On Type-holding and type-repelling lambda-term skeletons, with applications to all-term and random-term generation of simply-typed closed lambda terms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CLA'2017

Research supported by NSF grant **1423324**

Motivation

- *simply-typed closed lambda terms* enjoy a number of nice properties, but the study of their combinatorial properties is notoriously hard
- the two most striking things when inferring types:
  - non-monotonicity: crossing a lambda increases the size of the type, while crossings an application node trims it down
  - agreement via unification (with occurs check) between the types of each variable under a lambda
- as a “methodical” work-around: can we unwrap some of new “observables” that highlight interesting statistical properties?
- $\Rightarrow$ going deeper in the structure of the term than what their CF-syntax reveals
- $\Rightarrow$ we need abstraction mechanisms that “forget” properties of the difficult class (simply-typed closed lambda terms) to reveal equivalence classes that might be easier to grasp $\rightarrow$ k-colored Motzkin skeletons
- bijections with simpler things: from binary trees to 2-colored Motzkin trees

Outline

Making lambda terms (somewhat) more colorful

- *2-colored Motzkin trees*: the free algebra generated by the constructors `v/0`, `l/1`, `r/1` and `a/2`
- we split lambda constructors into 2 classes:
  - *binding lambdas*, `l/1` that are reached by at least one de Bruijn index
  - *free lambdas*, `r/1`, that cannot be reached by any de Bruijn index
- *a side note*: free lambda terms will have no say on terms being closed, but they will impact on which closed terms are simply typed
- *lambda terms in de Bruijn form*: as the free algebra generated by the constructors `l/1`, `r/1` and `a/2` with leaves labeled with natural numbers (and seen as wrapped with the constructor `v/1` when convenient)
- *2-colored Motzkin skeleton of a lambda term*: the tree obtained by erasing the de Bruijn indices labeling their leaves

A bijection between 2-colored Motzkin trees and non-empty binary trees

A bijection between 2-colored Motzkin trees and non-empty binary trees

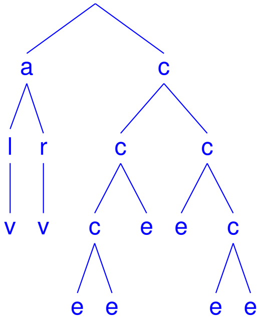
- *binary trees*: the free algebra generated by `e/0` and `c/2`
- *bijections between* the Catalan family of combinatorial objects and 2-colored Motzkin trees : they exist but they involve artificial constructs (e.g., depth first search on rose-trees, indirect definition of the mapping)
- a reason to find a new one ! In Prolog we need *one* relation for *f* and f^{-1}:

```
cat_mot(c(e,e),v).
cat_mot(c(X,e),l(A)):-X=c(_,_),cat_mot(X,A).
cat_mot(c(e,Y),r(B)):-Y=c(_,_),cat_mot(Y,B).
cat_mot(c(X,Y),a(A,B)):-X=c(_,_),Y=c(_,_),
  cat_mot(X,A),
  cat_mot(Y,B).
```

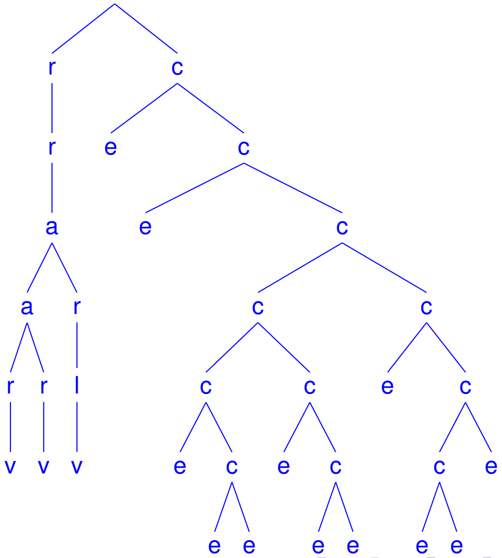
- one can include the empty tree `e` by an arithmetization mechanism that defines successor and predecessor (e.g., our PPDP'15 paper)

The 2-Motzkin to non-empty binary trees bijection at work

two “twinned” trees of size 4



two “twinned” trees size 10



Motzkin skeletons for closed, affine and linear terms

Generating 2-Motzkin trees

- size of constructors is their arity a=2, l=1, r=1, v=0
- in fact, this size definition matches the size of their heap representation
- generate all the splits of the size N into 2*A+L+R where A=size of a/2 etc.
- define predicates to consume size units as needed:

```
lDec(c(SL,R,A),c(L,R,A)):-succ(L,SL).
rDec(c(L,SR,A),c(L,R,A)):-succ(R,SR).
aDec(c(L,R,SA),c(L,R,A)):-succ(A,SA).
```

Proposition

If a Motzkin tree is a skeleton of a closed lambda term then it exists at least one lambda binder on each path from the leaf to the root.

Closable and unclosable Motzkin trees

- there are slightly more unclosable Motzkin trees than closable ones as size grows:
- *closable:* *0,1,1,2,5,11,26,65,163,417,1086,2858,7599,20391,55127,150028,410719, ...*
- *unclosable:* *1,0,1,2,4,10,25,62,160,418,1102,2940,7912,21444,58507,160544,442748, ...*
- Possibly easy *open problem*: what happens to them asymptotically?

Generating closed affine terms

```
afLam(N,T):-sum_to(N,Hi,Lo),has_enough_lambdas(Hi),afLinLam(T,[],Hi,Lo)

has_enough_lambdas(c(L,_,A)):-succ(A,L).

afLinLam(v(X),[X])-->[].
afLinLam(l(X,A),Vs)-->lDec,afLinLam(A,[X|Vs]).
afLinLam(r(A),Vs)-->rDec,afLinLam(A,Vs).
afLinLam(a(A,B),Vs)-->aDec,{subset_and_complement_of(Vs,As,Bs)},
  afLinLam(A,As), % a lambda cannot go
  afLinLam(B,Bs). % to both branches !!!

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]).
```

Linear terms, skeletons of affine and linear terms

```
linLam(N,T):-N mod 3=:=1,
  sum_to(N,Hi,Lo),has_no_unused(Hi),
  afLinLam(T,[],Hi,Lo).

has_no_unused(c(L,0,A)):-succ(A,L).
```

Proposition

If a Motzkin tree with n binary nodes is a skeleton of a linear lambda term, then it has exactly $n+1$ unary nodes, with one on each path from the root to its $n+1$ leaves.

- affine: 0,1,2,3,9,30,81,242,838,2799,9365,33616,122937,449698
- linear: 0,1,0,0,5,0,0,60,0,0,1105,0,0,27120,0,0,828250

K-colored lambda terms and type inference

K-colored closed lambda terms

```
kColoredClosed(N,X):-kColoredClosed(X,[],N,0).

kColoredClosed(v(I),Vs)-->{nth0(I,Vs,V),inc_var(V)}.
kColoredClosed(l(K,A),Vs)-->l, % <= the K-colored lambda binder
  kColoredClosed(A,[V|Vs]),
  {close_var(V,K)}.
kColoredClosed(a(A,B),Vs)-->a,
  kColoredClosed(A,Vs),
  kColoredClosed(B,Vs).

l(SX,X):-succ(X,SX). % <= the size-consuming l/2 and a/2 predicates
a-->l,l.

inc_var(X):-var(X),!,X=s(_). % count new variable under the binder
inc_var(s(X)):-inc_var(X).

close_var(X,K):-var(X),!,K=0. % close and convert to ordinary numbers
close_var(s(X),SK):-close_var(X,K),succ(K,SK).
```

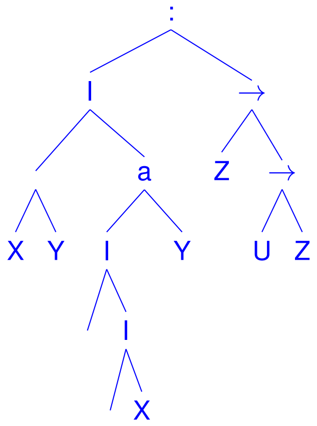
Examples of k-colored closed lambda terms

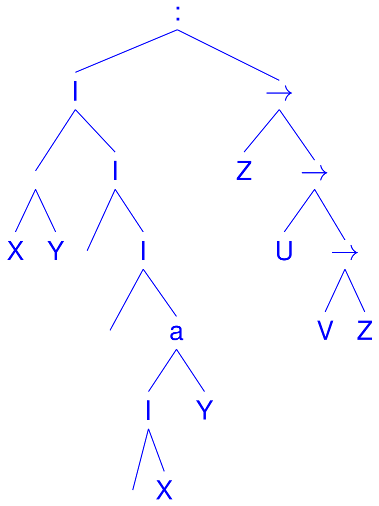
3-colored lambda terms of size 3, exhibiting colors 0,1,2.

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

- in a tree with *n* application nodes, the counts of k-colored lambdas must sum up to $n+1$
- $\Rightarrow$ we can generate a binary tree and then decorate it with lambdas satisfying this constraint
- the constraint holds for subtrees, recursively
- “open experiment”: can this mechanism reduce the amount of backtracking and accelerate term generation?

Two 3-colored lambda trees and their types





Type inference for k-colored terms

```
simplyTypedColored(N,X,T):-simplyTypedColored(X,T,[],N,0).

simplyTypedColored(v(X),T,Vss)-->{
    member(Vs:T0,Vss),
    unify_with_occurs_check(T,T0),
    addToBinder(Vs,X)
  }.
simplyTypedColored(l(Vs,A),S->T,Vss)-->l,
  simplyTypedColored(A,T,[Vs:S|Vss]),
  {closeBinder(Vs)}.
simplyTypedColored(a(A,B),T,Vss)-->a,
  simplyTypedColored(A,(S->T),Vss),
  simplyTypedColored(B,S,Vss).

addToBinder(Ps,P):-var(Ps),!,Ps=[P|_].
addToBinder([_|Ps],P):-addToBinder(Ps,P).

closeBinder(Xs):-append(Xs,[],_),!.
```

Counts for closed 2-colored simply-typed terms (i.e., affine)

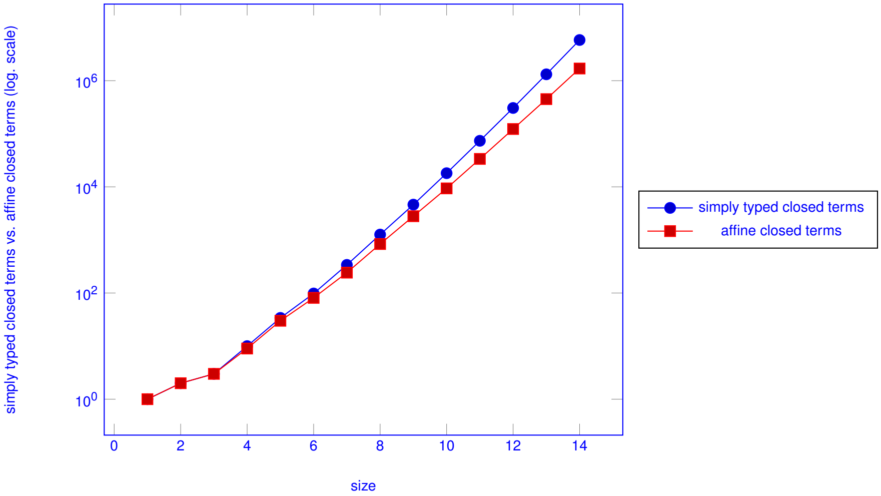


Figure: Simply typed closed terms and affine closed terms by increasing sizes

Type-holding and type-repelling skeletons

Skeletons of closed simply-typed terms

```
toSkels(v(_),v,v).
toSkels(l(Vs,A),l(K,CS),l(S)):-length(Vs,K),toSkels(A,CS,S).
toSkels(a(A,B),a(CA,CB),a(SA,SB)):-toSkels(A,CA,SA),toSkels(B,CB,SB).
```

generators for skeletons and k-colored skeletons by combining the generator `simplyTypedColored` with `toSkeleton`

```
genTypedSkels(N,CS,S):-genTypedSkels(N,_,_,CS,S).

genTypedSkels(N,X,T,CS,S):-simplyTypedColored(N,X,T),toSkels(X,CS,S).

typableColSkels(N,CS):-genTypedSkels(N,CS,_).

typableSkels(N,S):-genTypedSkels(N,_,S).
```

Are colors growing proportional to the log of their type-sizes?

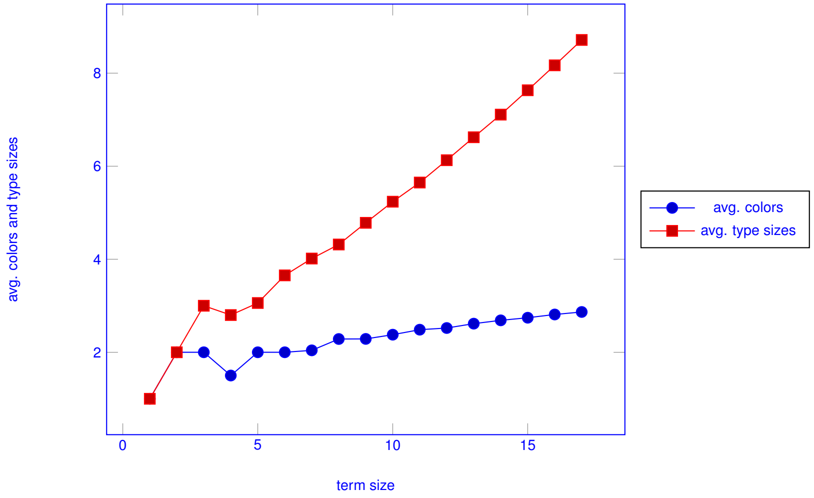


Figure: Growth of colors and type sizes

a most colorful term: reaches the maximum number of colors

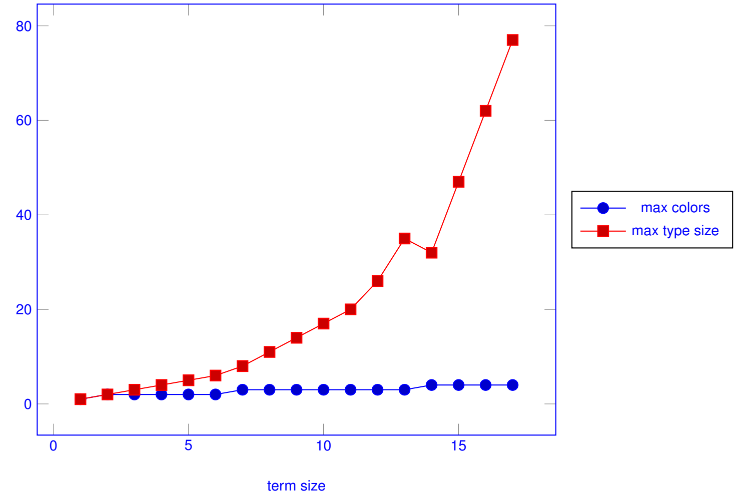


Figure: Colors of a most colorful term vs. its maximum type size

Does a most colorful term reach max type size?

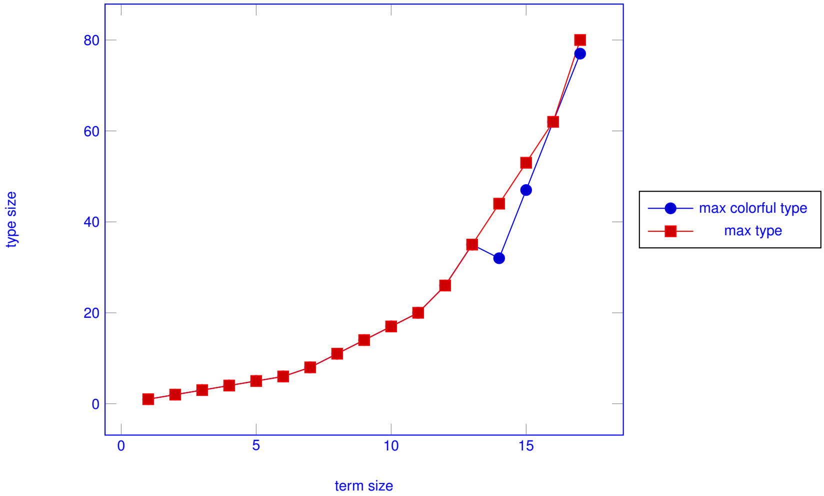


Figure: Largest type size of a most colorful term vs. largest type size

Type-holding and type-repelling Motzkin trees

- a *type-holding Motzkin tree* is one for which it exists a simply-typed closed term having it as its skeleton
- a *type-repelling Motzkin tree* is one for which no simply-typed closed term exists having it as its skeleton
- to efficiently generate these skeletons we will split the generation of lambda terms in two stages
  - *the first stage* will generate the unification equations that need to be solved for type inference as well as the ready to be filled out lambda trees
  - it is convenient to actually generate “on the fly” the code to be executed in the second stage
  - *the second stage* will just use Prolog’s metacall to activate this code

Generating the executable equation set

we generate a ready to run conjunction of unification constraints

```
genEqs(N,X,T,Eqs):-genEqs(X,T,[],Eqs,true,N,0).

genEqs(v(I),V,[V0|Vs],Es1,Es2)-->{add_eq(Vs,V0,V,I,Es1,Es2)}.
genEqs(l(A),(S->T),Vs,Es1,Es2)-->l,genEqs(A,T,[S|Vs],Es1,Es2).
genEqs(a(A,B),T,Vs,Es1,Es3)-->a,
  genEqs(A,(S->T),Vs,Es1,Es2),
  genEqs(B,S,Vs,Es2,Es3).

% solve this equation as it can either succeed once, or fail
add_eq([],V0,V,0,Es,Es):-unify_with_occurs_check(V0,V). % <==
add_eq([V1|Vs],V0,V,I,(el([V0,V1|Vs],V,0,I),Es),Es).

el(I,Vs,V):-el(Vs,V,0,I).

el([V0|_],V,N,N):-unify_with_occurs_check(V0,V).
el([_|Vs],V,N1,N3):-succ(N1,N2),el(Vs,V,N2,N3).
```

Efficient generation of type-holding and type-repelling skeletons

- we solve the equations for 2-colored terms to avoid backtracking
- Motzkin trees: `1,1,2,4,9,21,51,127,323,835,2188`
- Type equation trees: `0,1,1,1,5, 9,17, 55,122,289, 828`
- $\Rightarrow$ we can generate efficiently type-holding and type-repelling skeletons
  - type repelling: `Eqs` have no solution (their negation succeeds)
  - type holding: `Eqs` have at least one solution (no need to compute all)

```
untypableSkel(N,Skel):-genEqs(N,X,_,Eqs),not(Eqs),toMotSkel(X,Skel).

typableSkel(N,Skel):-genEqs(N,X,_,Eqs),once(Eqs),toMotSkel(X,Skel).
```

Are there uniquely typable skeletons?

- we compute efficiently such skeletons by generating the decoration code `Eqs` that we constrain to have exactly one solution when activated

```
uniquelyTypableSkel(N,Skel):-
  genEqs(N,X,_,Eqs),succeeds_once(Eqs),Eqs,
  toMotSkel(X,Skel).

succeeds_once(G):-findnsols(2,_,G,Sols),!,Sols=[_].
```

- not very many: 0,1,0,0,2,0,1,7,1,13,34,20,100,226,234
- open problem: *prove that they exist for all sizes*

Growth of uniquely typable skeletons

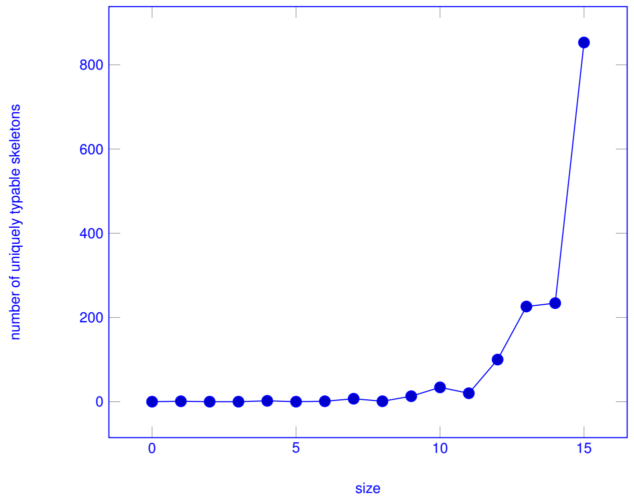
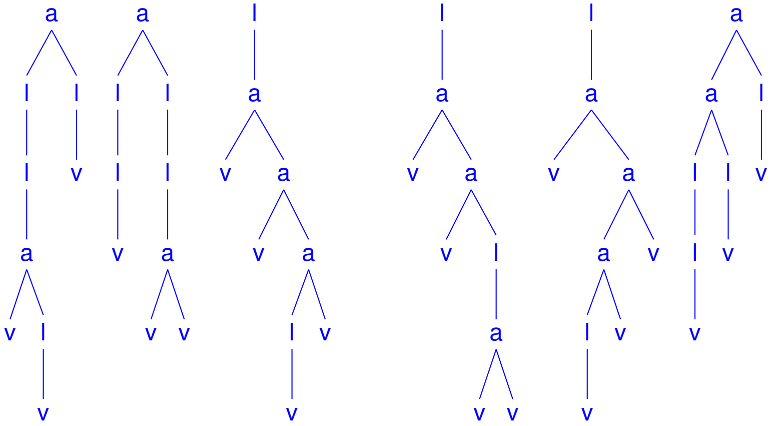


Figure: 3 Growth of the number of uniquely typable skeletons

3 type-holding and 3 type-repelling Motzkin trees



Growth of the number of skeletons is similar on a log-scale

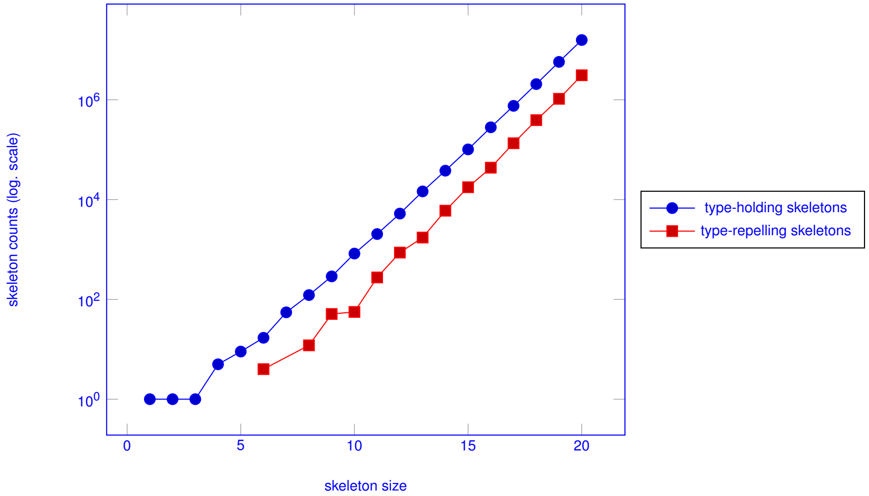


Figure: type-holding vs. type-repelling skeletons up to size 20

Counts of type repelling and type holding Motzkin skeletons

term size	type holding skeletons	type repelling skeletons
0	0	0
1	1	0
2	1	0
3	1	0
4	5	0
5	9	0
6	17	4
7	55	0
8	122	12
9	289	51
10	828	56
11	2037	275
12	5239	867
13	14578	1736
14	37942	5988
15	101307	17697
16	281041	43583
17	755726	134546
18	2062288	390872
19	5745200	1045248
20	15768207	3102275

Figure: Number of type-holding and type repelling skeletons

Growth rates of type-holding and type-repelling skeletons

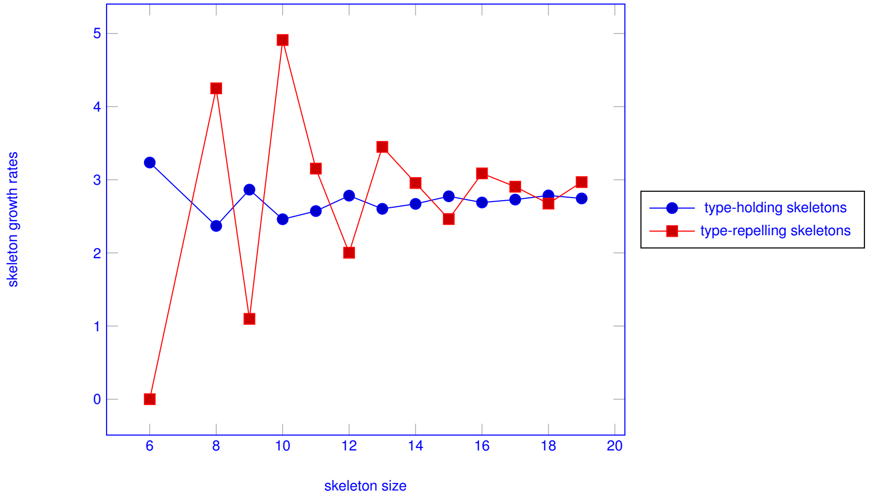


Figure: The similar growth rates of type-holding and type-repelling skeletons

Motzkin trees vs. their subset of type-holding skeletons

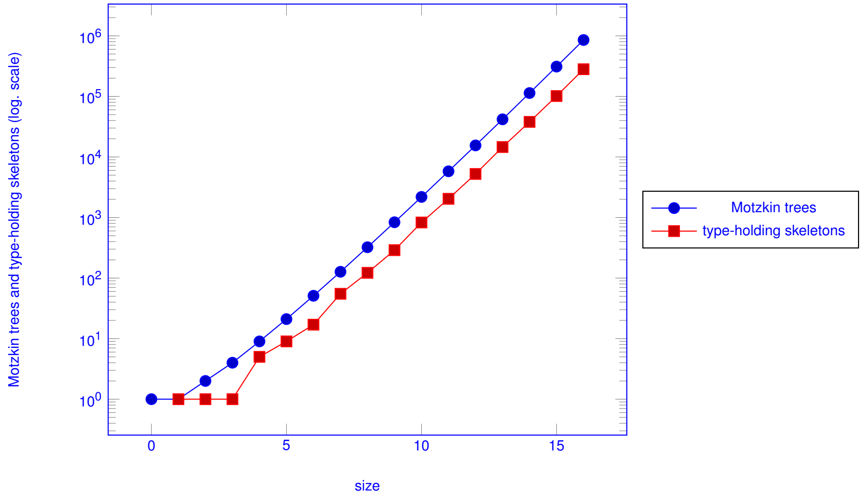


Figure: Counts for Motzkin trees and type-holding skeletons for increasing term sizes

Counts of simply typed closed terms vs. their skeletons

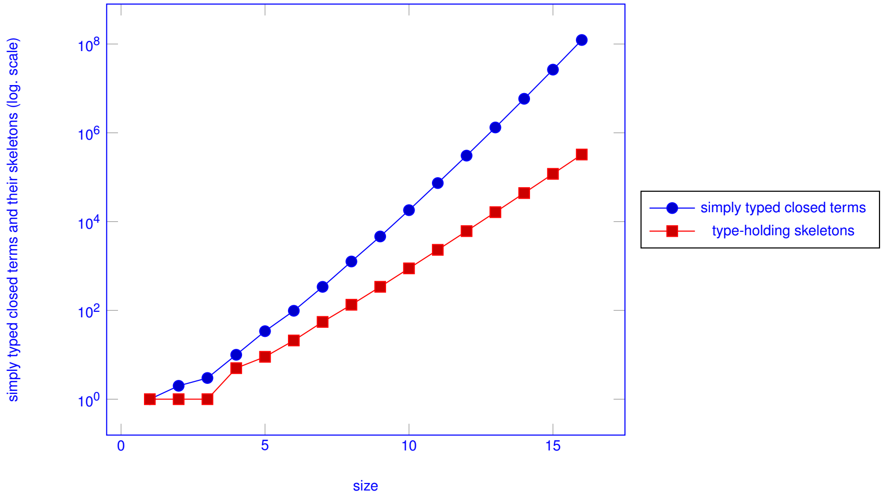


Figure: Counts of simply typed closed terms and their skeletons by increasing sizes

An application to random lambda term generators

An application to random lambda term generators

- Rémy's algorithm: exact size uniformly random binary trees
- binary trees - bijection to 2-colored Motzkin trees
- decorating 2-colored Motzkin trees to lambda terms

Revisiting Rémy's algorithm, declaratively

- émy's original algorithm [7] grows binary trees by grafting new leaves with equal probability for each node in a given tree
- an elegant procedural implementation is given in [**?**] as algorithm **R**, by using destructive assignments in an array representing the tree
- we will work with edges instead of nodes and graft new edges at each step
- a two stage algorithm: first sets of edges, then trees represented as Prolog terms
- some "magic with logic variables"
- while the algorithm handles terms with thousands of nodes its average performance is slightly above linear as it takes time proportional to the size of the set of edges to pick the one to be expanded

Same code for all-terms and random term generation

- initial set of two edges

```
remy_init([e(left,A,_),e(right,A,_)]).
```

- same code for all-term or random-term generation
- except for defining a random choice of the edges or a backtracking one, by replacing `choice_of/2` with its commented out alternative

```
left_or_right(I,J):-choice_of(2,Dice),left_or_right(Dice,I,J).

choice_of(N,K):-K is random(N).
% choice_of(N,K):-N>0,N1 is N-1,between(0,N1,K).

left_or_right(0,left,right).
left_or_right(1,right,left).
```

The grafting step

- we can grow a new edge by “splitting an existing edge in two”

```
grow(e(LR,A,B), e(LR,A,C),e(I,C,_),e(J,C,B)):-left_or_right(I,J).
```

- we add three new edges corresponding to arguments `2`, `3` and `4`
- we remove one, represented as the first argument
- contrary to Rémy’s original algorithm, our tree grows “downward” as new edges are inserted at the target of existing ones

Finding the spot where we graft

- chose the grafting point's position

```
remy_step(Es,NewEs,L,NewL):-NewL is L+2,
  choice_of(L,Dice),
  remy_step1(Dice,Es,NewEs).
```

- perform the graft

```
remy_step1(0,[U|Xs],[X,Y,Z|Xs]):-grow(U, X,Y,Z).
remy_step1(D,[A|Xs],[A|Ys]):-D>0,
  D1 is D-1,
  remy_step1(D1,Xs,Ys).
```

The iteration to desired size

- The predicate `remy_loop` iterates over `remy_step` until the desired $2K$ size is reached, in K steps
- we grow by `2` edges at each step, 2K edges total

```
remy_loop(0,[],N,N).
remy_loop(1,Es,N1,N2):-N2 is N1+2,remy_init(Es).
remy_loop(K,NewEs,N1,N3):-K>1,K1 is K-1,
  remy_loop(K1,Es,N1,N2),
  remy_step(Es,NewEs,N2,N3).
```

- an example of output

```
?- remy_loop(2,Edges,0,N).
Edges = [e(left, A, B), e(right, A,C),
         e(right,C,D), e(left,C,E)],
N = 4.
```

From sets of edges to trees as Prolog terms

- the predicate `bind_nodes/2` iterates over edges, and for each internal node it binds it with terms provided by the constructor c/2, left or right, depending on the type of the edge

```
bind_nodes([],e).
bind_nodes([X|Xs],Root):-X=e(_,Root,_),
  maplist(bind_internal, [X|Xs]),
  maplist(bind_leaf, [X|Xs]).
```

- bind an internal node with constructor `c/2`

```
bind_internal(e(left,c(A,_),A)).
bind_internal(e(right,c(_,B),B)).
```

- bind a leaf node with the constant `v/0`

```
bind_leaf(e(_,_,Leaf)):-Leaf=e->true;true.
```

Running the algorithm

- the predicate `remy_term/2` puts the two main steps together

```
remy_term(K,B):-remy_loop(K,Es,0,_),bind_nodes(Es,B).
```

- uniformly random generation of a random term with 4 internal nodes and timings for a large tree:

```
?- remy_term(4,T).
T = c(c(e, e), c(c(e, e), e)) .
?- time(remy_term(1000,_)).
526,895 inferences, 0.066 CPU in 0.078s (7,995,978 Lips)
```

- we obtain a 2-Motzkin tree generator from the binary tree generator via the bijection given by the (bi-directional) predicate `cat_mot/2`

```
mot_gen(N,M):-N>0,remy_term(N,C),cat_mot(C,M).
```

- this is unlikely to be a uniformly random generator as `l/1` nodes cover all colors except color `0` covered by `r/1`, and, via the bijection, the total count is a Catalan number rather than a Motzkin number

Decorating Motzkin trees to lambda terms

- if one wants uniformly random Motzkin trees a Boltzmann sampler or one based on via holonomic equations can be used
- we can mimic (actually in a stronger way) the ideas behind the “natural size” that favors variables closer to their binders
- one can build for each list of binders from a leaf to the root, a distribution that decays exponentially with each step

```
nat2probs(0,[]).
nat2probs(N,Ps):-N>0,Sum is 1-1/2^N,Last is 1-Sum,Inc is Last/N,
  make_probs(N,Inc,1,Ps).
```

- at each step, the probability to continue further is reduced to half

```
make_probs(0,_,_,[]).
make_probs(K,Inc,P0,[P|Ps]):-K>0,K1 is K-1,P1 is P0/2, P is P1+Inc
  make_probs(K1,Inc,P1,Ps).
```

The decoration algorithm

- given a Motzkin tree, we decorate each leaf `v/0` by turning it into a natural number representing a de Bruijn index
- the value of the de Bruijn index is determined for each leaf independently by walking up on the chain of lambda binders with decaying probabilities

```
decorate(v,0,v(X))-->[X]. % free variable
decorate(v,N,K)-->{N>0,nat2probs(N,Ps),walk_probs(Ps,0,K)},[].
decorate(l(X),N,l(Y))-->{N1 is N+1},decorate(X,N1,Y).
decorate(r(X),N,r(Y))-->decorate(X,N,Y).
decorate(a(X,Y),N,a(A,B))-->decorate(X,N,A),decorate(Y,N,B).
```

Plain lambda terms are generated as follows:

```
plain_gen(N,T,FreeVars):-mot_gen(N,B),decorate(B,0,T,FreeVars,[]).
```

Retrying for closed terms

- to ensure that a term is closed, we restarts until the list of free variables is empty
- we also returning the number of retries

```
closed_gen(N,T,I):- Lim is 100+2^min(N,24),try_closed_gen(Lim,0,I,N
```

- restarts are managed by the predicate `try_closed_gen/5`, which, when the decorated term is not closed, tries generating a new term

```
try_closed_gen(Lim,I,J,N,T):- I<Lim,
  ( mot_gen(N,B),decorate(B,0,T,[],[])*->J=I
  ; I1 is I+1, try_closed_gen(Lim,I1,J,N,T)
  ).
```

Example of generation of a random closed term

- random closed lambda terms obtained by decorating motzkin trees.

```
?- closed_gen(10,T,I).
T = l(l(l(r(r(r(a(l(2), 2))))))),
?- closed_gen(2000,_,I).
I = 3 .
```

- with a size definition that counts variables as being of size `0` or `1`, these lambda terms are actually of exactly the same size as their Motzkin tree skeletons

Generating random simply typed terms

- as before, we decorate the 2-Motzkin trees with de Bruijn indices
- we interleave the decoration process with early rejection of types that do not unify or de Bruijn indices that lead to terms that are not closed
- interesting size definitions depend mostly on the weight we attach to the de Bruijn indices
- $\Rightarrow$ we customize the code to “plug-in” a size definition of our choice
- in fact, one can use statistics from real programs to mimic any distribution of the de Bruijn indices

```
linChoice(K,Ts,I,T0):-K>0,I is random(K),nth0(I,Ts,T0).

expChoice(K,Ts,I,T):-K>0,N is 2^(K-1),
  R is random(N),N1 is N>>1,
  expChoice1(N1,R,Ts,T,0,I).

expChoice1(N,R,[X|_],Y,I,I):-R>=N,!,Y=X.
expChoice1(N,R,[_|Xs],Y,I1,I3):-N1 is N>>1,succ(I1,I2),
  expChoice1(N1,R,Xs,Y,I2,I3).
```

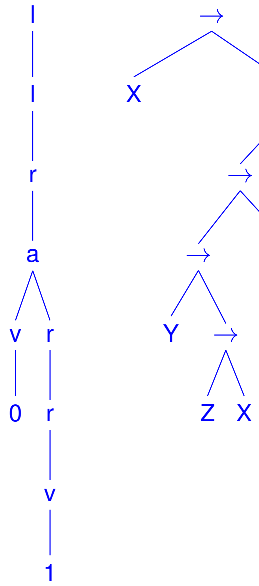
The random simply-typed closed lambda term generator

```
decorateTyped(M,X,T):-decorateTyped(M,X,T,0,[]).

decorateTyped(v,v(I),T,K,Ts):-
  linChoice(K,Ts,I,T0), % <= plug-in here the size definition!
  unify_with_occurs_check(T,T0).
decorateTyped(l(X),l(A),(S->T),N,Ts):-succ(N,SN),
  decorateTyped(X,A,T,SN,[S|Ts]).
decorateTyped(r(X),r(A),(_->T),N,Ts):-
  decorateTyped(X,A,T,N,Ts).
decorateTyped(a(X,Y),a(A,B),T,N,Ts):-
  decorateTyped(X,A,(S->T),N,Ts),
  decorateTyped(Y,B,S,N,Ts).

ranTyped(N,MaxI,MaxJ,X,T,I,J):-
  between(1,MaxI,I),
    mot_gen(N,Mot),%ppp(I=Mot),
    between(1,MaxJ,J),
      decorateTyped(Mot,X,T),
  !.
```

Example of random simply typed term of natural size 8



steps(1*2) natsize(8)+heapsize(7)+tsize(6)

Running the random simply-typed term generator

- Can we generate terms of natural size 1000?

```
?- ranTyped(400).
% 11,982,837 inferences, 1.170 CPU in 1.181 seconds (99% CPU, 1024

... big boring term here ...

steps(64*43)
natsize(486)+heapsize(399)+tsize(146)

?- ranTyped(400).
% 122,666,661 inferences, 11.919 CPU in 12.005 seconds (99% CPU, 1

... big boring term here ...

steps(644*10)
natsize(1162)+heapsize(399)+tsize(21)
```

Related work

Bacher, A., Bodini, O., Jacquot, A.:
Exact-size Sampling for Motzkin Trees in Linear Time via Boltzmann Samplers and Holonomic Specification.
In Nebel, M.E., Szpankowski, W., eds.: 2013 Proceedings of the Tenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO), SIAM (2013) 52–61

Tarau, P.:
A hiking trip through the orders of magnitude: Deriving efficient generators for closed simply-typed lambda terms and normal forms.
CoRR **abs/1608.03912** (2016)

Bendkowski, M., Grygiel, K., Tarau, P.:
Boltzmann samplers for closed simply-typed lambda terms.
In Lierler, Y., Taha, W., eds.: Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings. Volume 10137 of Lecture Notes in Computer Science., Springer (2017) 120–135

Deutsch, E., Shapiro, L.W.:
A bijection between ordered trees and 2-motzkin paths and its many consequences.
Discrete Mathematics **256**(3) (2002) 655 – 670

Lescanne, P.:
Quantitative aspects of linear and affine closed lambda terms.
CoRR **abs/1702.03085** (2017)

Tarau, P.:
On Logic Programming Representations of Lambda Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization.
In Pontelli, E., Son, T.C., eds.: Proceedings of the Seventeenth International Symposium on Practical Aspects of Declarative Languages PADL'15, Portland, Oregon, USA, Springer, LNCS 8131 (June 2015) 115–131

Rémy, J.L.:
Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire.
RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications **19**(2) (1985) 179–195

Conclusions

- we have introduced abstraction mechanisms that “forget” properties of the difficult class of simply-typed lambda terms, to reveal interesting structural properties
- k-colored terms subsume linear and affine terms and are likely to be usable to fine-tune random generators to more closely match color-distributions in lambda terms representing real programs
- type-repelling skeletons might be useful for efficient algorithms built on avoiding all “small” type-repelling skeletons stored in a database
- the new bijection between binary terms and 2-colored Motzkin terms and the generation algorithms centered on the distinction between free and binding lambda constructors has been useful to accelerate generation of affine and linear terms and random terms via Rémy’s algorithm
- the tools used: a language as simple as (mostly) Horn Clause Prolog can handle elegantly combinatorial generation problems when the synergy between sound unification, backtracking and DCGs is put at work

Questions?

slides: `http://www.cse.unt.edu/~tarau/research/2017/slides_cla17.pdf`

draft paper: `http://www.cse.unt.edu/~tarau/research/2017/repel.pdf`

code: `http://www.cse.unt.edu/~tarau/research/2017/repel.pro`

Picasso:
Questions tempt you to tell lies, particularly when there is no answer.