms*

From de Bruijn to compressed

- `b2c`: by case analysis on `v/1, a/2, l/1`
- it counts the binders `l/1` as it descends toward the leaves of the tree

```
b2c(v(X),v(0,X)).
b2c(a(X,Y),a(0,A,B)):-b2c(X,A),b2c(Y,B).
b2c(l(X),R):-b2c1(0,X,R).

b2c1(K,a(X,Y),a(K1,A,B)):-up(K,K1),b2c(X,A),b2c(Y,B).
b2c1(K, v(X),v(K1,X)):-up(K,K1).
b2c1(K,l(X),R):-up(K,K1),b2c1(K1,X,R).

up(From,To):-From>=0,To is From+1.
```

From compressed to de Bruijn

- `c2b` reverses the effect of `b2c` by expanding the `K` in `v(K,N)` and `a(K,X,Y)` into `K+1 l/1` binders
- `iterLam` performs this operation on both `v/2` and `a/3` terms

```
c2b(v(K,X),R):-X>=0,iterLam(K,v(X),R).
c2b(a(K,X,Y),R):-c2b(X,A),c2b(Y,B),iterLam(K,a(A,B),R).
```

surround applications or indices with K lambdas

```
iterLam(0,X,X).
iterLam(K,X,l(R)):-down(K,K1),iterLam(K1,X,R).

down(From,To):-From>0,To is From-1.
```

Composing relations with Definite Clause Grammars (DCGs)

A convenient way to simplify defining compositions of relations or functions is by using Prolog's Definite Clause Grammars (DCGs), which transform a clause defined with "`-->`" like

```
a0 --> a1,a2,...,an.
```

into

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

```
c2l --> c2b, b2l.

l2c --> l2b, b2c.
```

- the predicate `c2l/2` which can be seen as specifying a composition of two functions, expands to something like
  `c2l(X,Z):-c2b(X,Y),b2l(Y,Z)`
- it converts from compressed de Bruijn terms and standard lambda terms using de Bruijn terms as an intermediate step
- `l2c/2` works the other way around

Inferring simple types – for terms in standard notation

- *simple types*: binary trees built with the constructor "`->/2`"
- empty leaves: representing the unique primitive type "`o`"
- `extractType/2` works by turning each logical variable `X` into a pair `_:TX`, where `TX` is a fresh variable denoting its type
- as logic variable bindings propagate between binders and occurrences, this ensures that types are consistently inferred

```
extractType(_:TX,TX):-!. % this matches leaf variables
extractType(l(_:TX,A),(TX->TA)):- % lambda variable
  extractType(A,TA).
extractType(a(A,B),TY):- % application
  extractType(A,(TX->TY)),
  extractType(B,TX).
```

- binding with the base type "`o`":

```
bindType(o):-!.
bindType((A->B)):-bindType(A),bindType(B).
```

Ensuring that types are acyclic terms

```
hasType(CTerm,Type):-
  c2l(CTerm,LTerm), % from compressed to standard
  extractType(LTerm,Type),
  % we do this once -- instead of occurs-check at each step
  acyclic_term(LTerm),
  bindType(Type).
```

- typability of the term corresponding to the `S` combinator
  $\lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\ x_2)\ (x_1\ x_2))$
- untypabilty of the term corresponding to the `Y` combinator
  $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$

```
?- hasType(a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),T).
T = ((o->o->o)-> (o->o)->o->o).

?- hasType(
a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),T).
false.
```

Generating well typed de Bruijn terms of a given size

- we can interleave generation and type inference in one program
- DCG grammars control size of the terms with predicate `down/2`
- in terms of the Curry-Howard correspondence, the size of the generated term corresponds to the size of the (Hilbert-style) proof of the intuitionistic formula defining its type

```
genTypedB(v(I),V,Vs)-->
  {
    nth0(I,Vs,V0), % pick binder and ensure types match
    unify_with_occurs_check(V,V0)
  }.
genTypedB(a(A,B),Y,Vs)-->down, % application node
  genTypedB(A,(X->Y),Vs),
  genTypedB(B,X,Vs).
genTypedB(l(A),(X->Y),Vs)-->down, % lambda node
  genTypedB(A,Y,[X|Vs]).
```

Querying for inhabitants of a given type

```
genTypedB(L,B,T):-genTypedB(B,T,[],L,0),bindType(T).

queryTypedB(L,Term,QueryType):-
  genTypedB(L,Term,Type),
  Type=QueryType.
```

- Terms of type `x>x` of size 4

```
?- queryTypedB(4,Term,(o->o)).
Term = a(l(l(v(0))), l(v(0))) ;
Term = l(a(l(v(1)), l(v(0)))) ;
Term = l(a(l(v(1)), l(v(1)))) .

?- queryTypedB(10,Term,((o->o)->o)).
false.
```

- the last query, taking about half a minute, shows that no closed terms of type `(o->o)->o` exist up to size 10

Generating well-typed, closed BCK(p) terms of a given size

code not in the paper: we interleave generation and multiple constraints

BCK(p): at most p occurrences for each lambda binder (p>1: Turing-complete)

```
genTBCK(K,L,X,T):-genTBCX(X,T,K,_I,0,[],[],L,0). % for I_=0, BCI(p)

genTBCX(v(X),T,_K1,_K2,V,Vs1,Vs2)--> {
    selsub(V,X:C1:T0,X:C2:T,Vs1,Vs2),down(C1,C2),
    unify_with_occurs_check(T,T0)
  }.
genTBCX(l(A),(X->Y),K1,K2,V,Vs1,Vs2)-->down,
  {up(V,NewV)},
  genTBCX(A,Y,K1,K2,NewV,[V:K1:X|Vs1],[V:NewK:_|Vs2]),
  {\+ \+ (NewK=K2)}.
genTBCX(a(A,B),Y,K1,K2,V,Vs1,Vs3)-->down,
  genTBCX(A,(X->Y),K1,K2,V,Vs1,Vs2),
  genTBCX(B,X,K1,K2,V,Vs2,Vs3).

selsub(I,X,Y,[X|Xs],[Y|Xs]):-down(I,_).
selsub(I,X,Y,[Z|Xs],[Z|Ys]):-down(I,I1),selsub(I1,X,Y,Xs,Ys).
```

Size-proportionate bijective encodings of lambda terms

- injective encodings are easy: encode each symbol as a small integer and use a separator – or use Gödel's original prime number products
- in the presence of a bijection between two infinite sets of data objects, it is possible that representation sizes on one side or the other are exponentially larger that on the other
- e.g., Ackerman's bijection from hereditarily finite sets to natural numbers

Definition

Given a bijection between sets of terms of two datatypes denoted M and N, $f : M \to N$, let $m(x)$ be the representation size of a term $x \in M$ and $n(y)$ be the representation size of $y \in N$. Then f is called *size-proportionate* if $|m(x) - n(y)| \in O(log(max(m(x), n(y))))$.

- Informally we also assume that the constants involved are small enough such that the printed representation of two data objects connected by the bijections is about the same

The generalized Cantor k-tupling bijection

- we need a bijection between $\mathbb{N}^k$ and $\mathbb{N}$ to aggregate / separate between a list of codes and a code
- it is conjectured that the only one given by a polynomial formula (up to a permutation of its variables) is the generalized Cantor n-tupling bijection:

$$K_n(x_1,\ldots,x_n) = \binom{n-1+x_1+\ldots+x_n}{n} + \ldots + \binom{1+x_1+x_2}{2} + \binom{x_1}{1} \quad (1)$$

- $\binom{n}{k}$: binomial coefficient, "n choose k"
- note the prefix sums $X_1,\ X_1+X_2,\ X_1+X_2+\ldots+X_K \Rightarrow$ next slide
- EXAMPLE: $K_3(x_1,x_2,x_3) = \binom{2+x_1+x_2+x_3}{3} + \binom{1+x_1+x_2}{2} + \binom{x_1}{1}$
- $K_3(2,0,3) = \binom{2+2+0+3}{3} + \binom{1+2+0}{2} + \binom{2}{1} = \binom{7}{3} + \binom{3}{2} + \binom{2}{1}$
- $K_3(2,0,3) = 35+3+2 = 40$

```
?- from_cantor_tuple([2,0,3],N).
N = 40.
```

Encoding λ-terms: we use compressed de Bruijn forms!

to encode a compressed deBruijn term as a natural number we need:

- an encoding of the applicative skeleton (a binary tree)
- an encoding of the symbol lists that is size-proportionate
- a mechanism to merge the skeleton and content encodings together

a first step is:

- lists and sets of natural numbers (represented canonically) can be morphed into each other using a simple bijection
- the prefix sums of a sequence encode a set! (code in online appendix)

```
?- list2set([2,0,1,2],Set).
Set = [2, 3, 5, 8].

?- set2list([2, 3, 5, 8],List).
List = [2, 0, 1, 2].
```

The generalized Cantor tupling: from $\mathbb{N}^K$ to $\mathbb{N}$

quite simple: morph lists to sets, then sum up binomials

```
fromCantorTuple(Ns,N) :-list2set(Ns,Xs), fromKSet(Xs,N).

fromKSet(Xs,N):-untuplingLoop(Xs,0,0,N).

untuplingLoop([],_L,B,B).
untuplingLoop([X|Xs],L1,B1,Bn) :-
  L2 is L1+1,
  binomial(X,L2,B),
  B2 is B1+B,
  untuplingLoop(Xs,L2,B2,Bn).
```

The case $n = 2$: Cantor's pairing function

- $K_2(x_1, x_2) = x_1 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$
- $K_2'(x_1, x_2) = x_2 + \frac{(x_1+x_2+1)(x_1+x_2)}{2}$ (symmetric alternative)
- K_2 and K_2' are the only ones known to be polynomials in x_1, x_2
- the inverse of $K_2(x_1, x_2)$ has a simple closed formula - involving an integer square root operation, but we have found no algorithm for computing the inverse in the n>2 case in the literature
- the naive inversion algorithm - "enumerate until you hit the right one" becomes quickly intractable
- we needed to invent one – see ICLP'13 paper for its derivation steps

The inverse of Cantor's pairing function: a geometric view

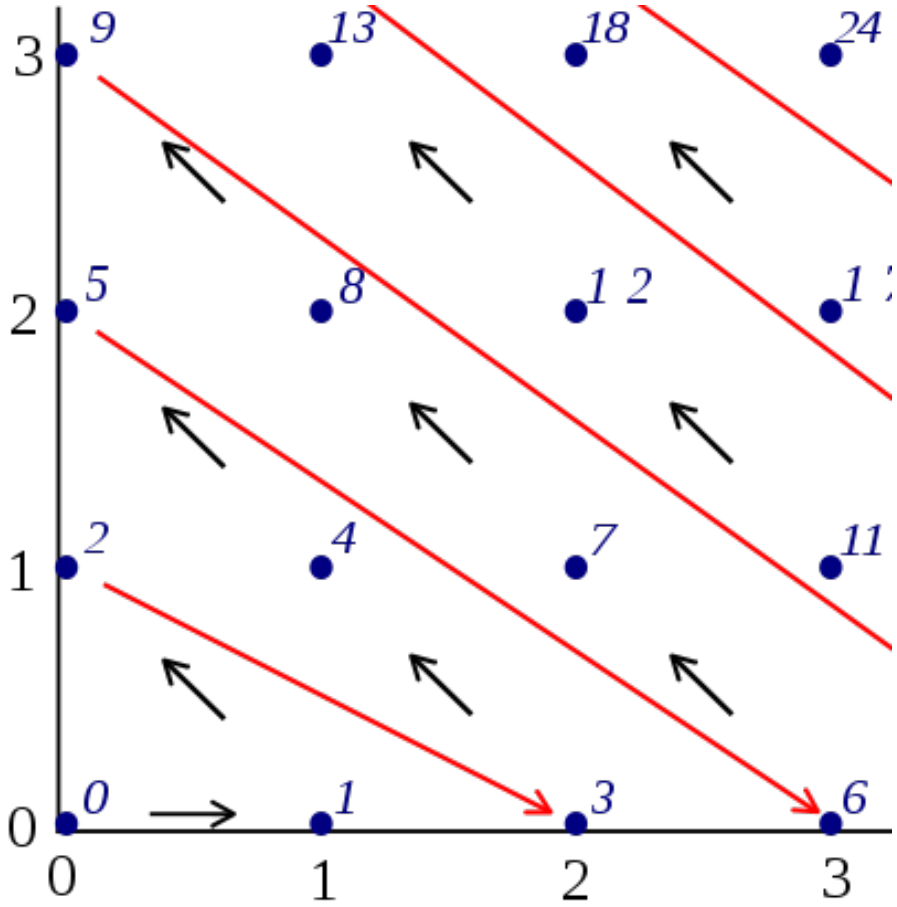

Figure: Path connecting pairs associated to successive natural numbers by the inverse of Cantor's pairing function (credit: Wikipedia)

A fast algorithm for the inverse $\mathbb{N} \rightarrow \mathbb{N}^K$ bijection

- we need this for efficient unranking from $\mathbb{N}$ of terms
- we split our problem in two simpler ones: inverting `fromKSet` and then applying `set2list` to get back from sets to lists
- the predicate `untuplingLoop` used by `fromKSet` implements the sum of the combinations $\binom{x_1}{1} + \binom{x_2}{2} + \ldots + \binom{x_K}{K} = N$, the representation of N in the *combinatorial number system of degree K*
- conversion algorithms between the conventional and the combinatorial number system are well known, except that one should be careful to ensure they scale up when very large numbers are involved

Finding the combinatorial digits efficiently

```
toKSet(K,N,Ds):-combinatoriallDigits(K,N,[],Ds).

combinatoriallDigits(0,_,Ds,Ds).
combinatoriallDigits(K,N,Ds,NewDs):-K>0,K1 is K-1,
  upperBinomial(K,N,M),   % <<<<<<<<<< key optimization here
  M1 is M-1,
  binomial(M1,K,BDigit),
  N1 is N-BDigit,
  combinatoriallDigits(K1,N1,[M1|Ds],NewDs).
```

inside `upperBinomial`:

- the predicate `roughLimit` narrows the range for the largest digit
- the predicate `binarySearch` finds the largest digit
- `combinatoriallDigits` iterates to the next digit

The optimizations for finding the largest combinatorial digit

```
roughLimit(K,N,I, L):-binomial(I,K,B),B>N,!,L=I.
roughLimit(K,N,I, L):-J is 2*I,roughLimit(K,N,J,L).
```

The predicate `binarySearch` finds the exact value of the combinatorial digit in the interval `[L,M]`, narrowed down by `roughLimit`.

```
binarySearch(_K,_N,From,From,R):-!,R=From.
binarySearch(K,N,From,To,R):-Mid is (From+To) // 2,
  binomial(Mid,K,B),
  splitSearchOn(B,K,N,From,Mid,To,R).

splitSearchOn(B,K,N,From,Mid,_To,R):-B>N,!,
   binarySearch(K,N,From,Mid,R).
splitSearchOn(_B,K,N,_From,Mid,To,R):-Mid1 is Mid+1,
   binarySearch(K,N,Mid1,To,R).
```

The efficient $\mathbb{N} \rightarrow \mathbb{N}^K$ inverse mapping at work

The predicates `toKSet` and `fromKSet` implement inverse functions, mapping between natural numbers and canonically represented sets of `K` natural numbers.

```
?- toKSet(5,2014,Set),fromKSet(Set,N).
Set = [0, 3, 4, 5, 14], N = 2014 .
```

The efficient inverse of Cantor's N-tupling is now simply:

```
toCantorTuple(K,N,Ns):-toKSet(K,N,Ds),set2list(Ds,Ns).
```

The following example illustrates that it works as expected, including on very large numbers:

```
?- K=1000,pow(2014,103,N),
   toCantorTuple(K,N,Ns),fromCantorTuple(Ns,N).
K = 1000,
N = 208029545585703688484419...567744,
Ns = [0, 0, 2, 0, 0, 0, 0, 0, 1|...] .
```

Ranking/unranking with Catalan skeletons

Catalan skeleton: the binary tree describing the applicative structure of a compressed de Bruijn term

- we use the bijection between two members of the Catalan family of combinatorial objects:
  - rooted ordered binary trees with empty leaves
  - the language of balanced parentheses
- the bijection between them is *size-proportionate*
- implemented by the reversible Prolog predicate `catpar`
- we extract the Catalan skeleton of a compressed de Bruijn term
- we transform it to a string of balanced parentheses
- we rank/unrank them to natural numbers, via a size-proportionate bijection
- details of the code in the paper, we use a well-known ranking/unranking algorithm

Ranking

- we “split” a lambda tree into its Catalan skeleton and the list of atomic objects labeling its nodes
- the predicate `rankTerm/2` defines the bijective encoding of an (open) compressed de Bruijn term

```
rankTerm(Term,Code):-
  toSkel(Term,Ps,Ns), % split into skeleton + labels
  rankCatalan(Ps,CatCode), % rank skeleton
  fromCantorTuple(Ns,VarsCode), % rank list of labels
  fromCantorTuple([CatCode,VarsCode],Code). % merge encodings
```

Unranking

the steps of the ranking algorithm, in reverse order

```
unrankTerm(Code,Term):-
  toCantorTuple(2,Code,[CatCode,VarsCode]), % unpair encodings
  unrankCatalan(CatCode,Ps),                % rank skeleton
  length(Ps,L2),
  L is (L2-2) div 2,
  L3 is 3*L+2,                  % compute size of list of labels
  toCantorTuple(L3,VarsCode,Ns), % decode labels
  fromSkel(Ps,Ns,Term).         % decode skeleton + labels
```

- given the unranking of `CatCode` as a list of balanced parentheses of length `2*L+2`, we can determine the number `L` of internal nodes of the tree and the number `L+1` of leaves
- then we have `2*(L+1)` labels for the leaves and `L` labels for the internal nodes, for a total of `3L+2`, value needed to decode the labels encoded as `VarsCode`

Example

"size-proportionate" encoding of the compressed de Bruijn terms corresponding to the combinators `S` and `Y`.

```
?- T = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
   rankTerm(T,R),unrankTerm(R,T1).
T = T1,T1 = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
R = 56493141 .

?- T=a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))
   rankTerm(T,R),unrankTerm(R,T1).
T=T1,T1=a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)
R = 261507060 .
```

Generation of random lambda terms via unranking

- generation of random lambda terms via unranking of random integers of a given bit-size, is implemented by the predicate `ranTerm/3`
- it applies the predicate `Filter` repeatedly until a term is found for which the predicate `Filter` holds

```
ranTerm(Filter,Bits,T):-X is 2^Bits,N is X+random(X),M is N+X,
  between(N,M,I),    % generate integers between N and M
   unrankTerm(I,T), % unrank term corresponding to integer I
   call(Filter,T),  % pipe generated terms through Filter
  !.        % cut generation when first desired term found
```

- random open terms are generated by `ranOpen/2`
- random closed terms are generated by by the predicate `ranClosed`
- random typable terms are generated by `ranTyped`
- closed typable terms are generated by `closedTypable/2`

```
ranOpen(Bits,T):-ranTerm(=(_),Bits,T).
ranClosed(Bits,T):-ranTerm(isClosed,Bits,T).
ranTyped(Bits,T):-ranTerm(closedTypable,Bits,T).
closedTypable(T):-isClosed(T),typable(T).
```

Conclusion

Prolog code at:

`http://www.cse.unt.edu/~tarau/research/2015/dbr.pro`

- logic programming was used as a meta-language for lambda terms and types – with focus on a bijective size-proportionate encodings
- possible applications:
  - we can generate very large random open terms
  - for closed and well-typed random terms, their size is limited by their asymptotic sparsity
  - also: serialization or and storage of lambda terms (possibly representing proofs)

Compactness and simplicity of the code is coming from a combination of:

- logic variables / unification with occurs check / acyclic term testing
- Prolog's backtracking – and occasional CUTs `:-)`
- DCGs for size testing in generators and for relation composition

The same is doable in functional programming - but with a much richer "language ontology" needed for managing state, backtracking, unification.

Future work (and work rom the close enough past)

Extending our “declarative playground” for lambda terms and combinators:

- PADL’15: generation of various families of lambda terms
- PPDP’15: a uniform representation of combinators, arithmetic, lambda terms, ranking/unranking to tree-based numbering systems
- ICLP’15: type-directed generation of lambda terms
- plans to release it all together as a large arxiv draft + github code