On k-colored Lambda Terms and their Skeletons

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. The paper describes an application of logic programming to the modeling of difficult combinatorial properties of lambda terms, with focus on the class of simply typed terms.
Lambda terms in de Bruijn notation are Motzkin trees (also called binary-unary trees) with indices at their leaves counting up on the path to the root the steps to their lambda binder.
As a generalization of *affine lambda terms*, we introduce *k-colored lambda terms* obtained by labeling their lambda nodes with counts of the variables they bind. We define the *skeleton of a k-colored lambda term* as the Motzkin tree obtained by erasing the de Bruijn indices labeling its leaves. A new bijection between 2-colored skeletons and binary trees reveals their connection to the Catalan family of combinatorial objects.
After a statistical study of properties of *k-colored lambda terms* and their skeletons, we focus on the case of *simply-typed closed k-colored* lambda terms for which a new combinatorial generation algorithm is given and some interesting relations between maximal coloring, size of type expressions and typability are explored.
The paper is structured as a literate Prolog program to facilitate an easily replicable, concise and declarative expression of our concepts and algorithms.

Keywords: declarative modeling of combinatorial classes, families of lambda terms, simply-typed closed lambda terms, Motzkin trees, bijections between data types.

1 Introduction

Lambda terms, in de Bruijn notation [1], 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. We call the *skeleton of a lambda term* the Motzkin tree obtained by erasing its de Bruijn indices.

A useful distinction can be made between lambda constructors that bind variables and those that do not. Among other benefits, distinguishing them makes the analysis of linear and affine terms simpler and puts their skeletons, the 2-colored Motzkin trees, in bijection with the well-known Catalan family of combinatorial objects.

More generally, can we classify lambda nodes by inverting the function from indices at the leaves to their binders?

This leads to the concept of *k-colored lambda terms* where colors classify binders depending on the number of variables they bind. It also brings us to the main focus of

this paper, the interaction of k-coloring, skeletons and the class of simply-typed terms, starting with the easy case of always typable affine and linear terms and then empirically approaching some interesting observables for the notoriously difficult general case.

Despite the asymptotically vanishing density of simply-typed lambda terms [2], 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 [3, 4]. 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 [4], by mimicking their exhaustive generation with a recursive structure that, while omitting the actual lambda terms, keeps the type-inference mechanism intact.

These difficulties are the main motivation of this paper, which suggests a fresh look at the structure of simply typed terms and their type expressions, via their relations to their k-colored skeletons, revealing insights on the structure of simply-typed closed lambda terms.

The paper is organized as follows. Section 2 describes a new bijection between binary trees and 2-colored Motzkin trees. Section 3 discusses the case of closed, linear and affine lambda terms. Section 4 focuses on the case of k-colored simply typed closed lambda terms and their statistical properties. Section 5 overviews related work and section 6 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/klambs.pro`, tested with SWI-Prolog [5] version `7.4.2`.

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 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. We can think about these as *2-colored lambda terms*. 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 define the *2-colored Motzkin skeleton of a lambda term* (shortly *skeleton*) as the 2-Motzkin 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 [6]. 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. As an application, this allows one to use a uniform random binary tree generation algorithm like [7] to generate random 2-Motzkin trees.

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.

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 2-colored 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).
```

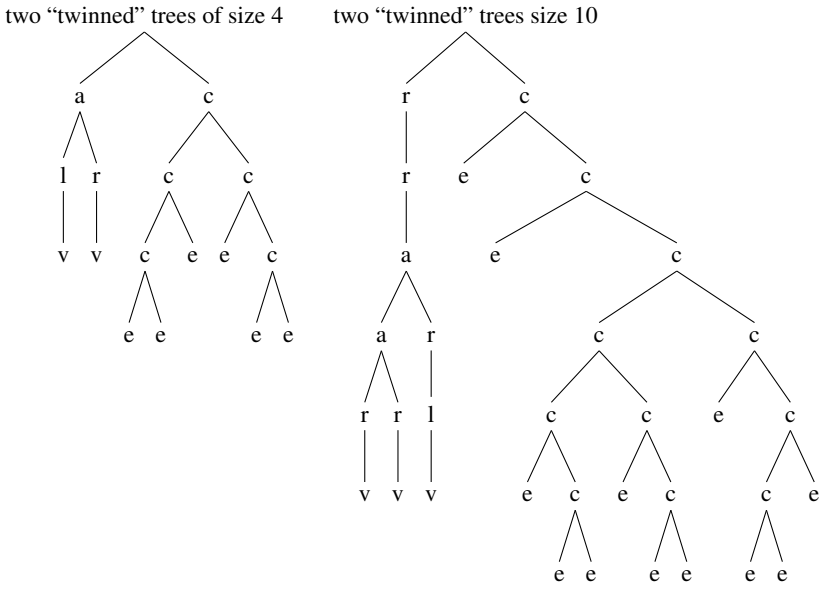


Fig. 1. The 2-colored Motzkin trees to non-empty binary trees bijection

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

The **Appendix** shows ranking and unranking functions (bijections to/from the set of natural numbers) based on the arithmetization of binary trees described in [8].

3 Closed, affine and linear terms

We can see 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.

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).

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 [9], 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 skeletons 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 skeletons of sizes 0,1,2,... :
1,0,1,2,4,10,25,62,160,418,1102,2940,7912,21444,58507,160544,442748, ...

We refer to [10] for an analytic solution proving that asymptotically $\frac{1}{\sqrt{5}}$ of the skeletons are closable.

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 [11], 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.

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

```
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 general case of k-colored terms.

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.

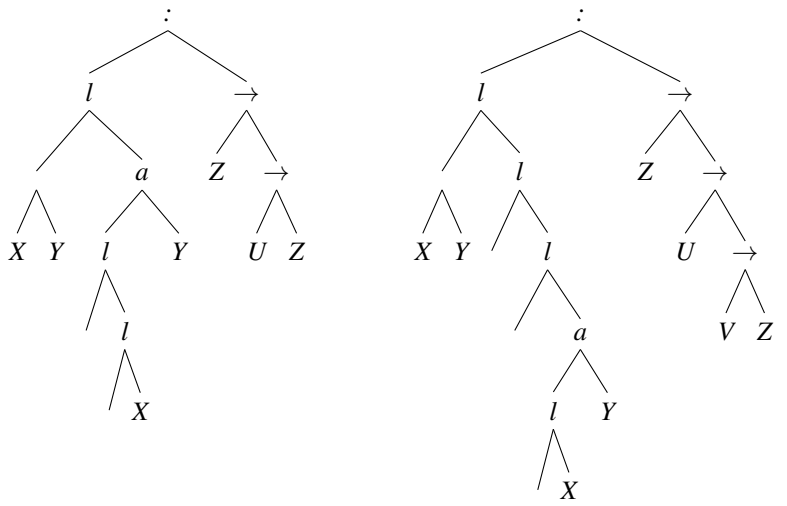
4.1 Type inference for k-colored terms

The study of the combinatorial properties of simply-typed lambda terms is notoriously hard. 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 crossing an application node trims it down
- *agreement via unification (with occurs check)* between the types of each variable under a lambda.

Interestingly, at our best knowledge, no SAT or ASP algorithms exist in the literature that attack the combined type inference and combinatorial generation problem for lambda terms, most likely because of the complexity of emulating unification-with-occurs-check steps in propositional logic. Thus we will follow the interleaving of term generation, checking for closedness and type inference steps shown in [8], 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 [8], 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 are now ready to make some empirical observations on terms, colors and type sizes. 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

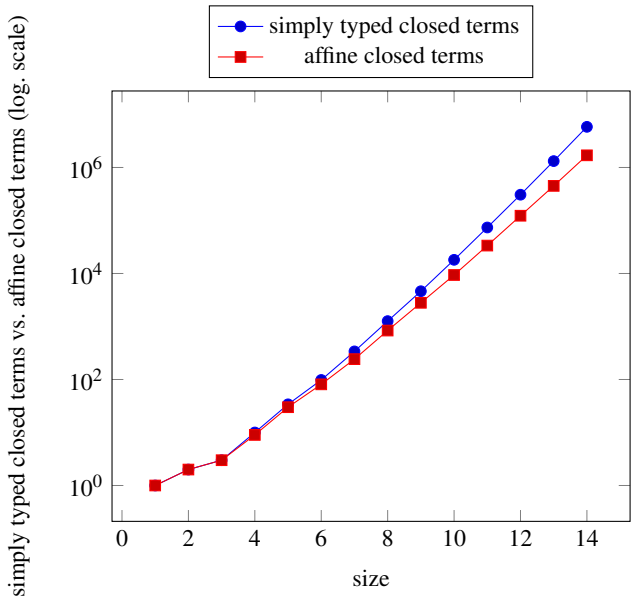


Fig. 2. Counts of simply typed closed terms and affine closed terms by increasing sizes

define 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 the set of typable skeletons from the multiset of skeletons by using the built-in `distinct/2` that trims duplicate solutions.

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

simpleTypableColSkel(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. This suggest an investigation of the relations between the complexity of type expressions and coloring.

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.

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.

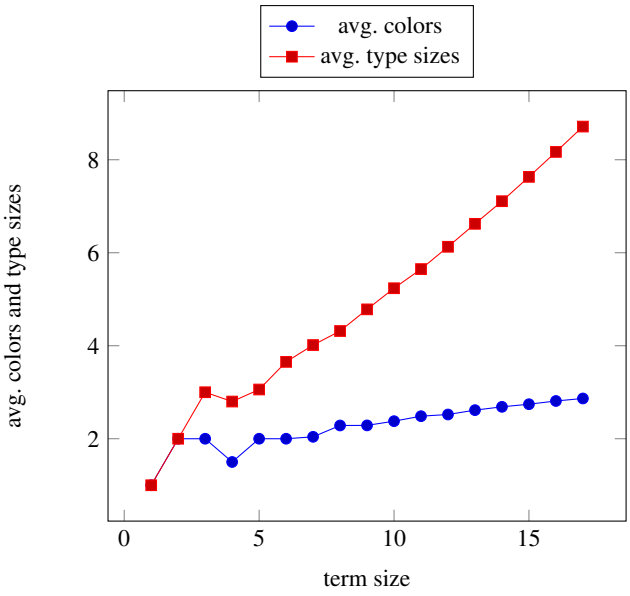


Fig. 3. Growth of colors and type sizes

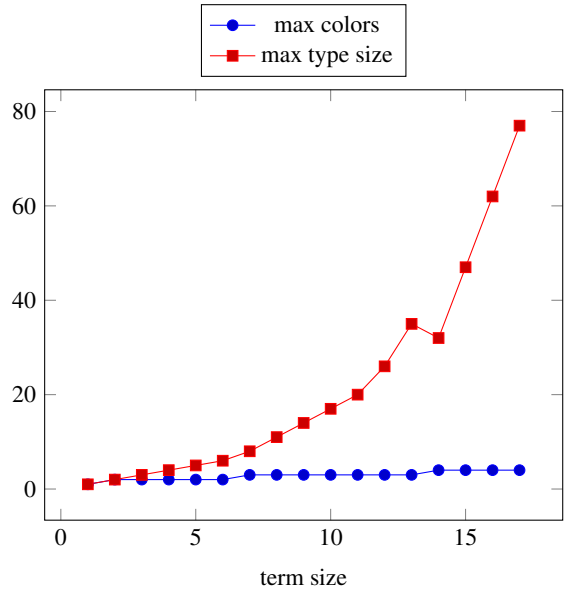


Fig. 4. Colors of a most colorful term vs. its maximum type size

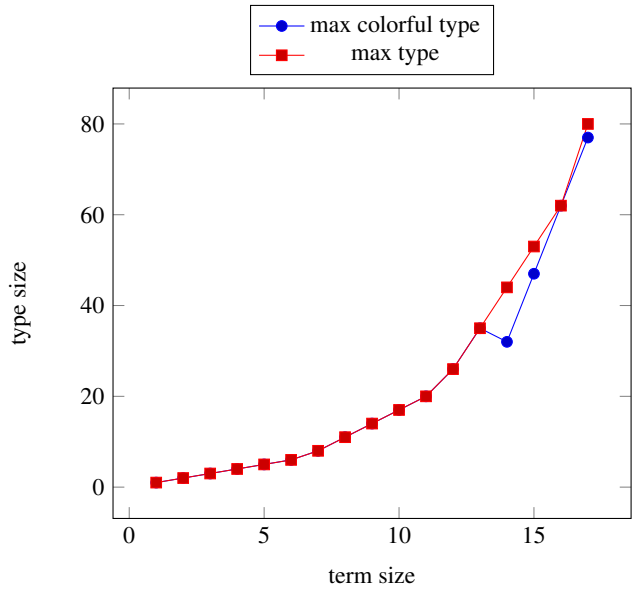


Fig. 5. Largest type size of a most colorful term vs. largest type size

5 Related work

Several papers exist that define bijections between 2-Motzkin trees and members of the Catalan family of combinatorial (e.g., in [6]), typically via depth-first walks in trees connected to Motzkin, Dyck or Schröder paths. However, we have not found any 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 [12]. Various instances of typed lambda calculi are overviewed in [13]. The use of de Bruijn indices for the study of combinatorial properties of lambda terms is introduced in [14].

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [15, 16]. Distribution and density properties of random lambda terms are described in [17].

The generation and counting of affine and linear lambda terms is extensively covered in [9], 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 [18]. Among them, we have covered linear, affine linear terms as well as terms of bounded unary height and in the binary lambda calculus encoding. In [10] analytic models are used to solve the problem of the asymptotic density of closable skeletons and their subclass of uniquely closable skeletons.

Asymptotic density properties of simple types (corresponding to tautologies in minimal logic) have been studied in [19] with the surprising result that "almost all" classical tautologies are also intuitionistic ones.

6 Conclusions

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 Rémy's algorithm [7], for the generation of random binary trees, it can also be used to produce large random simply-typed terms.

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. The relations identified between colors and type sizes of lambda terms have led to some interesting (but possibly very hard) open problems.

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.

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. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae **34** (1972) 381–392
2. Kostrzycka, Z., Zaionc, M.: Asymptotic densities in logic and type theory. Studia Logica **88**(3) (2008) 385–403
3. Tarau, P. In: A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms. Springer LNCS, volume 10184 (2017) 240–255 Best paper award.

4. 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 Best student paper award.
5. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12** (1 2012) 67–96
6. 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
7. 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
8. Tarau, P.: On a Uniform Representation of Combinators, Arithmetic, Lambda Terms and Types. In Albert, E., ed.: PPDP'15: Proceedings of the 17th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, New York, NY, USA, ACM (July 2015) 244–255
9. Lescanne, P.: Quantitative aspects of linear and affine closed lambda terms. CoRR **abs/1702.03085** (2017)
10. Bodini, O., Tarau, P.: On Uniquely Closable and Uniquely Typable Skeletons of Lambda Terms. CoRR **abs/1709.04302** (September 2017)
11. 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
12. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. Revised edn. Volume 103. North Holland (1984)
13. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1991)
14. Lescanne, P.: On counting untyped lambda terms. Theoretical Computer Science **474** (2013) 80 – 97
15. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5) (2013) 594–628
16. David, R., Grygiel, K., Kozik, J., Raffalli, C., Theyssier, G., Zaionc, M.: Asymptotically almost all λ-terms are strongly normalizing. Preprint: arXiv: math. LO/0903.5505 v3 (2010)
17. David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., Zaionc, M.: Some properties of random lambda terms. Logical Methods in Computer Science **9**(1) (2009)
18. Bodini, O., Gardy, D., Jacquot, A.: Asymptotics and random sampling for BCI and BCK lambda terms. Theoretical Computer Science **502** (2013) 227 – 238
19. Genitrini, A., Kozik, J., Zaionc, M.: Intuitionistic vs. Classical Tautologies, Quantitative Comparison. In Miculan, M., Scagnetto, I., Honsell, F., eds.: 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., Springer (2007) 100–109

Appendix

Bijection between natural number and binary tree arithmetics from [8]

```
% 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 [8]

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.
```