On Uniquely Closable and Uniquely Typable Skeletons of Lambda Terms

Olivier Bodini[1] and Paul Tarau[2]

[1] Laboratoire d'Informatique de Paris-Nord
UMR CNRS 7030
olivier.bodini@lipn.univ-paris13.fr
[2] Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. Uniquely closable skeletons of lambda terms are Motzkin-trees that predetermine the unique closed lambda term that can be obtained by labeling their leaves with de Bruijn indices. Likewise, uniquely typable skeletons of closed lambda terms predetermine the unique simply-typed lambda term that can be obtained by labeling their leaves with de Bruijn indices.
We derive, through a sequence of logic program transformations, efficient code for their combinatorial generation and study their statistical properties.
As a result, we obtain context-free grammars describing closable and uniquely closable skeletons of lambda terms, opening the door for their in-depth study with tools from analytic combinatorics.
Our empirical study of the more difficult case of (uniquely) typable terms reveals some interesting open problems about their density and asymptotic behavior.
As a connection between the two classes of terms, we also show that uniquely typable closed lambda term skeletons of size $3n+1$ are in a bijection with binary trees of size n.
Keywords: *deriving efficient logic programs, logic programming and computational mathematics, combinatorics of lambda terms, inferring simple types, uniquely closable lambda term skeletons, uniquely typable lambda term skeletons.*

1 Introduction

Lambda calculus [1] has been, together with Turing machines and combinators a key foundational framework describing the essence of universal computations on which computers and their smaller siblings help running our everyday digital lives.

Lambda calculus has started being used as an actual programming language construct in early functional languages like LISP and is prevalent in this role in modern functional languages like Haskell, ML and F#. In the last few years it has also made it as an actual language construct in virtually all widely used programming languages ranging from C++, C# and Java to Python, Javascript, Ruby and Scala.

Computation in the lambda calculus operate on *lambda terms*, an amazingly simple data structure, conveniently seen, in *de Bruijn notation* [2], as trees made of unary and binary nodes with leaves labeled with integers indicating their way up to their lambda binders, as described by the following Haskell data type declaration:

```
data DeBruijnTerm =
  DeBruijnIndex Integer |
  Lambda DeBruijnTerm |
  Application DeBruijnTerm DeBruijnTerm
```

When computations are triggered via binary application nodes, lambda binders on their left side direct substitutions of terms on their right side to the leaf nodes the binders cover.

When every de Bruijn index has a lambda binder a term is called *closed*. Among closed lambda terms, *simply typed lambda terms* stand out as a model of well-behaved computations that mimic the semantics of mathematical functions operating on sets.

The study of combinatorial properties of lambda terms has theoretical ramifications ranging from their connection to proofs in intuitionistic logic via the Curry-Howard correspondence [3] and their role as a foundation of Turing-complete as well as expressive but terminating computations in the case of simply typed lambda terms [4]. At the same time, lambda terms are used in the internal representations of compilers for functional programming languages and proof assistants, for which the generation of large random lambda terms helps with automated testing [5].

This paper focuses on binary-unary trees that are obtained from lambda terms in de Bruijn form, represented as trees, by erasing the de Bruijn indices labeling variables at their leaves. Such "skeletons" of the lambda terms turn out to predetermine some non-trivial properties the lambda terms they host, e.g., if such terms are closed or simply-typed. Of particular interest are the cases when unique such terms exist.

Our declarative meta-language is Prolog, which turns out to provide everything we need: easy combinatorial generation via backtracking over the set of all answers, specified as a Definite Clause Grammar (DCG) that enforces size constraints and allows placing more complex constraints at points in the code where they ensure the earliest possible pruning of the search space.

Our meta-language also facilitates program transformations that allow us to derive step-by-step faster programs as well as simpler expressions of the underlying combinatorial mechanisms, e.g., a context-free grammar in the case of uniquely closable skeletons, that in turn makes them amenable to study with powerful techniques from analytical combinatorics.

The paper is organized as follows. Section 2 describes generators for closed lambda terms and their Motzkin-tree skeletons. Section 3 introduces closable skeletons and studies their statistical properties. Section 4 derives algorithms (including a CF-grammar) for efficient generation of uniquely closable skeletons. Section 5 discusses typable and untypable closed skeletons. Section 6 introduces uniquely typable closed skeletons, studies the special case of uniquely closable and uniquely typable skeletons and establishes their connection to members of the Catalan family of combinatorial objects. 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, tested with SWI-Prolog [6] version `7.5.3`, is available at: `http://www.cse.unt.edu/~tarau/research/2017/uct.pro` .

2 Closed Lambda Terms and their Motzkin-tree Skeletons

A *Motzkin tree* (also called binary-unary tree) is a rooted ordered tree built from binary nodes, unary nodes and leaf nodes. Thus the set of Motzkin trees can be seen as the free algebra generated by the constructors `v/0`, `l/1` and `a/2`.

We define lambda terms in de Bruijn form [2] as the free algebra generated by the constructors `l/1`, and `a/2` and leaves labeled with natural numbers wrapped with the constructor `v/1`.

A lambda term in de Bruijn form is *closed* if for each of its de Bruijn indices it exists a lambda binder to which it points, on the path to the root of the tree representing the term. They are counted by sequence **A135501** in [7].

The predicate `closedTerm/2` specifies an all-terms generator, which, given a natural number `N` backtracks one member `X` at a time, over the set of terms of size `N`.

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

closedTerm(v(I),V)-->{V>0,V1 is V-1,between(0,V1,I)}.
closedTerm(l(A),V)-->l,{succ(V,NewV)},closedTerm(A,NewV).
closedTerm(a(A,B),V)-->a,closedTerm(A,V),closedTerm(B,V).
```

The *size definition* is expressed by the work of the predicates `l/1`, consuming *1* size unit for each *lambda binder* and `a/2` consuming *2* size units for each `a/2` *application* constructor and *0* units for variables `v/1`. The initial term which is just a unique variable has size *0*.

Given that the number of leaves in a Motzkin tree is the number of binary nodes + *1*, it follows that:

Proposition 1 *The set of terms of size n for the size definition* {application=2, lambda=1, variable=0} *is equal to the set of terms of size* $n+1$ *for the size definition* {application=1, lambda=1, variable=1}.

Thus our size definition gives the sequence **A135501** of counts, first introduced in [8], shifted by one. For instance, the term $l(a(v(0),v(0)))$ will have size $3 = 1+2$ with our definition, which corresponds to size $4 = 1+1+1+1$ using the size definition of [8].

Our size definition is implemented as

```
l(SX,X):-succ(X,SX). % true if SX>0 and X is SX-1
a-->l,l.
```

with Prolog's DCG notation controlling the consumption of size units from `N` to `0`.

The predicate `toMotSkel/2` computes the Motzkin skeleton of a term.

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

The predicate `motSkel/2` generates Motzkin trees `X` of size `N`, using the same size definition as the lambda terms for which they serve as skeletons.

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

```
motSkel(v)-->[].
motSkel(l(X))-->l,motSkel(X).
motSkel(a(X,Y))-->a,motSkel(X),motSkel(Y).
```

3 Closable and Unclosable Skeletons

We call a Motzkin tree *closable* if it is the skeleton of at least one closed lambda term.

The predicate `isClosable/1` tests if it exists a closed lambda term having `X` as its skeleton. For each lambda binder it increments a count `V` (starting at `0`), and ensures that it is strictly positive for all leaf nodes.

```
isClosable(X):-isClosable(X,0).

isClosable(v,V):-V>0.
isClosable(l(A),V):-succ(V,NewV),isClosable(A,NewV).
isClosable(a(A,B),V):-isClosable(A,V),isClosable(B,V).
```

We define generators for closable and unclosable skeletons by filtering the stream of answers of the Motzkin tree generator `motSkel/2` with the predicate `isClosable/1` and its negation.

```
closableSkel(N,X):-motSkel(N,X),isClosable(X).

unClosableSkel(N,X):-motSkel(N,X),not(isClosable(X)).
```

In Fig. 1 we show 3 closable and 3 unclosable Motzkin skeletons.

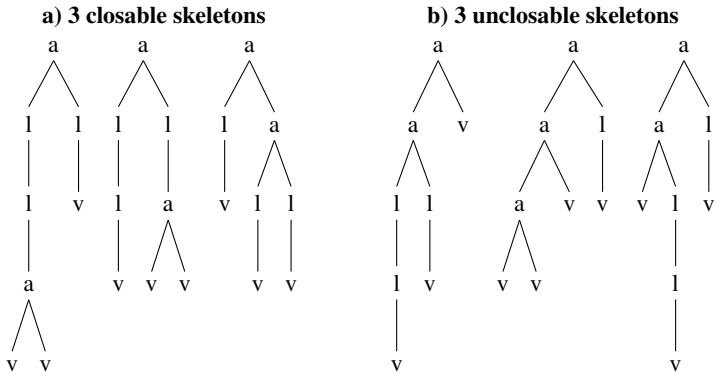


Fig. 1. Closable vs. unclosable skeletons of size 7

Next, we derive the predicate `quickClosableSkel/2` that generates closable skeletons about 3 times faster by testing directly that lambda binders are available at each leaf node, resulting in earlier pruning of those that do not satisfy this constraint.

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

quickClosableSkel(v,V)-->{V>0}.
quickClosableSkel(l(A),V)-->l,{succ(V,NewV)},quickClosableSkel(A,NewV).
quickClosableSkel(a(A,B),V)-->a,
  quickClosableSkel(A,V),
  quickClosableSkel(B,V).
```

One step further, we can derive a grammar generating closable skeletons, by observing that they require at least one lambda (`l/1` constructor) on each path, with Motzkin trees below the `l/1` constructor, as generated by the predicate `motSkel/3` introduced in section 2. Thus, the following holds:

Proposition 2 *A Motzkin tree is a skeleton of a closed lambda term if and only if it exists at least one lambda binder on each path from the leaf to the root.*

The predicate `closable/2`, implementing the corresponding CF-grammar as a generator, runs about 3 times as fast as `closableSkel/2`.

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

closable(l(Z))-->l,motSkel(Z).
closable(a(X,Y))-->a,closable(X),closable(Y).
```

By entering this grammar as input to Maciej Bendkowski's Boltzmann sampler generator [9] we have obtained a Haskell program generating uniformly random closable skeletons of one hundred thousand nodes in a few seconds. The probability to pick `l/1` and enter a Motzkin subtree instead of an `a/2` constructor was `0.8730398709632761`. The threshold within the Motzkin subtree to pick a leaf was `0.3341408333975344`, then `0.667473848839429` for a unary constructor, over which a binary constructor was picked.

We observe that 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, ...

Let us denote by $M(z) = \sum m_n z^n$ the ordinary generating function for Motzkin trees (m_n is the number of Motzkin trees of size n). It is well known [10] that $M(z)$ follows the algebraic functional equation $M = z + zM + zM^2$ which can be obtained directly from the symbolic method and we get $M(z) = \frac{1-z-\sqrt{-3z^2-2z+1}}{2z}$. From this, we obtain the classical result that asymptotically the number m_n of Motzkin trees of size n is equivalent to $\frac{\sqrt{3}}{2\sqrt{\pi}} 3^n n^{-3/2}$.

Now, following the proposition 1 (and the predicate `closable/2` providing the corresponding grammar definition), we can deduce that the ordinary generating function $C(z)$ for closable lambda terms satisfies $C(z) = zC(z)^2 + zM(z)$. Indeed, a closable term has either an application at the root followed by two sub-closable terms (which gives

rise to $zC(z)^2$), either an abstraction at the root followed by a term (which gives rise to $zM(z)$). Consequently, $C(z) = \frac{1-\sqrt{2z\sqrt{-3z^2-2z+1}+2z^2-2z+1}}{2z}$. Now, we are in the framework of the Flajolet-Odlysko transfer theorems [11] which gives directly the asymptotics of the number c_n of closable skeletons: $c_n \sim \frac{\sqrt{15}}{10\sqrt{\pi}} 3^n n^{-3/2}$. By dividing c_n with m_n we obtain:

Proposition 3 *When n tends to the infinity, the proportion of closable lambda term skeletons tends to* $\frac{1}{\sqrt{5}} \doteq 44.7\%$.

It is possible to calculate very efficiently the coefficients c_n. For that purpose, from the equation $C(z) = zC(z)^2 + zM(z)$, an easy calculation gives that $C(z)$ satisfies the algebraic equation $z^2C(z)^4 - 2zC(z)^3 + (-z^2+z+1)C(z)^2 + (z-1)C(z) + z^2$. Thus, dealing with classical tools (in order to pass from an algebraic equation into a holonomic one), we can deduce a linear differential equation from it:

$$\begin{gathered}
0 = -208z^6 - 168z^5 + 12z^4 + 94z^3 - 42z^2 + 6z + \\
\left(-16z^6 + 24z^5 + 36z^4 - 92z^3 + 60z^2 - 12z\right) C(z) + \\
\left(768z^9 - 480z^8 - 1088z^7 - 64z^6 + 216z^5 + 44z^4 + 30z^3 - 54z^2 + 18z - 2\right) \tfrac{\mathrm{d}}{\mathrm{d}z} C(z) + \\
\left(384z^{10} - 32z^9 - 368z^8 - 56z^7 - 4z^6 + 110z^5 - 21z^4 - 21z^3 + 9z^2 - z\right) \tfrac{\mathrm{d}^2}{\mathrm{d}z^2} C(z)
\end{gathered}$$

with the initial condition $C(0) = 0$. Now, extracting a relation on the coefficients from this holonomic equation, we obtain the following P-recurrence for the coefficient c_n:

$$\begin{gathered}
\left(384n^2 + 384n\right) c_n + \\
\left(-32n^2 - 512n - 480\right) c_{n+1} + \\
\left(-368n^2 - 2192n - 2928\right) c_{n+2} + \\
\left(-56n^2 - 344n - 504\right) c_{n+3} + \\
\left(-4n^2 + 188n + 852\right) c_{n+4} + \\
\left(110n^2 + 1034n + 2328\right) c_{n+5} + \\
\left(-21n^2 - 201n - 390\right) c_{n+6} + \\
\left(-21n^2 - 327n - 1272\right) c_{n+7} + \\
\left(9n^2 + 153n + 648\right) c_{n+8} + \\
\left(-n^2 - 19n - 90\right) c_{n+9} = 0
\end{gathered}$$

with the initial conditions $c_0 = 0, c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 2, c_5 = 5, c_6 = 11, c_7 = 26, c_8 = 65$

Note that by a guess-and-prove approach, we can a little simplify the recurrence into:

$$\begin{gathered}
\left(1200n^5+18480n^4+90816n^3+161088n^2+87552n\right)c_n+\\
\left(800n^5+13520n^4+79024n^3+202312n^2+231768n+95760\right)c_{n+1}+\\
\left(-100n^5-1840n^4-12848n^3-38792n^2-44100n-9576\right)c_{n+2}+\\
\left(-100n^5-1990n^4-14648n^3-48254n^2-66276n-23940\right)c_{n+3}+\\
\left(-225n^5-4815n^4-38883n^3-147519n^2-260286n-167580\right)c_{n+4}+\\
\left(150n^5+3435n^4+29817n^3+120441n^2+218739n+131670\right)c_{n+5}+\\
\left(-25n^5-610n^4-5642n^3-24128n^2-45405n-26334\right)c_{n+6}=0
\end{gathered}$$

with the initial conditions $c_0=0,c_1=0,c_2=1,c_3=1,c_4=2,c_5=5$

This recurrence is extremely efficient in order to calculate the coefficient c_n.

Alternatively, the expansion into Taylor series of $C(z)$ gives $z^2+z^3+2z^4+5z^5+11z^6+26z^7+65z^8+163z^9+417z^{10}+1086z^{11}+2858z^{12}+7599z^{13}+20391z^{14}+55127z^{15}...$ with its coefficients matching the number of terms of sizes given by the exponents, corresponding to the number of solutions of the predicate `closableSkel/2`.

4 Uniquely Closable Skeletons

We call a skeleton *uniquely closable* if it exists exactly one closed lambda term having it as its skeleton.

Proposition 4 *A skeleton is uniquely closable if and only if exactly one lambda binder is available above each of its leaf nodes.*

Proof. Note that if more than one were available for any leaf `v`, one could choose more then one de Bruijn index at the corresponding leaf `v/1` of a lambda term, resulting in more than one possible lambda terms having the given skeleton.

The predicate `uniquelyClosable1/2` derived from `quickClosableSkel1/2` ensures that for each leaf `v/0` exactly one lambda binder is available.

```
uniquelyClosable1(N,X):-uniquelyClosable1(X,0,N,0).

uniquelyClosable1(v,1)-->[].
uniquelyClosable1(l(A),V)-->l,{succ(V,NewV)},uniquelyClosable1(A,NewV).
uniquelyClosable1(a(A,B),V)-->a,uniquelyClosable1(A,V),
  uniquelyClosable1(B,V).
```

As a skeleton is uniquely closable if on any path from a leaf to the root there's exactly one `l/1` constructor, we derive the predicate `uniquelyClosable2/2` that marks subtrees below a lambda `l1/1` constructor to ensure no other l/1 constructor is used in them.

```
uniquelyClosable2(N,X):-uniquelyClosable2(X,hasNoLambda,N,0).

uniquelyClosable2(v,hasOneLambda)-->[].
uniquelyClosable2(l(A),hasNoLambda)-->l,
  uniquelyClosable2(A,hasOneLambda).
uniquelyClosable2(a(A,B),Has)-->a,uniquelyClosable2(A,Has),
  uniquelyClosable2(B,Has).
```

By specializing with respect to having or not having a lambda binder above, we obtain `uniquelyClosable/2` which mimics a context-free grammar generating all uniquely closable skeletons of a given size.

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

uniquelyClosable(l(A))-->l,closedAbove(A).
uniquelyClosable(a(A,B))-->a,uniquelyClosable(A),uniquelyClosable(B).

closedAbove(v)-->[].
closedAbove(a(A,B))-->a,closedAbove(A),closedAbove(B).
```

In fact, if one wants to only count the number of solutions, the actual term (argument 1) can be omitted, resulting in the even faster predicate `uniquelyClosableCount/1`.

```
uniquelyClosableCount(N):-uniquelyClosableCount(N,0).

uniquelyClosableCount-->l,closedAboveCount.
uniquelyClosableCount-->a,uniquelyClosableCount,uniquelyClosableCount.

closedAboveCount-->[].
closedAboveCount-->a,closedAboveCount,closedAboveCount.
```

This sequence of program transformations results in code running an order of magnitude faster, with all counts up to size 30, shown in Fig. 2, obtained in less than a minute. Fig. 2 shows the growths of the set of uniquely closable skeletons.

If expressed as a Haskell data type, the grammar describing the set of closable skeletons becomes:

```
data UniquelyClosable =  L ClosedAbove
  | A UniquelyClosable UniquelyClosable deriving(Eq,Show,Read)

data ClosedAbove = V | B ClosedAbove ClosedAbove deriving(Eq,Show,Read)
```

With this notation, a skeleton, with the constructor `B` used for binary trees not containing an `L` constructor, is `A (A (L V) (L V)) (L (B (B V V) V))`.

One can transliterate the Prolog DCG grammar into Haskell by using list comprehensions to mimic backtracking as follows.

```
genA 0 = []
genA n | n>0 =
   [L x | x <- genB (n-1)] ++
   [A x y | k <- [0..n-2], x <- genA k,  y <- genA (n-2-k)]
```

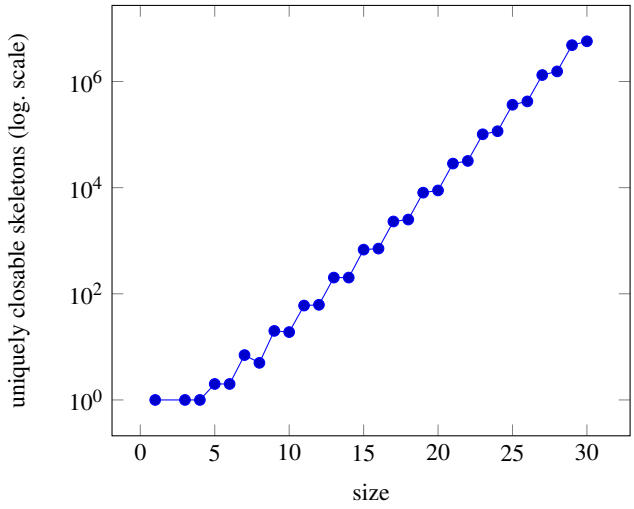


Fig. 2. Uniquely closable skeletons by increasing sizes

```
genB 0 = [V]
genB n | n>0 = [B a b | k <- [0..n-2], a <- genB k, b <- genB (n-2-k)]
```

By entering the equivalent of this data type definition as input to Maciej Bendkowski's Boltzmann sampler generator [9] we have obtained a Haskell program generating uniformly random terms of one hundred thousand nodes in a few seconds. The probability threshold for a unary constructor was below `0.5001253328728457` and then, once having entered a closed above subtree, it was `0.5001253328728457` to stop at a leaf rather than continuing with a binary node.

Let us denote by $B(z)$ the ordinary generating function for binary trees. The series $B(z)$ follows the algebraic functional equation $B = z + zB^2$ and consequently $B(z) = \frac{1-\sqrt{-4z^2+1}}{2z}$.

The ordinary generating function $U(z)$ for uniquely closable lambda terms satisfies $U(z) = zU(z)^2 + zB(z)$. Indeed, a uniquely closable term has either an application at the root followed by two sub uniquely closable terms (which gives rise to $zC(z)^2$), either an abstraction at the root followed by a term with no abstraction (which gives rise to $zB(z)$). Consequently, $U(z) = \frac{1-\sqrt{2z\sqrt{-4z^2+1}-2z+1}}{2z}$. We are again in the framework of the Flajolet-Odlysko transfer theorems [11] which gives directly the asymptotics of the number u_n of uniquely closable terms: $u_n \sim \frac{2^{1/4+n}}{4\Gamma(3/4)n^{5/4}}$.

We can follow the same approach that for $C(z)$ to calculate quickly the coefficients u_n. In particular, $U(z)$ satisfies the algebraic equation $z^2U(z)^4 - 2zU(z)^3 + (z+$

$1)U(z)^2 - U(z) + z^2 = 0$. From which we deduce a linear differential equation:

$$0 = -128z^5 - 40z^4 + 52z^3 + 18z^2 - 6z + (16z^5 + 56z^4 - 20z^3 - 20z^2 + 8z - 2)U(z) + (512z^8 - 512z^7 - 320z^6 + 96z^5 + 144z^4 + 16z^3 - 24z^2 - 6z + 2)(\frac{\mathrm{d}}{\mathrm{d}z}U(z)) + (256z^9 - 128z^8 - 128z^7 - 32z^6 + 64z^5 + 24z^4 - 16z^3 - 2z^2 + z)(\frac{\mathrm{d}^2}{\mathrm{d}z^2}U(z))$$

with the initial condition $U(0) = 0$.

Thus, we can efficiently compute the coefficient u_n using the P-recurrence:

$$\begin{aligned}
&\left(256n^2+256n\right)u_n + && \left(-128n^2-640n-512\right)u_{n+1} + \\
&\left(-128n^2-704n-880\right)u_{n+2} + && \left(-32n^2-64n+152\right)u_{n+3} + \\
&\left(64n^2+592n+1324\right)u_{n+4} + && \left(24n^2+232n+540\right)u_{n+5} + \\
&\left(-16n^2-200n-616\right)u_{n+6} + && \\
&\left(-2n^2-32n-128\right)u_{n+7} + && \left(n^2+17n+72\right)u_{n+8} = 0
\end{aligned}$$

with the initial conditions $u_0 = 0, u_1 = 0, u_2 = 1, u_3 = 0, u_4 = 1, u_5 = 1, u_6 = 2, u_7 = 2$

The Taylor series expansion of $U(z)$ gives $z^2 + z^4 + z^5 + 2z^6 + 2z^7 + 7z^8 + 5z^9 + 20z^{10} + 19z^{11} + 60z^{12} + 62z^{13} + 202z^{14} + 202z^{15}$... with coefficients of z matching the number of solutions of the predicate `uniquelyClosable/2` for sizes given by the exponents of z.

Let us notice that the polynomial factor in the asymptotics is not in $n^{-3/2}$ as it is universal for tree-like structures. Here we have an interesting polynomial factor in $n^{-5/4}$ which appears when two square-root singularities coalesce. This fact has a positive effect on the performance of the Boltzmann random sampling.

5 Typable and Untypable Closable Skeletons

Let us denote $x : T$ the fact that x has type T. In the simply typed lambda calculus, given a context (a set of *variable:type* pairs), types are inferred using the following rules:

1. if $x : T$ in a context then $x : T$
2. term constants (if part of the language) have appropriate base type
3. if in a context, x has type S and an expression e has type T, then in the same context, with the binding of x removed from the context, $\lambda x.e$ has type $S \to T$
4. if in a context the term e has type $S \to T$ and the term f has type S then the application of e to f has type T.

We call a Motzkin skeleton *typable* if it exists at least one simply-typed closed lambda term having it as its skeleton. An *untypable* skeleton is a closable skeleton for which no such term exists.

We will follow the interleaving of term generation, checking for closedness and type inference steps shown in [12], but split it into a two stage program, with the first

stage generating code to be executed, via Prolog's metacall by the second, while also ensuring that the terms generated by the second stage are closed.

The predicate `genSkelEqs/4` generates type unification equations, that, if satisfied by a closed lambda term, ensure that the term is simply-typable.

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

genSkelEqs(v,V,Vs,(el(V,Vs),Es),Es)-->{Vs=[_|_]}.
genSkelEqs(l(A),(S->T),Vs,Es1,Es2)-->l,genSkelEqs(A,T,[S|Vs],Es1,Es2).
genSkelEqs(a(A,B),T,Vs,Es1,Es3)-->a,genSkelEqs(A,(S->T),Vs,Es1,Es2),
  genSkelEqs(B,S,Vs,Es2,Es3).

el(V,Vs):-member(V0,Vs),unify_with_occurs_check(V0,V).
```

Note that each lambda binder adds a new type variable to the list (starting empty at the root) on the way down to a leaf node. A term is then closed if the list of those variables `Vs` is not empty at each leaf node.

Thus, to generate the typable terms, one simply *executes the equations* `Eqs`, as shown by the predicate `typableClosedTerm/2`.

```
typableClosedTerm(N,Term):-genSkelEqs(N,Term,_,Eqs),Eqs.
```

The predicate `typableSkel/2` generates skeletons that are typable by running the same equations `Eqs` and ensuring they have *at least one solution* using the Prolog built-in `once/1`. The predicate `untypableSkel/2` succeeds, when the negation of these equations succeeds, indicating that no simply-typed lambda term exists having the given skeleton. Clearly, this is much faster than naively generating all the closed lambda terms and then finding their distinct skeletons.

```
typableSkel(N,Skel):-genSkelEqs(N,Skel,_,Eqs),once(Eqs).
untypableSkel(N,Skel):-genSkelEqs(N,Skel,_,Eqs),not(Eqs).
```

In Fig. 3 we show 3 typable and 3 untypable Motzkin skeletons.

An interesting question arises at this point about the relative density of closable and typable skeletons. Fig. 4, shows how many typable skeletons are among the closable skeletons for sizes up to `18`. We leave as an *open problem* finding out the asymptotic behavior of the relative density of the typable skeletons in the set of closable ones.

6 Uniquely Typable Skeletons and their Relation to Uniquely Closable Skeletons

A *uniquely typable skeleton* is one for which it exists exactly one simply-typed closed lambda term having it as a skeleton.

The predicate `uniquelyTypableSkel/2` generates unification equations for which, with the use of the built-in `findnsols/4`, it ensures efficiently that they have unique solutions. Fig. 5 shows the counts of the skeletons it generates up to size `21`.

```
uniquelyTypableSkel(N,Skel):-
  genSkelEqs(N,Skel,_,Eqs),has_unique_answer(Eqs).

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

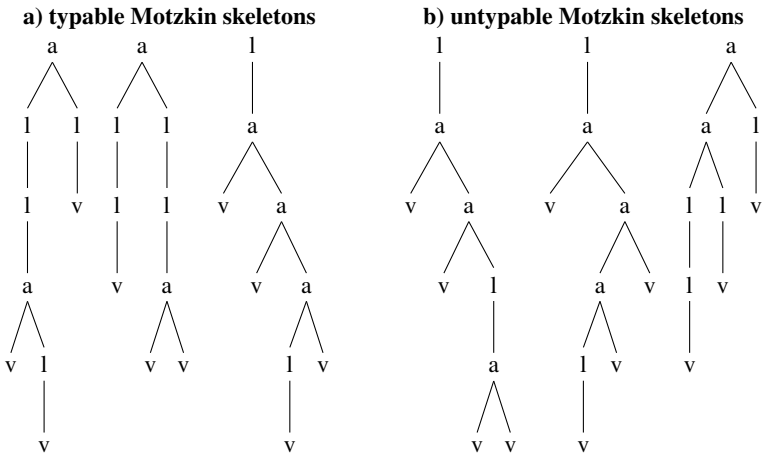


Fig. 3. Typable vs. untypable skeletons of size 8

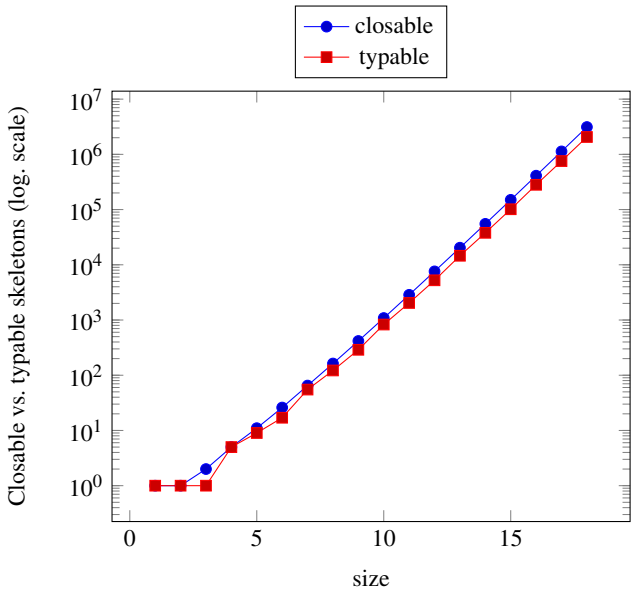


Fig. 4. Closable skeletons vs. typable skeletons by increasing sizes

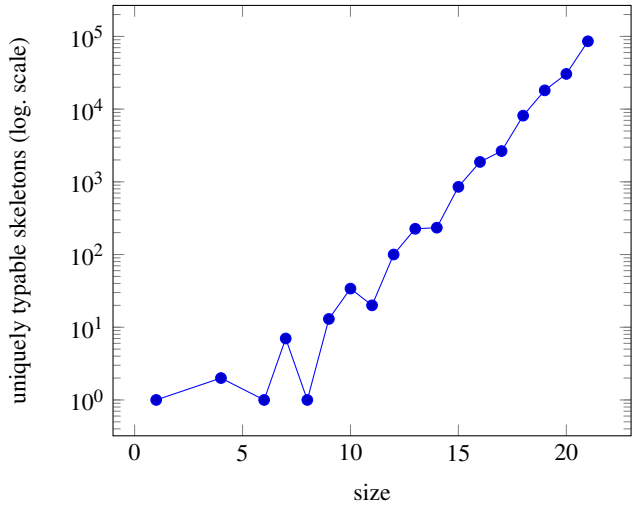


Fig. 5. Uniquely typable skeletons by increasing sizes

The natural question arises at this point: are there (uniquely) typable skeletons among the set of uniquely closable ones? The predicate `uniquelyTypableSkel/2` generates them by filtering the answer stream of `uniquelyClosable/2` with the predicate `isUniquelyTypableSkel/1`.

```
uniquelyClosableTypable(N,X):-
  uniquelyClosable(N,X),isUniquelyTypableSkel(X).

isUniquelyTypableSkel(X):-skelType(X,_).
```

The predicate isUniquelyTypableSkel/2 works by trying to infer the simple type of a uniquely typable lambda term corresponding to the skeleton. Note that this is a specialization of a general type inferencer to the case when exactly one type variable is available for each leaf node.

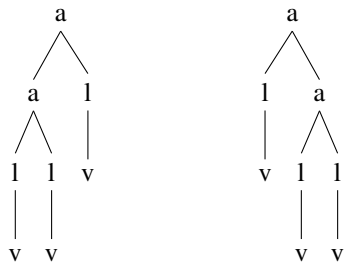
```
skelType(X,T):-skelType(X,T,[]).

skelType(v,V,[V0]):-unify_with_occurs_check(V,V0).
skelType(l(A),(S->T),Vs):-skelType(A,T,[S|Vs]).
skelType(a(A,B),T,Vs):-skelType(A,(S->T),Vs),skelType(B,S,Vs).
```

Proposition 5 *Uniquely closable typable skeletons of size* $3n+1$ *are in bijection with Catalan objects (binary trees) of size n.*

Proof. We will exhibit a simple bijection to binary trees. We want to show that terminal subtrees must be of the form l(v(0)). As there's a unique lambda above each leaf, closing it, the leaf should be (in de Bruijn notation), v(0) pointing to the first and only lambda above it. Assume a terminal node of the form a(v(0),v(0)). Then the two leaves must

share a lambda binder resulting in a circular term when unifying their types (i.e., as in the case of the well-known term ω=l(a(v(0),v(0)))) and thus it could not be typable.

The following two trees illustrate the shape of such skeletons and their bijection to binary trees. Note that the skeleton is mapped into a binary tree simply by replacing its terminal subtrees of the form `l(v)` with a leaf node `v`.



In this case, terms of size $3n+1 = 7 = 2+2+1+1+1+0+0+0$ *are mapped to binary trees of size* $n = 2 = 1+1+0+0+0$ *(with* `a/2 nodes` *there counted as 1 and v/0 nodes as 0) after replacing* `l(v)` *nodes with* `v` *nodes*.

As a consequence, *each uniquely closable term that is typable is uniquely typable*, as identity functions of the form `l(v(0))` would correspond to the end of each path from the root to a leaf in a lambda term having this skeleton. This tells us that there are no "interesting" uniquely closable terms that are typable. However, as there are normalizable terms that are not simply typed, an interesting *open problem* is to find out if uniquely closable terms, other than those ending with `l(v(0))` are (weakly) normalizable.

7 Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [4].

The first paper where de Bruijn indices are used in counting lambda terms is [8], which also uses a size definition equivalent to ours (but shifted by 1). The idea of using Boltzmann samplers for lambda terms was first introduced in [13].

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [14, 15]. However, the concepts of closable and typable skeletons of lambda terms and their uniquely closable and typable variants are new and have not been studied previously. The second author has used extensively Prolog as a meta-language for the study of combinatorial and computational properties of lambda terms in papers like [16, 12] covering different families of terms and properties, but not in combination with the precise analytic methods as developed in this paper.

It has been a long tradition in logic programming to use step-by-step program transformations to derive semantically simpler as well as more efficient code, going back as far as [17], that we have informally followed. In [18] a general constraint logic programming framework is defined for size-constrained generation of data structures as well as a program-transformation mechanism. By contrast, we have not needed to use

constraint solvers in our code as our derivation steps allowed us to place constraints explicitly at the exact program points where they were needed, for both the case of closable and typable skeletons. Keeping our programs close to Horn Clause Prolog has helped deriving CF-grammars for closable and uniquely closable skeletons and has enabled the use of tools from analytic combinatorics to fully understand their asymptotic behavior.

8 Conclusion

We have used simple program transformations to derive more efficient or conceptually simpler logic programs in the process of attempting to state, empirically study and solve some interesting problems related to the combinatorics of lambda terms. Several open problems have also been generated in the process with interesting implications on better understanding structural properties of the notoriously hard set of simply-typed lambda terms.

The lambda term skeletons introduced in the paper involve abstraction mechanisms that "forget" properties of the difficult class of simply-typed closed lambda terms to reveal classes of terms that are easier to grasp with analytic tools. In the case of the combinatorially simpler set of closed lambda terms, we have found that interesting subclasses of their skeletons turn out to be easier to handle. The case of uniquely closable terms turned out to be covered by a context free grammar, after several program transformation steps, and thus amenable to study analytically.

The focus on uniquely closable and uniquely typable Motzkin-tree skeletons of lambda terms, as well as their relations, has shown that closability and typability are properties that *predetermine* which lambda terms in de Bruijn notation can have such Motzkin trees as skeletons. Our analytic and experimental study has shown exponential growth for each of these families and suggests possible uses as positive or negative lemmas for all-term and random lambda term generation in dynamic programming algorithms.

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 anonymous reviewers of LOPSTR'17 for their careful reading and valuable suggestions, Maciej Bendkowski for salient comments on an earlier draft of this paper and the participants of the 10th Workshop on Computational Logic and Applications (`https:/cla.tcs.uj.edu.pl`) for enlightening discussions on the combinatorics of lambda terms.

References

1. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics. Revised edn. Volume 103. North Holland (1984)
2. 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
3. Howard, W.: The formulae-as-types notion of construction. In Seldin, J., Hindley, J., eds.: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, London (1980) 479–490
4. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1991)
5. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. AST'11, New York, NY, USA, ACM (2011) 91–97
6. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12** (1 2012) 67–96
7. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences. (2017) Published electronically at https://oeis.org/.
8. Lescanne, P.: On counting untyped lambda terms. Theoretical Computer Science **474** (2013) 80 – 97
9. Bendkowski, M.: Boltzmann-brain. (2017) Software (Haskell stack module), published electronically at `https://github.com/maciej-bendkowski/boltzmann-brain`.
10. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. 1 edn. Cambridge University Press, New York, NY, USA (2009)
11. Flajolet, P., Odlyzko, A.: Singularity analysis of generating functions. SIAM Journal on discrete mathematics **3**(2) (1990) 216–240
12. 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
13. Grygiel, K., Lescanne, P.: Counting and generating terms in the binary lambda calculus. J. Funct. Program. **25** (2015)
14. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5) (2013) 594–628
15. Bodini, O., Gardy, D., Jacquot, A.: Asymptotics and random sampling for BCI and BCK lambda terms. Theoretical Computer Science **502** (2013) 227 – 238
16. 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
17. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. The Journal of Logic Programming **19** (1994) 261–320
18. Fioravanti, F., Proietti, M., Senni, V.: Efficient generation of test data structures using constraint logic programming and program transformation. Journal of Logic and Computation **25**(6) (2015) 1263–1283