On logic programming representations of lambda terms: de Bruijn indices, compression, type inference, combinatorial generation, normalization

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PADL'2015

Motivation

- 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)
- λ-calculus is possibly the most heavily researched computational mechanism – newcomers beware!
- it is also thrilling: despite its apparent simplicity there are endless surprises at every turn
- Prolog's backtracking, sound unification and DCGs make it an ideal meta-language for studying various families of lambda terms
- our double temptation $\Rightarrow$ while (re)visiting the use of logic programming as a modeling tool, we will lay down some building blocks of a *declarative playground* for λ-terms, types and combinators

Outline

De Bruijn Indices

- 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 ...

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).

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.
```

Generating binary trees

- binary trees are a member of the Catalan family of combinatorial objects
- compressed de Bruijn terms can be seen as as binary trees decorated with integer labels
- with DCGs: a generator / recognizer of binary trees of a fixed size

```
scat(N,T):-scat(T,N,0).

scat(o)-->[].
scat((X->Y))-->down,scat(X),scat(Y).

?- scat(3,BT).
BT = (o->o->o->o) ;
...
BT = (((o->o)->o)->o) .
```

also, DCGs express nicely 1-argument function compositions!

```
% from compressed deBruijn to standard and back
c2l --> c2b, b2l.
l2c --> l2b, b2c.
```

Simple types

- *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 base type:

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

Ensuring types are acyclic terms

```
hasType(CTerm,Type):-
  c2l(CTerm,LTerm), % from compressed to standard
  extractType(LTerm,Type),
  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 Motzkin trees

- Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2
- $\Rightarrow$ like lambda term trees, for which we ignore the de Bruijn indices that label their leaves

```
motzkinTree(L,T):-motzkinTree(T,L,0).

motzkinTree(u)-->down.
motzkinTree(l(A))-->down,
  motzkinTree(A).
motzkinTree(a(A,B))-->down,
  motzkinTree(A),
  motzkinTree(B).
```

Generating closed de Bruijn terms

- we can derive a generator for closed lambda terms in de Bruijn form by extending the Motzkin-tree generator to keep track of the lambda binders
- when reaching a leaf `v/1`, one of the available binders (expressed as a de Bruijn index) will be assigned to it nondeterministically

```
genDB(v(X),V)-->
   {down(V,V0),
   between(0,V0,X)}. % we pick one of the binders here
genDB(l(A),V)-->down,
   {up(V,NewV)},
   genDB(A,NewV).
genDB(a(A,B),V)-->down,
   genDB(A,V),
   genDB(B,V).
```

Generating closed de Bruijn terms – continued

```
genDB(L,T):-genDB(T,0,L,0). % terms of size L
genDBs(L,T):-genDB(T,0,L,_). % terms of size up to L
```

generation of terms with 3 internal nodes.

```
?- genDB(3,T).
T = l(l(l(v(0)))) ;
T = l(l(l(v(1)))) ;
T = l(l(l(v(2)))) ;
T = l(l(a(v(0), v(0)))) ;
T = l(l(a(v(0), v(1)))) ;
T = l(l(a(v(1), v(0)))) ;
T = l(l(a(v(1), v(1)))) ;
T = l(a(v(0), l(v(0)))) ;
T = l(a(v(0), l(v(1)))) ;
T = l(a(v(0), a(v(0), v(0)))) ;
T = l(a(l(v(0)), v(0))) ;
T = l(a(l(v(1)), v(0))) ;
T = l(a(a(v(0), v(0)), v(0))) ;
T = a(l(v(0)), l(v(0))) .
```

Generators for compressed de Bruijn and standard terms

- we compose generators with transformers between representations
- we use DCGs to compose binary relations

de Bruijn terms

```
genCompressed --> genDB,b2c.    % of size L
genCompresseds --> genDBs,b2c. % up to size L
```

standard terms

```
genStandard --> genDB,b2l.    % of size L
genStandards --> genDBs,b2l. % up to size L
```

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).
?- genStandard(2,T).
T = l(_G3434, l(_G3440, _G3440)) ;
T = l(_G3434, l(_G3440, _G3434)) ;
T = l(_G3437, a(_G3437, _G3437)).
```

Generating closed typable terms

- again, we compose generators with transformers between isomorphic representations

```
genTypable(L,T):-genCompressed(L,T),typable(T).
genTypables(L,T):-genCompresseds(L,T),typable(T).

?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).
```

Generation of linear lambda terms

Linear lambda terms restrict binders to *exactly one* occurrence.

- The predicate `linLamb/4` uses logic variables both as leaves and as lambda binders and generates terms in standard form
- binders accumulated on the way down from the root, must be split between the two branches of an application node

```
linLamb(X,[X])-->[].
linLamb(l(X,A),Vs)-->down,
  linLamb(A,[X|Vs]).
linLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)}, % splits Vs, ensures linearity
  linLamb(A,As),
  linLamb(B,Bs).
```

Ensuring linearity

At each step of `subset_and_complement_of/3`, `place_element/5` is called to distribute each element of a set to exactly one of two disjoint subsets.

```
subset_and_complement_of([],[],[]).
subset_and_complement_of([X|Xs],NewYs,NewZs):-
  subset_and_complement_of(Xs,Ys,Zs),
  place_element(X,Ys,Zs,NewYs,NewZs).

place_element(X,Ys,Zs,[X|Ys],Zs).
place_element(X,Ys,Zs,Ys,[X|Zs]).
```

interface generating linear compressed de Bruijn terms:

```
linLamb(L,CT):-linLamb(T,[],L,0),l2c(T,CT).
```

Generating linear affine lambda terms

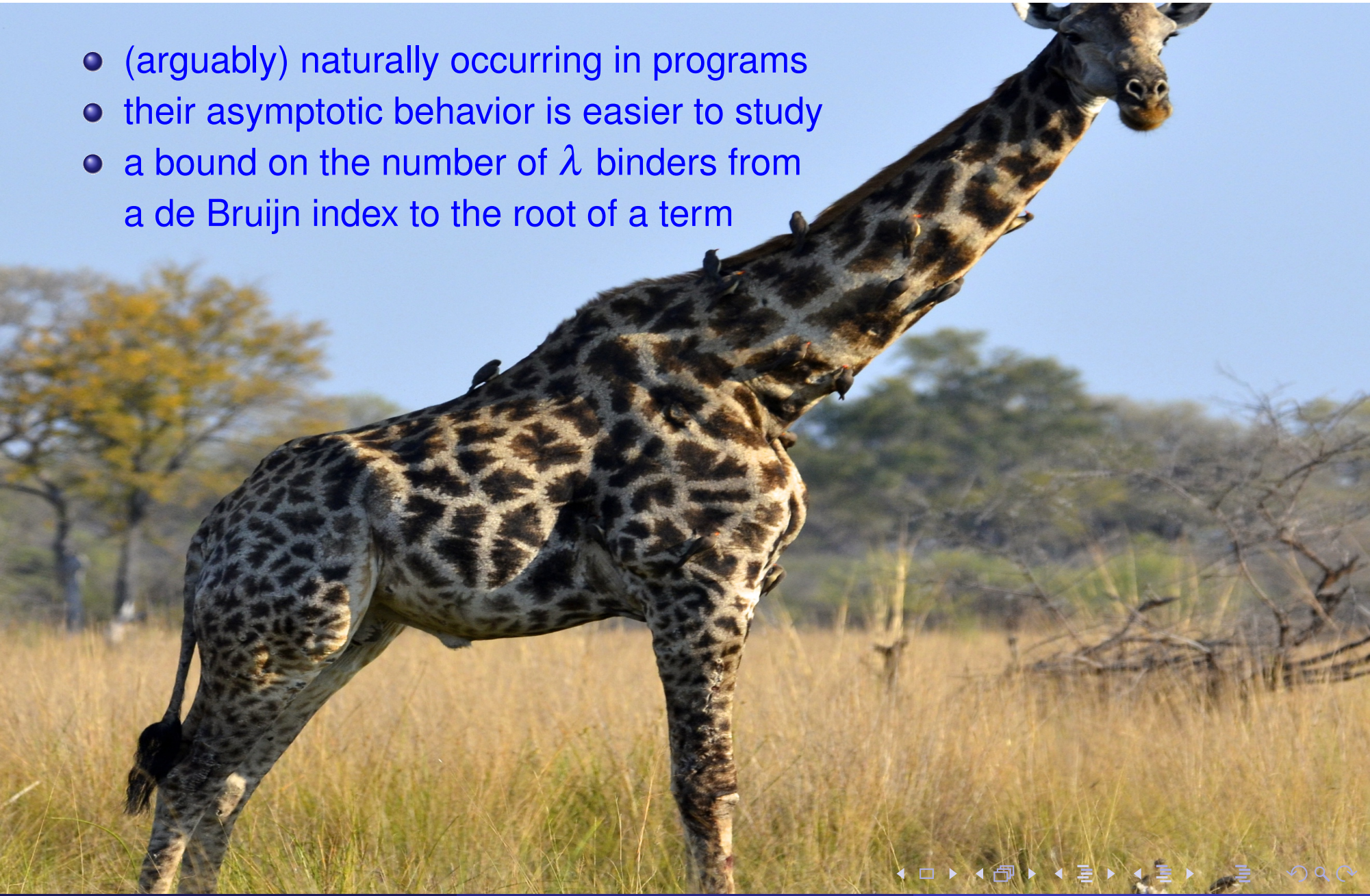
- Linear affine lambda terms restrict binders to *at most one* occurrence.
- code is similar to the generator for linear terms `linLamb` – see paper

Lambda terms for `L=3`.

```
?- linLamb(3,T).
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) .

?- afLinLamb(3,T).
T = v(3, 0) ;                  % 3 lambdas, 1 de Bruijn index
T = a(2, v(0, 1), v(0, 0)) ; % these are the same
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) .
```

Lambda terms of bounded unary height



- (arguably) naturally occurring in programs
- their asymptotic behavior is easier to study
- a bound on the number of λ binders from a de Bruijn index to the root of a term

Generating lambda terms of bounded unary height

- lambdas decrease the available depth parameter (D1 to D2)
- recursion is controlled both by size (DCGs do it implicitly) and by depth (via the parameter D)

```
boundedUnary(v(X),V,_D)-->{down(V,V0),between(0,V0,X)}.
boundedUnary(l(A),V,D1)-->down,
   {down(D1,D2),up(V,NewV)},
   boundedUnary(A,NewV,D2).
boundedUnary(a(A,B),V,D)-->down,
   boundedUnary(A,V,D),boundedUnary(B,V,D).
```

two interfaces:

```
boundedUnary(D,L,T):-boundedUnary(B,0,D,L,0),b2c(B,T).  % of size L
boundedUnarys(D,L,T):-boundedUnary(B,0,D,L,_),b2c(B,T). % up to size L
```

Generating terms of the binary lambda calculus

- the predicate `blc(L,T,Cs)` generates the binary (0,1-list) Cs of length `L` representing the de Bruijn term `T`

```
blc(L,T,Cs):-length(Cs,L),blc(B,0,Cs,[]),b2c(B,T).

blc(v(X),V)-->{between(1,V,X)},encvar(X).
blc(l(A),V)-->[0,0],{up(V,NewV},blc(A,NewV).
blc(a(A,B),V)-->[0,1],blc(A,V),blc(B,V).
```

- de Bruijn binders are encoded as `00`, applications as `01` and de Bruijn indices in unary notation are encoded as `00...01`.
- the predicate `encvar/3`, uses `down/2` at each step to generate the sequence of `1` terminated `0` digits

```
encvar(0)-->[0].
encvar(N)-->{down(N,N1)},[1],encvar(N1)
```

Generating normal forms

- normal forms are lambda terms that cannot be further reduced
- a normal form should not be an application with a lambda as its left branch and, recursively, its subterms should also be normal forms
- the predicate `nf/4` defines this inductively and generates all normal forms with `L` internal nodes in de Bruijn form

```
nf(v(X),V)-->{down(V,V0),between(0,V0,X)}.
nf(l(A),V)-->down,
   {up(V,NewV)},
   nf(A,NewV).
nf(a(v(X),B),V)-->down,
   nf(v(X),V),
   nf(B,V).
nf(a(a(X,Y),B),V)-->down,
   nf(a(X,Y),V),
   nf(B,V).
```

Generating compressed de Bruijn normal forms

Two interfaces:

```
nf(L,T):-nf(B,0,L,0),b2c(B,T).  % size exactly L
nfs(L,T):-nf(B,0,L,_),b2c(B,T). % size up to L

?- nf(3,T).
T = v(3, 0) ;
T = v(3, 1) ;
T = v(3, 2) ;
T = a(2, v(0, 0), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 1), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(0, 0), v(1, 1)) ;
T = a(1, v(0, 0), a(0, v(0, 0), v(0, 0))) ;
T = a(1, a(0, v(0, 0), v(0, 0)), v(0, 0)) ;
false.
```

Normalization: β-reduction

- the predicate `subst/4` counts, starting from `0` the lambda binders down to an occurrence `v(N)`. Replacement occurs at at level `I` when `I=N`

```
beta(l(A),B,R):-subst(A,0,B,R).

subst(a(A1,A2),I,B,a(R1,R2)):-I>=0,
  subst(A1,I,B,R1),
  subst(A2,I,B,R2).
subst(l(A),I,B,l(R)):-I>=0,I1 is I+1,subst(A,I1,B,R).
subst(v(N),I,_B,v(N1)):-I>=0,N>I,N1 is N-1.
subst(v(N),I,_B,v(N)):-I>=0,N<I.
subst(v(N),I,B,R):-I>=0,N=:=I,shift_var(I,0,B,R).
```

- when the right occurrence `v(N)` is reached, the term substituted for it is shifted such that its variables are marked with the new, incremented distance to their binders
- code for `shift_var` – in the paper

The normalization algorithm

- not all derivations terminate - the calculus is only weakly normalizing!
- focus on leftmost outermost term - handled by `wh_nf` that reduces to *weak head normal form* – code in the paper

```
to_nf(v(X),v(X)).
to_nf(l(E),l(NE)):-to_nf(E,NE).
to_nf(a(E1,E2),R):-wh_nf(E1,NE),to_nf1(NE,E2,R).
```

- case analysis of application terms for possible β-reduction is performed by `to_nf1/3`
- the second clause calls `beta/3` and recurses on its result

```
to_nf1(v(E1),E2,a(v(E1),NE2)):-to_nf(E2,NE2).
to_nf1(l(E),E2,R):-beta(l(E),E2,NewE),to_nf(NewE,R). % reduction step
to_nf1(a(A,B),E2,a(NE1,NE2)):-to_nf(a(A,B),NE1),to_nf(E2,NE2).
```

Extending the evaluator to standard and compressed λ-terms

- DCGs used for function composition "borrowing" normalization from de Bruijn terms

```
evalStandard-->l2b,to_nf,b2l.
evalCompressed-->c2b,to_nf,b2c.
```

- evaluation of the lambda term $SKK =$ $((\lambda x_0.\, \lambda x_1.\, \lambda x_2.((x_0\ x_2)\ (x_1\ x_2)))\ \lambda x_3.\, \lambda x_4.x_3)\ \lambda x_5.\, \lambda x_6.x_5)$ in compressed de Brijn form
- reduction to the definition of the identity combinator $I = \lambda x_0.x_0$

```
?- S=a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),K=v(2,1),
     evalCompressed(a(0,a(0,S,K),K),R).
S = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
K = v(2, 1),
R = v(1, 0).
```

Matching the integer sequences in Sloane's OEIS

- a simple sanity check
- two definitions of term size can be used
  - leaves can be of size `0` or size `1`
  - the two options lead to different sequences

- A003095 counts binary trees, by depth
- A000108 counts binary trees, by size
- A001006 and A006318 count Motzkin-trees, by size
- A220894 and A135501 count closed de Bruijn terms by number of internal nodes
- A220471 counts closed typable terms, by size
- A224345 counts closed normal forms, by size
- A114852: counts terms of binary lambda calculus, by size

Conclusion

Prolog code at:
`http://www.cse.unt.edu/~tarau/research/2015/dbx.pro`

We have (re)visited the use of logic programming as a meta-language for lambda terms and types - with focus on generation and type inference.

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 implicit depth or size testing in generators
- DCGs for function and relation composition

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

Future work – from the close enough past :-)

Ongoing work on extending the playground:

- PPDP'15: a uniform representation of combinators, arithmetic, lambda terms, ranking/unranking to tree-based numbering systems
- CICM/Calculemus'15: size-proportionate Gödel numberings, random generation
- ICLP'15: type-directed generation of lambda terms

Playing with the playground - a Minecraft-like experience :-)