On 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
paul.tarau@unt.edu

ABSTRACT

Lambda terms in de Bruijn notation are Motzkin trees (also called binary-unary trees) with indices at their leaves counting up to a binder among the lambdas on the path to the root labeling their leaves. Define the *skeleton of a lambda term* as the Motzkin tree obtained by erasing the de Bruijn indices labeling their leaves.

Then, given a Motzkin tree, one can ask if it is the skeleton of at least one closed lambda term. More interestingly, one can ask the same question for simply-typed closed terms. 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. We study some of their statistical properties with focus on the relative density of various classes of terms.

As a generalization of *affine lambda terms*, we introduce *k-colored lambda terms* obtained by labeling their unary nodes with counters for the variables they bind. This leads to a study their *k-colored skeletons* and their statistical properties.

We design a declarative implementation of Rémy's algorithm and define a new *bijection from binary trees to 2-colored Motzkin trees* from where we derive simply-typed closed lambda terms by decorating them with de Bruijn indices. The resulting sequential and parallel random simply-typed closed lambda term generators produce terms above sizes of **1000** and **2000**, an order of magnitude above the best previously known results.

The paper is structured as a literate Prolog program to facilitate an easily replicable, concise and declarative expression of our concepts and algorithms.

KEYWORDS

simply-typed closed lambda terms, Motzkin trees, bijections between data types, generation of all terms of a given size, generation of random lambda terms, type inference, unification, logic variables, parallel algorithms

1 INTRODUCTION

Lambda terms, in de Bruijn notation [9], can be seen as Motzkin-trees built of unary lambda nodes, binary application nodes and variables at their leaves, labeled with de Bruijn indices pointing toward their lambda binder.

This brings up some natural questions about the underlying tree structure of specific families of lambda terms. Can we classify lambda nodes by inverting the function from indices at the leaves to their binders? Will this result in interesting bijections to simpler data types, e.g., members of the Catalan family of combinatorial objects? Can we single out Motzkin trees as *skeletons*, that, because of some observable structure, can or cannot hold a given family of terms?

We will see that the answers are trivial for some families, e.g., in the sense that any such skeleton can be decorated to at least one lambda term that is a member of the family, but it suggests investigating if it can expose some interesting structural properties for others, leading to algorithms connecting with cases that are known to be notoriously hard.

This brings us to the main focus of this paper, the case of simply-typed terms and some of their sub-families like affine or linear terms.

Despite the vanishing asymptotic density of simply-typed lambda terms [15], their all-term and random-term generation has been speeded-up significantly by the use of Prolog-based algorithms that interleave generation and type-inference steps [23, 5]. However, the structure of simply-typed lambda terms has so far escaped handling by analytical methods. Basic combinatorial properties like counts for terms of a given size have been obtained so far only by generating all terms, or, as in [5], by mimicking their exhaustive generation with a recursive structure that, while omitting the actual lambda terms, keeps the type-inference mechanism intact.

These difficulties suggest to start by laying down some empirical background, hoping that it will help us learn more about the structure of simply-typed lambda terms, and as a practical outcome, push further all-term and random-term generation, including via better parallelization opportunities.

A useful distinction can be made between lambda constructors that bind variables and those that do not. Among other benefits, distinguishing them will make the analysis of linear and affine terms simpler and put their skeletons, the 2-colored Motzkin trees, in bijection with the well-known Catalan family of combinatorial objects. This bijection will also enable fast algorithms for generating random binary trees to be transmigrated into 2-Motzkin trees and then lambda terms.

The separation of the generation process into stages allows will allow on-the fly code generation that speeds-up both sequential and parallel processing.

The rest of the paper is organized as follows. Section 2 describes a new bijection between binary trees and 2-colored Motzkin trees.

*

PPDP'2017,
 978-x-xxxx-xxxx-x/YY/MM. . . $0.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Section 3 discusses the case of closed, linear and affine lambda terms. Section 4 introduces type-holding and type-repelling Motzkin trees and studies some empirical properties related to their density and growth. Section 5 describes a random-term generator combining a uniformly random 2-Motzkin-tree generator and its decoration into a simply-typed lambda term, via Remy's algorithm and a bijection to binary trees. Section 6 describes a parallel algorithm for speeding up random term generators and illustrates their performance gains. Section 7 overviews related work and section 8 concludes the paper.

The paper is structured as a literate Prolog program to facilitate an easily replicable, concise and declarative expression of our concepts and algorithms. The code extracted from the paper is available at: http://www.cse.unt.edu/~tarau/research/2017/repel.pro, tested with SWI-Prolog [24] version `7.5.3`.

2 A BIJECTION BETWEEN 2-COLORED MOTZKIN TREES AND BINARY TREES

A *Motzkin tree* (also called binary-unary tree) is a rooted ordered tree built from binary nodes, unary nodes and leaf nodes. A *k-colored Motzkin tree* is obtained by labeling its unary nodes with colors from a set of `k` elements.

We define *2-colored Motzkin trees* (shortly *2-Motzkin trees*) as the free algebra generated by the constructors `v/0`, `l/1`, `r/1` and `a/2`.

We define 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).

Thus, we can see lambda terms in de Bruijn form as Motzkin trees with leaves labeled with natural numbers. We interpret the labels as pointing to their lambda binder on a path to the root of the tree. If each leaf reaches via this its de Bruijn index at least one unary constructor, we call the term closed, otherwise we call it *plain*.

We observe that the constructors marking lambdas may have at least one de Bruijn index pointing to them or have none. Thus, we classify our unary constructors into:

- *binding lambdas*, that are reached by at least one de Bruijn index (denoted `l/1`)
- *free lambdas* that cannot be reached by any de Bruijn index, denoted `r/1`.

We can think about these as *2-colored lambda terms*.

We define the *2-colored Motzkin skeleton of a lambda term* (shortly *skeleton*) as the tree obtained by erasing the de Bruijn indices labeling their leaves.

It is well-known that 2-Motzkin trees are counted by the Catalan numbers and several bijections between them to members of the Catalan family of combinatorial objects have been identified in the past [10]. We will introduce here a new one that is defined inductively in a "compositional way", based on a mapping between small tree components on the two sides.

We describe binary trees as the free algebra generated by the constructors `e/0` and `c/2`. Binary trees are a well known member of the Catalan family of combinatorial objects. Our bijection can be seen as connecting any other member of this family to 2-colored Motzkin trees (from now on, 2-Motzkin trees).

We define the bijection between non-empty binary trees and 2-Motzkin trees simply by encoding each of the nodes `v/0`, `l/1`, `r/1` and `a/2` by a unique small binary tree as shown by the reversible bidirectional predicate `cat_mot/2`, with the binary tree as its first argument and the 2-Motzkin tree as its second.

```
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).
```

Proposition 1. The predicate `cat_mot/2` defines a bijection between non-empty binary trees and Motzkin trees.

PROOF. It follows by structural induction by observing that the 4 clauses cover via disjoint unification patterns all the 4 possible tree shapes matched one-to-one on the two sides. □

Example 1. We illustrate The bidirectional Prolog predicate `cat_mot/2` with the two trees also shown in Fig. 1, together with two larger trees on the right side, "twinned" in a similar way, Motzkin-tree on the left, binary tree on the right.

```
?- cat_mot(BinTree,a(l(v),r(v))),cat_mot(BinTree,MotTree).
BinTree = c(c(c(e, e), e), c(e, c(e, e))),
MotTree = a(l(v), r(v)) .
```

One can test it also with input from the following simple binary tree generator `cat(N,T)` which, given a natural number `N` returns a tree `X` of size `N`, assuming a size definition that counts each internal node as `1`.

```
cat(N,X):-cat(X,N,0).

cat(e,N,N).
cat(c(A,B),SN,N3):-succ(N1,SN),cat(A,N1,N2),cat(B,N2,N3).
```

Note the use of the bidirectional `succ/2` built-in, which also tests for being larger than 0, when working as predecessor.

3 CLOSED, AFFINE AND LINEAR TERMS

We can define a lambda term in de Bruijn form as a Motzkin tree decorated with natural numbers at its leaves. With a *size definition* (assumed here), that gives `2` units to binary constructors, `1` unit to unary constructors and `0` units to the leaves of the tree, a lambda term and its skeleton can be, conveniently, seen as having the same size, in fact corresponding (up to a constant factor) to its *heap representation* in the runtime system of all programming languages we know of.

Semantically, the labels are understood as pointing to a unary node seen as a lambda binder on a the path to the root, starting with 0 for the closest one.

We define the *skeleton* of a lambda tree in de Bruijn form as the 2-Motzkin tree obtained by erasing the de Bruijn indices at its leaves.

Thus a lambda term is built with the constructors `a/2` representing applications, `l/1` and `r/1` representing lambda nodes and natural numbers marking leaves (possibly wrapped as `v/1` nodes, when convenient).

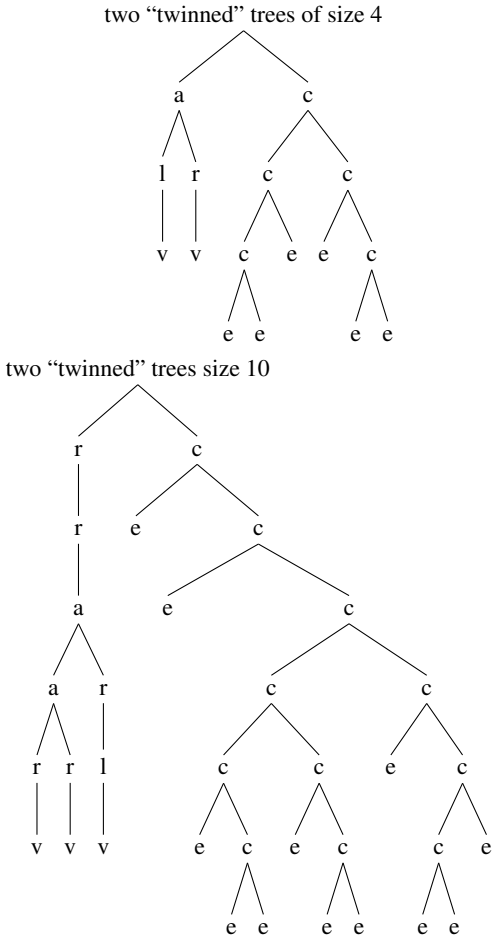


Figure 1: The 2-colored Motzkin trees to non-empty binary trees bijection

A 2-Motzkin tree is built with `a/2` representing binary nodes, `l/1` and `r/1` representing unary nodes and `v/0` standing for leaf nodes. Thus we compute a skeleton by replacing the de Bruijn indices at the leaves of a lambda term with the constant `v/0`.

When generating trees of a given size, with several node constructors, it makes sense to have separate counters for each.

The predicate `sum_to/2` maintains such counters for nodes of types `l/1`, `r/1` and `a/2`.

```
sum_to(N,c(L,R,A),c(0,0,0)):-N>=0,
  between(0,N,A2),0=:=A2/\1,A is A2>>1,
  LR is N-A2,
  between(0,LR,L),
  R is LR-L.
```

The predicates (suggestively named) `lDec/2, rDec/2` and `aDec/2` define single steps consuming one available unit of size for each of the corresponding constructors. Note the use of the bidirectional built-in predicate `succ/2` that computes in this case the predecessor of a natural number and fails after reaching `0`.

```
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).
```

We will start with generators for closed, affine and linear terms.

As analytic methods are known for computing counts for closed terms as well as closed affine and linear terms [16], we will focus here on some simple properties of their skeletons and on their efficient generators.

3.1 Closed lambda terms

A lambda term in de Bruijn form is closed, if for each of its de Bruijn indices, there is a lambda binder to which it points, on the path to the root of the tree representing the term. We call a Motzkin tree *closable* if it is the skeleton of at least one closed lambda term.

It immediately follows that:

Proposition 2. 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.

There are slightly more unclosable Motzkin trees than closable ones as size grows:

number of closable terms of sizes 0,1,2,... :
0,1,1,2,5,11,26,65,163,417,1086,2858,7599,20391,55127,150028,410719, ...
number of unclosable terms of sizes 0,1,2,... :
1,0,1,2,4,10,25,62,160,418,1102,2940,7912,21444,58507,160544,442748, ...

We leave as an *open problem* to study what happens to them asymptotically.

3.2 Closed affine lambda terms

An *affine lambda term* has one or zero variables bound by each lambda constructor.

Proposition 3. If a 2-Motzkin tree with n binary nodes is a skeleton of an affine lambda term, then it has exactly $n+1$ unary `l` nodes, with at least one on each path from the root to its $n+1$ leaves.

This suggest generators that separate unary and binary node counts for the skeletons and enforce this constraint on their respective sizes.

The predicate `afLam/2`, follows closely the one described in detail in [21], except that it handles `l/1` and `r/1` as separate cases.

```
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).
```

The predicate `has_enough_lambdas/1` is used to express the constraint that the number of application nodes `a/2` should be one more than the number of `l/1` constructors (in bijection with the leaves they bind). The predicate `afLinLam/4` is defined via Definite Clause Grammars (DCGs) that encapsulate the consumption of

the size units[1]. It uses the predicate `subset_and_complement_of/3` to direct each lambda binder on either a left or a right path at an application node.

```
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),
  afLinLam(B,Bs).

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]).
```

Erasure of de Bruijn indices turns a 2-colored lambda term into a 2-colored Motzkin tree.

```
toMotSkel(v(_),v).
toMotSkel(l(X),l(Y)):-toMotSkel(X,Y).
toMotSkel(l(_,X),l(Y)):-toMotSkel(X,Y).
toMotSkel(r(X),l(Y)):-toMotSkel(X,Y).
toMotSkel(a(X,Y),a(A,B)):-toMotSkel(X,A),toMotSkel(Y,B).
```

The predicates `afSkelGen/2` and `linSkelGen/2` transform the generator for lambda terms into generators for their skeletons.

```
afSkelGen(N,S):-afLam(N,T),toMotSkel(T,S).

linSkelGen(N,S):-linLam(N,T),toMotSkel(T,S).
```

The multiset of skeletons is trimmed to a set of unique skeletons using SWI-Prolog's `distinct/2` built-in.

```
afSkel(N,T):-distinct(T,afSkelGen(N,T)).

linSkel(N,T):-distinct(T,linSkelGen(N,T)).
```

3.3 Closed linear lambda terms

Proposition 4. 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.

```
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).
```

Note the use of the predicate `has_no_unused/1` that expresses, quite concisely, the constraints that `r/1` nodes should not occur in the term and that the set of `l/1` nodes should be in a bijection with the set of leaves.

It is immediate that closed affine and linear terms are well-typed. The unary nodes of the skeletons of affine term can be seen as having 2 colors, `l/1` and `r/1`. This suggest to investigate next the case of k-colored terms.

[1] Functional programmers might notice here the analogy with the use of monads encapsulating state changes with constructs like Haskell's do notation.

3.4 K-colored simply-typed closed lambda terms

As a natural generalization derived from k-colored Motzkin trees, we define a *k-colored lambda term* having as its lambda constructor `l/1` labeled with the number of variables that it binds. Thus an affine term is a 2-colored lambda term.

The predicate `kColoredClosed/2` generates terms while partitioning lambda binders in k-colored classes. It works by incrementing the count of leaf variables a lambda binds, in a "backtrackable way" by using successor arithmetic with the deepest node kept as a free logical variable at each step.

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

kColoredClosed(v(I),Vs)-->{nth0(I,Vs,V),inc_var(V)}.
kColoredClosed(l(K,A),Vs)-->l,
  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).
a-->l,l.

inc_var(X):-var(X),!,X=s(_).
inc_var(s(X)):-inc_var(X).

close_var(X,K):-var(X),!,K=0.
close_var(s(X),SK):-close_var(X,K),l(SK,K).
```

Note also the DCG-mechanism that controls the intended size of the terms via the (conveniently named) predicates `l/2` and `a/2` that decrement available size by 1 and respectively 2 units.

Example 2. 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))) .
```

Given a tree with n application nodes, the counts for all k-colored lambdas in it must sum up to $n+1$. Thus we can generate a binary tree and then decorate it with lambdas satisfying this constraint. Note that the constraint holds for subtrees, recursively. We leave it as an "open experiment" to find out if this mechanism can reduce the amount of backtracking and accelerate term generation.

3.5 Type inference for k-colored terms

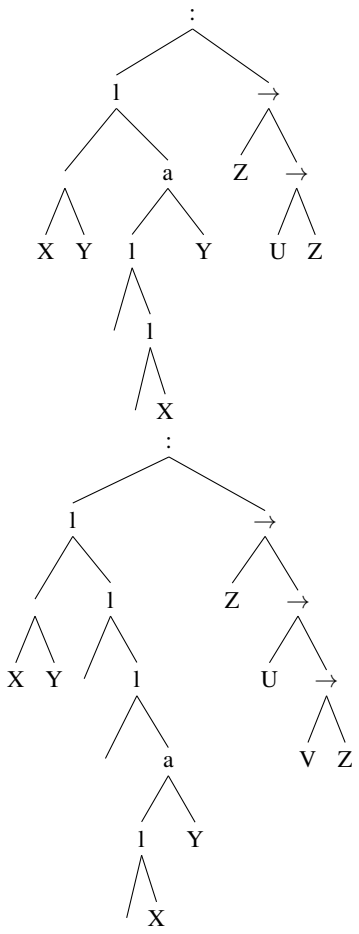
The study of the combinatorial properties of simply-typed lambda terms is notoriously hard. In practice, the two most striking things when inferring types that one might notice are

- *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.

We will follow the interleaving of term generation, checking for closedness and type inference steps shown in [19], but enhance it to also identify variables covered by each lambda binder. In fact, given the surjective function $f : V \to L$ that associates to each leaf variable in a closed lambda term its lambda binder, one can compute the set $f^{-1}(l)$ for each $l \in L$, expressing which variables are mapped to each binder.

Example 3. We illustrate two 2-colored simply typed terms with lambda nodes shown as `l/2` constructors having as their left child a multi-way tree collecting the set of variables that it binds. We place the inferred type as the right child of a "root" labeled with ":".



As in [19], our type inference algorithm ensures that variables under the same binder agree on their type via unification with occurs check, to avoid formation of cycles in the types, represented as binary trees with internal nodes "`->/2`" and logic variables as leaves.

```
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).
```

Note that `addToBinder/2` adds each leaf under a binder to the open end of the list of variable/type pairs list, closed by `closeBinder/1`.

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

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

Example 4. Some terms of size 5 generated by the predicate `simplyTypedColored/3` and their types.

```
?- simplyTypedColored(5,Term,Type).
Term = l([], l([], l([], l([], l([A], v(A))))),
Type =  (B->C->D->E->F->F) ;
...
Term = l([A, B], a(l([], v(A)), l([], v(B)))),
Type =  (C->C) ;
...
Term = l([A, B], a(l([], l([], v(A))), v(B))),
Type =  (C->D->C) ;
...
```

We have noticed that both average and maximum number of colors of lambda terms grow very slowly with size. Fig. 2 compares on a log-scale the growths of simply typed closed terms and their closed affine terms subset. As for de Bruijn terms, we can define

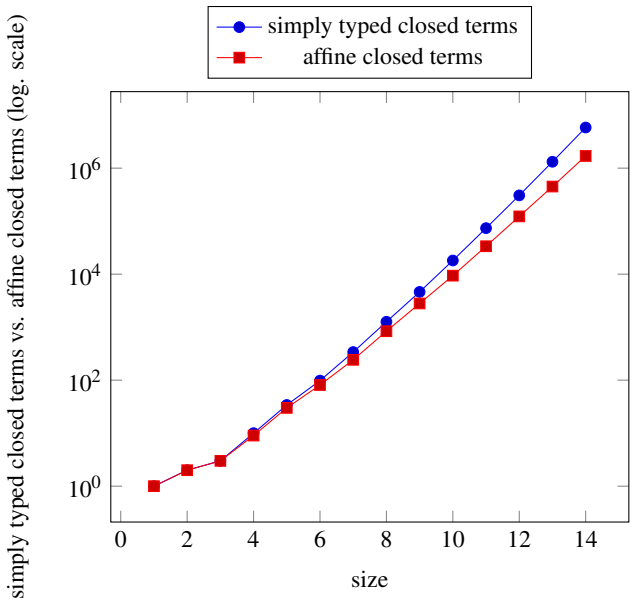


Figure 2: Counts of simply typed closed terms and affine closed terms by increasing sizes

the *Motzkin skeletons* of k-colored lambda terms by erasing the first

argument of the `l/2` and `v/1` constructors. We can also define the *k-colored Motzkin skeletons* of these terms by replacing the variable lists in argument 1 of `l/2` constructors by their length and by erasing the arguments of the `v/1` constructors.

The predicate `toSkels/3` computes the (k-colored) Motzkin skeletons by measuring the length of the list of variables for each binder.

```
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).
```

We obtain 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).
```

We can generate (but quite inefficiently) the set of typable skeletons from the multiset of skeletons by using the built-in `distinct/2` that trims duplicate solutions. We will revisit efficient generation in the next section.

```
slowTypableColSkel(N,CS):-
  distinct(CS,typableColSkels(N,CS)).

slowTypableSkel(N,S):-
  distinct(S,typableSkels(N,S)).
```

We define the *type size* of a simply typed term as the number of arrow nodes "`->`" its type contains, as computed by the predicate `tsize/2`.

```
tsize(X,S):-var(X),!,S=0.
tsize((A->B),S):-tsize(A,SA),tsize(B,SB),S is 1+SA+SB.
```

Now that we can count, for a given term size, how many k-colored terms exists, one might ask if we can say something about the sizes of their types.

Figure 3 shows the significantly slower growths of the average number of colors of colored terms vs. the average size of their types, with a possible log-scale correlation between them.

We call *a most colorful term* of a given size a term that reaches the maximum number of colors.

Fig. 4 show the relation between the number of colors of a most colorful term and a maximum size reached by the type of such a term.

Fig. 5 shows the relation between the largest type sizes the most colorful terms of a given size can attain and the maximum possible type size of those terms.

We can observe that the largest most colorful terms reach the largest possible type size for a given term size, most of the time, but as Fig. 5 shows, there are exceptions.

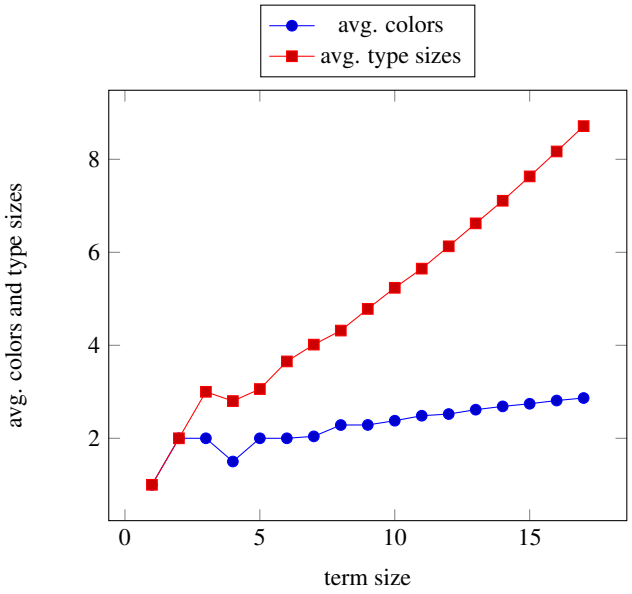


Figure 3: Growth of colors and type sizes

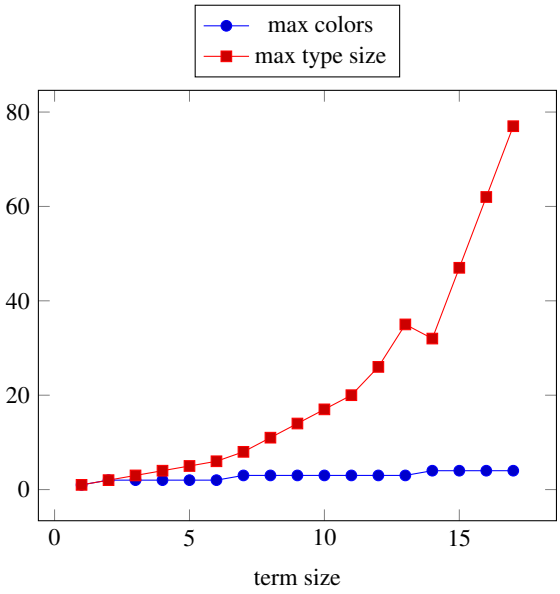


Figure 4: Colors of a most colorful term vs. its maximum type size

We leave as an open problem to prove or disprove that there's a term size such that for larger terms, the most colorful such terms reach the largest type size possible.

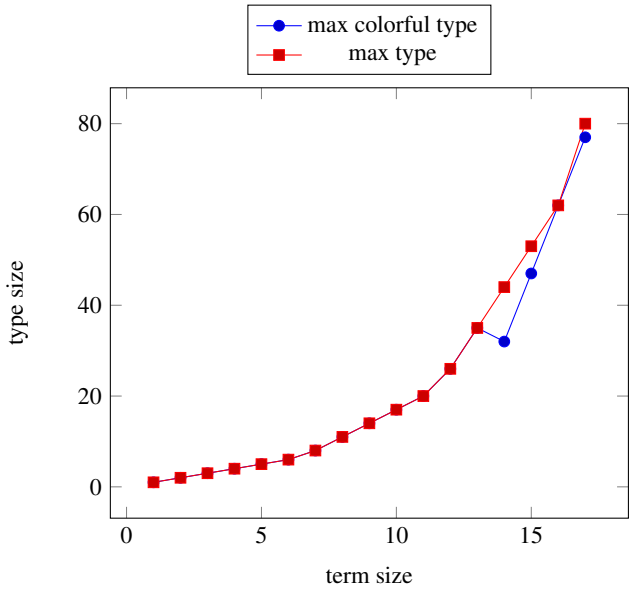


Figure 5: Largest type size of a most colorful term vs. largest type size

4 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. This leaves to the second stage to just use Prolog's metacall to activate this code.

```
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).

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).
```

The second stage code is compactly expressed as a conjunction of `el/3` predicates, each enforcing the constraint that type variable `V` corresponding to de Bruijn index `I` unifies with the type collected so far in the type variable at position `I` on the list `Vs`.

```
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).
```

One could also unfold this into a conjunction of disjunctions, to possibly rely on the services of a SAT-solver to test if the resulting CNF is satisfiable, but we leave as an "open experiment" to work out the details of such unfoldings and study their effectiveness.

For now, let us observe that this leads to a very efficient method for testing if a skeleton is typable or not, by testing if these equations have a solution.

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

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

Note the use of negation as well as the built-in `once` used to detect the existence of a solution.

An interesting problem is to find out if *there are skeletons that hold exactly one type*. In a way, that implies that the their type and set of bindings would be *pre-determined* completely by the underlying Motzkin tree.

The predicate `uniquelyTypableSkel/2` computes 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=[_].
```

Note the use of the SWI-Prolog built-in `findnsols/4` that efficiently detects the existence of exactly one solution to our unification equations.

Fig. 6 shows the existence of uniquely typable terms of for large enough sizes but also seems to indicate that upwards jumps happen for terms of an odd size. Note that one can also ask a similar

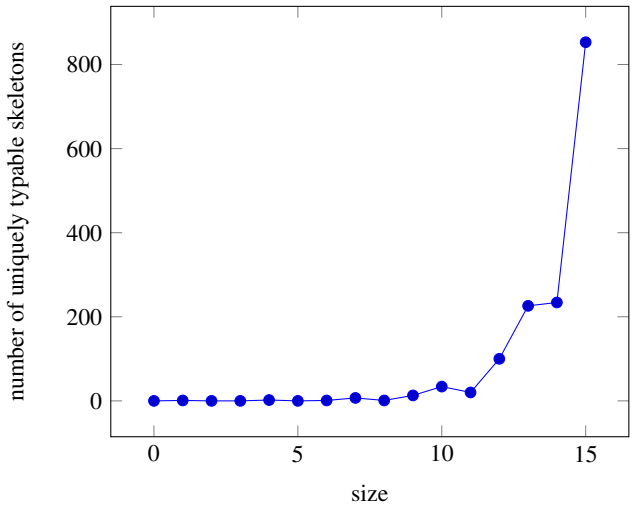


Figure 6: 3 Growth of the number of uniquely typable skeletons

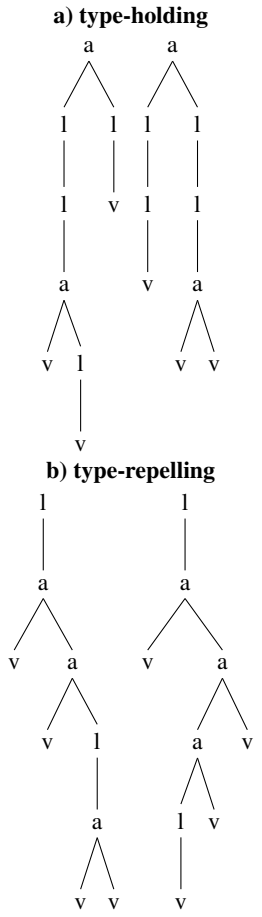


Figure 7: Type-holding, type-repelling Motzkin trees of size `8`

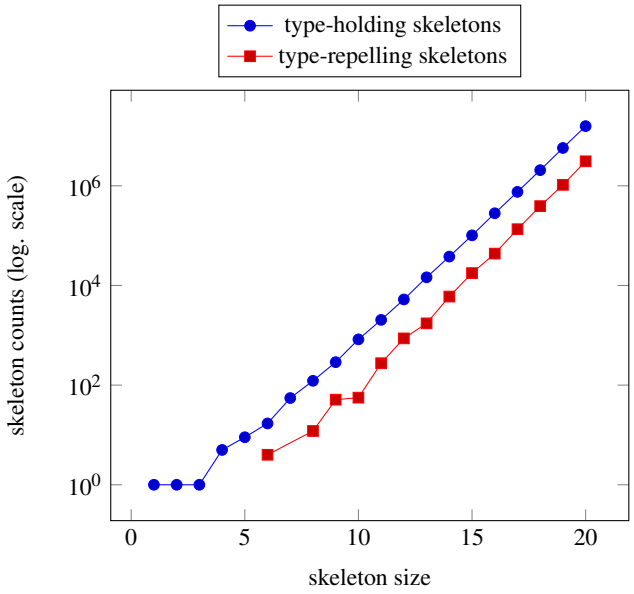


Figure 8: type-holding vs. type-repelling skeletons

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 9: Number of type-holding and type repelling skeletons

question about the class of closed lambda terms, which is likely to be tractable by analytic methods.

In Fig. 7 we show 2 type-holding and 2 type repelling Motzkin trees.

Fig. 8 and the table in Fig. 9 compare counts of type repelling and type holding Motzkin skeletons for term sizes up to `20`.

Fig. 10 shows the similar growth rates of type-holding and type-repelling skeletons.

Fig. 11 compares counts for Motzkin trees and their subset made of type-holding skeletons for increasing tree sizes.

Fig. 12 compares, on a log-scale, counts of simply typed closed terms and their skeletons.

5 AN APPLICATION TO RANDOM LAMBDA TERM GENERATORS

As a case study, putting several of our concepts together, we design a declarative implementation of Rémy's algorithm that provides random binary trees of a specified size. We then use our bijection to Motzkin trees to decorate to simply-typed closed lambda terms.

5.1 Revisiting Rémy's algorithm, declaratively

Rémy's original algorithm [18] grows binary trees by grafting new leaves with equal probability for each node in a given tree. An elegant procedural implementation is given in [14] as algorithm **R**, by using destructive assignments in an array representing the tree. While one could emulate it on top of a procedural or declarative

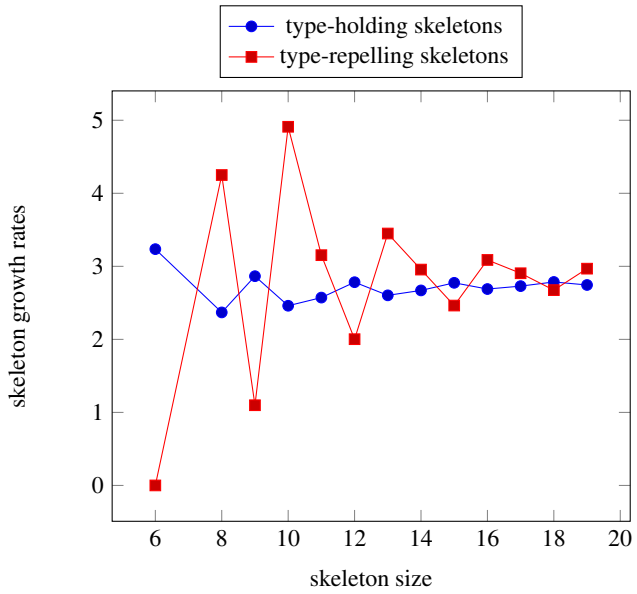


Figure 10: Growth rates of type-holding vs. type-repelling skeletons

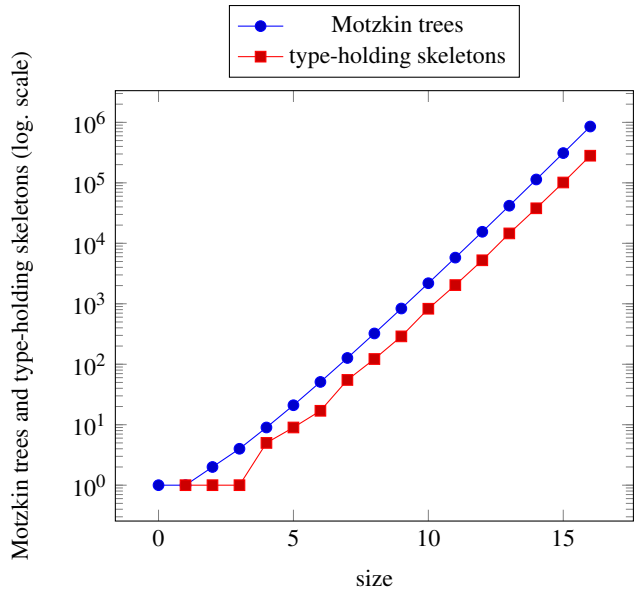


Figure 11: Counts for Motzkin trees and type-holding skeletons for increasing term sizes

emulation of updatable arrays (e.g., with `nb_setarg/3` in SWI-Prolog), we will design here a declarative implementation.

5.1.1 Trees are connected graphs: let's build them as sets of edges. First, as trees are (connected) graphs, one can represent them as sets of edges. We will use *logic variables* to label their ends representing either internal or leaf nodes. We would also label each edge as

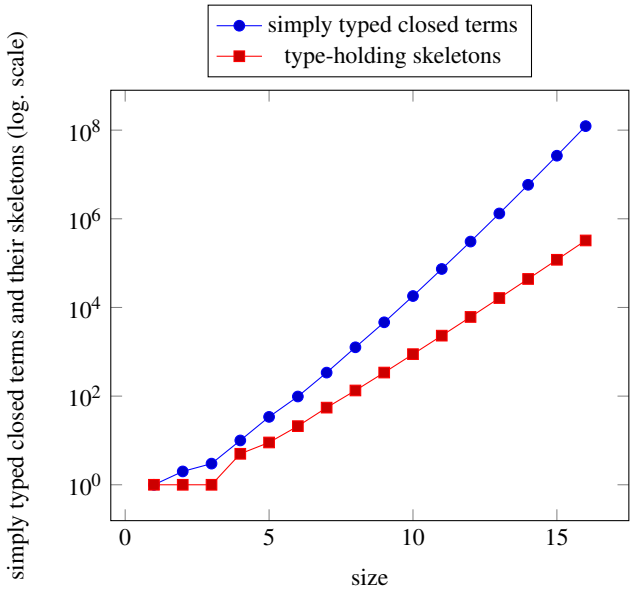


Figure 12: Counts of simply typed closed terms and their skeletons by increasing sizes

`left` or `right` to indicate their position relative to a node in the binary tree. Thus a `left` edge originating in `A` with target `B` will be represented as `e(left,A,B)`. We start with a list of two edges from root node `A` returned by the predicate `remy_init/1`.

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

The random choice of the edges (or the non-deterministic one, by replacing `choice_of/2` with its *commented out alternative*) is generated by the predicate `left_or_right/2` as:

```
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).
```

We can grow a new edge by "splitting an existing edge in two" via the predicate `grow/3`:

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

Note that a single clause defines `grow/3`, independently of the `left` or `right` denoting the relation of the edge to its source node `A`. It adds three new edges corresponding to arguments `2`, `3` and `4` and removes one, represented as its first argument. Note also, that contrary to Rémy's original algorithm, our tree grows "downward" as new edges are inserted at the target of existing ones, although this is likely to be an arbitrary choice.

The new node `C`, connected to `A` by inheriting the type `LR` of `e(LR,A,B)` will be made to point to a new leaf "`_`" via the edge

e(I,C,_) and to the tree below node B via the edge e(J,C,B). The left / right choice among I and J, is done by left_or_right(I,J).

This leads us the the basic step of the algorithm, where a set of edges Es is rewritten as a set of new edges NewEs as given by the predicate remy_step/4. To avoid computing the size L of the set Es we maintain it by adding 2=3−1 nodes, as one node is removed and 3 are added at a given step. Note that we pick an edge *randomly* among the L available by calling the choice_of/2, operation provided by remy_step1, that navigates the list to the point where the rewriting step grow/3 happens.

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

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 predicate remy_loop iterates over remy_step until the desired $2K$ size is reached, in K steps as we grow by 2 edges at each step. Note also that the initial 2 edges are added when K=1 by calling remy_init.

```
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).
```

Example 5. The generation of a random list of edges of size 4:

```
?- 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.
```

5.1.2 From sets of edges to trees as Prolog terms. The final step, "unleashing"the power of logic variables, extracts a Prolog term representing the binary tree from the list of edges labeled with unbound variables. 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. Note that, given the order-independence of the binding of logical variables, the same term is built independently of the order of the edges.

Next, the predicate bind_leaf binds the remaining unbound nodes with the constant v/0 labeling the leaf nodes. Correctness follows from the fact that a node is a leaf if and only if it remains unlabeled when the source of an edge is marked with the c/2 constructor, i.e, if it is not the source of an edge.

Note that we use maplist to iterate over lists and to apply a predicate to their corresponding elements.

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

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

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

The predicate remy_term/2 puts the two main steps together.

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

Example 6. The generation of a random term with 4 internal nodes as well timings for a larger random trees.

```
?- 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
0.078 seconds (85% CPU, 7995978 Lips)
```

While the algorithm handles fairly large terms in reasonable time, we do not claim that its average performance is linear, like in the case of Knuth's procedural implementation, given that it takes time proportional to the size of the set of edges to pick the one to be expanded. Note however, that one can improve its expected $O(N^2)$ performance with a tree representation of the set of edges to $O(Nlog(N))$ or even to amortized $O(N)$ with a dynamically growing array representation using arbitrary arity compound terms as containers.

5.2 Generating random simply typed terms

We obtain a 2-Motzkin tree generator from the binary tree generator via our bijection.

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

By starting from our random generator for Motzkin trees, or, if one prefers a uniform distribution for a given size, by using a Boltzmann sampler as the one automatically generated by [4], one can "decorate" it to lambda terms in de Bruijn notation simply by labeling its leaves with de Bruijn indices, indicating their binder as the number of l/1 constructors encountered on the path to the root of the tree.

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. As interesting size definitions depend mostly on the weight we attach to the de Bruijn indices, 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.

The predicate linChoice/4 picks an element T0 among the K elements of the list Ts, with the same probability for each.

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

The predicate expChoice does the same thing with probabilities decaying exponentially the farther the element is from the beginning of the list.

```
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).
```

Our random simply-typed closed lambda term generator proceeds by decorating a 2-colored Motzkin tree, with retries triggered by failure of being either closed or not well-typed.

The predicate `decorateTyped/3` adds de Bruijn indices to a 2-colored Motzkin tree, but quickly fails if it is not closed or not well-typed.

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

decorateTyped(v,v(I),T,K,Ts):-
  linChoice(K,Ts,I,T0), % <= any 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).
```

Retries, for generating random terms of size `N` with `MaxI` random 2-colored Motzkin trees tried, and `maxJ` decoration attempts for each, is expressed as the predicate `ranTyped/7`. On success, it returns a random term `X` of type `T` as well as the number `I` of retries for Motzkin trees and the number `J` of retries of decoration attempts.

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

Example 7. Running the random simply-typed term generator.

```
?- ranTyped(400,1000,200).
% 8.596 seconds
... large term and its type here ...
steps(791*107),natsize(1174),heapsize(400),type_size(19)
```

Note that we obtain terms of significantly larger size than with the Boltzmann sampler of [5], in exchange for giving up uniformity during the decoration step, while preserving it for the generation of 2-colored Motzkin trees.

One could also adapt this decoration algorithm to work on top of a Boltzmann sampler for Motzkin trees.

6 A PARALLEL GENERATOR FOR RANDOM SIMPLY-TYPED CLOSED LAMBDA TERMS

An important consideration when designing parallel algorithms for irregular data-types (trees in our case) is to ensure that load balancing happens efficiently and without prohibitive communication costs. However, speeding-up random generation is surprisingly easy and effective, given that one can simply start as many independent search processes as available threads.

The predicate `parTyped/7` allows passing a few parameters to fine-tune the search, `N`=heap size of the term, `MaxI, MaxI` = number of retry steps passed to `ranTyped/7`, `X, T` = term and its type, `I, J` = number of retries needed to find a solution. Otherwise, we rely on SWI-Prolog's `first_solution/3` built-in predicate, that given an answer specification, runs a list of goals, directed here via an option to continue despite of some threads terminating without finding an answer.

```
parTyped(N,MaxI,MaxJ,X,T,I,J):-
  Res=[X,T,I,J],
  prolog_flag(cpu_count,MaxThreads),
  G=ranTyped(N,MaxI,MaxJ,X,T,I,J),
  length(Goals,MaxThreads),
  maplist(=(G),Goals),
  first_solution(Res,Goals,[on_fail(continue)]).
```

Example 8. Running the parallel random simply-typed term generator on a 44 CPU / 88 hyper-thread machine.

```
?- parTyped(600,3000,600).
% 39.443 seconds
... large term and its type ...
steps(591*355),natsize(2014),heapsize(600),type_size(1244)
```

This brings us to random simply-typed terms of natural size 2000 (or heap-representation size 600), an order of magnitude above [5].

7 RELATED WORK

Several papers exist that define bijections between 2-Motzkin trees and members of the Catalan family of combinatorial (e.g., in [10]), typically via depth-first walks in trees connected to Motzkin, Dyck or Schröder paths. However, we have not found any a simple and intuitive bijection that connects components of the two families, or one that connects directly binary trees and 2-Motzkin trees, like the one shown in this paper.

The classic reference for lambda calculus is [2]. Various instances of typed lambda calculi are overviewed in [3].

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [13, 7]. Distribution and density properties of random lambda terms are described in [8].

Generation of random simply-typed lambda terms and its applications to generating functional programs from type definitions is covered in [11].

The generation and counting of affine and linear lambda terms is extensively covered in [16], with limits for counting larger than in this paper reachable using efficient recurrence formulas. Their asymptotic behavior, in relation with the BCK and BCI combinator systems, as well as bijections to combinatorial maps are studied in [6].

Asymptotic density properties of simple types (corresponding to tautologies in minimal logic) have been studied in [12] with the surprising result that "almost all" classical tautologies are also intuitionistic ones.

In [17] a "type-directed" mechanism for the generation of random terms is introduced, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some bugs in the Glasgow Haskell Compiler (GHC). A statistical exploration of the structure of the simple types of lambda terms of a given size in [22] gives indications that some types frequent in human-written programs are among the most frequently inferred ones.

Generators for closed simply-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in [13], derived from combinatorial recurrences. However,

they are significantly more complex than the ones described here in Prolog, and limited to terms up to size 10.

Rémy's algorithm [18], procedurally implemented as algorithm **R** in [14], has generated a significant number of attempts to adapt it to uniformly generate similar data types. Among them we mention [1] where it is also shown how difficult it is to ensure uniformity. For uniform generation of arbitrary data-types specified by a context-free grammar, the Boltzmann sampler generator of [4] stands out, as it actually produces efficient Haskell programs generating uniformly terms of an expected size or above.

In the unpublished draft [20] we have collected several lambda term generation algorithms written in Prolog and covering mostly de Bruijn terms and a compressed de Bruijn representation. Among them, we have covered linear, affine linear terms as well as terms of bounded unary height and in the binary lambda calculus encoding.

8 CONCLUSIONS

Contrary to closed, linear and affine lambda terms (as well as several other classes of terms subject to similar constraints) the structure of simply-typed terms has so far escaped a precise characterization. While the focus of the paper is mostly empirical, it has unwrapped some new "observables" that highlight interesting statistical properties.

In a way, the new concepts introduced involve abstraction mechanisms that "forget" properties of the difficult class of simply-typed closed lambda terms to reveal equivalence classes that are likely to be easier to grasp with analytic tools. Among them, 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" of lambda terms representing real programs.

The existence of type-repelling skeletons shows that there are Motzkin trees that cannot be skeletons of simply-typed closed terms. This suggests exploring the design of efficient algorithms built on avoiding all "small" type-repelling skeletons stored in a database.

The new, intuitive 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. In combination with our declarative implementation of Rémy's algorithm, it has also helped produce random simply-typed terms of size as large as more than **1000** nodes. Our random term generation algorithms turned out to be easy to parallelize, resulting on a machine with a large number of processor to produce simply-typed closed terms of sizes above **2000** nodes.

Last but not least, we have shown that a language as simple as side-effect-free Prolog, with limited use of impure features and meta-programming, can handle elegantly complex combinatorial generation problems, when the synergy between sound unification, backtracking and DCGs is put at work.

ACKNOWLEDGEMENT

This research has been supported by NSF grant 1423324. We thank the participants of the *CLA'2017* workshop (https://cla.tcs.uj.edu.pl/programme.html) for illuminating discussions and their comments on our presentation covering the main ideas of this paper.

REFERENCES

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

[2] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[3] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[4] M. Bendkowski. Boltzmann-brain. 2017. Software (Haskell stack module), published electronically at https://github.com/maciej-bendkowski/boltzmann-brain.

[5] M. Bendkowski, K. Grygiel, and P. Tarau. Boltzmann samplers for closed simply-typed lambda terms. In Y. Lierler and W. Taha, editors, *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*, pages 120–135. Springer, 2017.

[6] O. Bodini, D. Gardy, and A. Jacquot. Asymptotics and random sampling for BCI and BCK lambda terms. *Theoretical Computer Science*, 502:227 – 238, 2013.

[7] R. David, K. Grygiel, J. Kozik, C. Raffalli, G. Theyssier, and M. Zaionc. Asymptotically almost all λ-terms are strongly normalizing. *Preprint: arXiv: math. LO/0903.5505 v3*, 2010.

[8] R. David, C. Raffalli, G. Theyssier, K. Grygiel, J. Kozik, and M. Zaionc. Some properties of random lambda terms. *Logical Methods in Computer Science*, 9(1), 2009.

[9] N. G. de Bruijn. 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.

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

[11] B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 383–405, 2015.

[12] A. Genitrini, J. Kozik, and M. Zaionc. Intuitionistic vs. Classical Tautologies, Quantitative Comparison. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 100–109. Springer, 2007.

[13] K. Grygiel and P. Lescanne. Counting and generating lambda terms. *J. Funct. Program.*, 23(5):594–628, 2013.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional, 2006.

[15] Z. Kostrzycka and M. Zaionc. Asymptotic densities in logic and type theory. *Studia Logica*, 88(3):385–403, 2008.

[16] P. Lescanne. Quantitative aspects of linear and affine closed lambda terms. *CoRR*, abs/1702.03085, 2017.

[17] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST'11, pages 91–97, New York, NY, USA, 2011. ACM.

[18] J.-L. Rémy. 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):179–195, 1985.

[19] P. Tarau. On a Uniform Representation of Combinators, Arithmetic, Lambda Terms and Types. In E. Albert, editor, *PPDP'15: Proceedings of the 17th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 244–255, New York, NY, USA, July 2015. ACM.

[20] P. Tarau. A logic programming playground for lambda terms, combinators, types and tree-based arithmetic computations. *CoRR*, abs/1507.06944, 2015.

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

[22] P. Tarau. On Type-directed Generation of Lambda Terms. In M. De Vos, T. Eiter, Y. Lierler, and F. Toni, editors, *31st International Conference on Logic Programming (ICLP 2015), Technical Communications*, Cork, Ireland, Sept. 2015. CEUR. available online at http://ceur-ws.org/Vol-1433/.

[23] P. Tarau. 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.

[24] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12:67–96, 1 2012.

APPENDIX

Bijection between natural number and binary tree arithmetics from [19]

```
% bijection between N x N and N+
cons(I,J,C) :- I>=0,J>=0,
  D is mod(J+1,2),
  C is 2^(I+1)*(J+D)-D.

% inverse bijection between N+ and N x N
decons(K,I1,J1):-K>0,B is mod(K,2),KB is K+B,
  dyadicVal(KB,I,J),
  I1 is max(0,I-1),J1 is J-B.

% dyadic valuation of KB and residue
dyadicVal(KB,I,J):-I is lsb(KB),J is KB // (2^I).

% bijection between N and set of binary trees
n(e,0).
n(c(A,B),K):-n(A,I),n(B,J),cons(I,J,K).

% inverse bijection between the set of binary trees and N
t(0,e).
t(K,(c(A,B))):-K>0,decons(K,I,J),t(I,A),t(J,B).

% parity of the natural number associated to a tree
parity(e,0).
parity(c(_,e),1).
parity(c(_,c(X,Xs)),P1):-parity(c(X,Xs),P0),P1 is 1-P0.

% image of even in N+
even_(c(_,Xs)):-parity(Xs,1).

% image of odd in N+
odd_(c(_,Xs)):-parity(Xs,0).
```

Successor and predecessor predicates in binary tree arithmetic, from [19]

These predicates are compatible with the definitions of arithmetic operations in N, i.e., if the image of a tree is n then the image of its successor is n+1.

```
% successor
s(e,c(e,e)).
s(c(X,e),c(X,(c(e,e)))):-!.
s(c(X,Xs),Z):-parity(c(X,Xs),P),s1(P,X,Xs,Z).

s1(0,e,c(X,Xs),c(SX,Xs)):-s(X,SX).
s1(0,c(X,Xs),Xs,c(e,c(PX,Xs))):-p(c(X,Xs),PX).
s1(1,X,c(e,c(Y,Xs)),c(X,c(SY,Xs))):-s(Y,SY).
s1(1,X,c(Y,Xs),c(X,c(e,c(PY,Xs)))):-p(Y,PY).

% predecessor
p(c(e,e),e).
p(c(X,c(e,e)),c(X,e)):-!.
p(c(X,Xs),Z):-parity(c(X,Xs),P),p1(P,X,Xs,Z).

p1(0,X,c(e,c(Y,Xs)),c(X,c(SY,Xs))):-s(Y,SY).
p1(0,X,c(c(Y,Ys),Xs),c(X,c(e,c(PY,Xs)))):-p(c(Y,Ys),PY).
p1(1,e,c(X,Xs),c(SX,Xs)):-s(X,SX).
p1(1,c(X,Ys),Xs, c(e,(c(PX,Xs)))):-p(c(X,Ys),PX).
```

Shifting the binary trees to Motzkin tree bijection to include empty binary trees is achieved with the predicates `cat2mot/2` and `mot2cat/2`.

```
cat2mot(C,M):-s(C,SuccC),cat_mot(SuccC,M).
mot2cat(M,C):-cat_mot(SuccC,M),p(SuccC,C).
```

Ranking and unranking of 2-colored Motzkin trees

Ranking of 2-colored Motzkin trees via this bijection is defined as

```
rank(M,N):-mot2cat(M,C),n(C,N).
```

Unranking can then be defined as:

```
unrank(N,M):-t(N,C),cat2mot(C,M).
```

The predicates rank and unrank work as shown below:

```
?- between(0,15,N),unrank(N,M),rank(M,N1),
   N==N1, % tests assertion that it is a bijection
   writeln(N -> M),fail;nl.

0->v
1->r(v)
2->l(v)
3->a(v,v)
5->r(l(v))
6->l(r(v))
7->a(r(v),v)
8->r(a(v,v))
9->r(r(r(v)))
10->a(v,r(v))
11->a(v,l(v))
13->r(l(r(v)))
14->l(l(v))
15->a(l(v),v)

true.
```