On a Uniform Representation of Combinators, Arithmetic, Lambda Terms and Types

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
http://www.cse.unt.edu/~tarau

Abstract

A uniform representation, as binary trees with empty leaves, is given to expressions built with Rosser's X-combinator, natural numbers, lambda terms and simple types. Type inference, normalization of combinator expressions and lambda terms in de Bruijn notation, ranking/unranking algorithms and tree-based natural numbers are described as a literate Prolog program.

With sound unification and compact expression of combinatorial generation algorithms, logic programming is shown to conveniently host a declarative playground where interesting properties and behaviors emerge from the interaction of heterogenous but deeply connected computational objects.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Data types and structures

General Terms Algorithms, Languages, Theory

Keywords Rossser's X-combinator, tree-based numbering systems, lambda calculus, normalization of de Bruijn terms, type inference, bijective Gödel numberings, logic programming as metalanguage.

1. Introduction

Logic programming languages provide a convenient metalanguage for building declarative playgrounds for specification and experiments with data types and computations often taken from other programming paradigms.

Properties of logic variables, unification with occurs check, and exploration of solution spaces via backtracking facilitate compact algorithms for inferring types or generate terms for various calculi. This holds in particular for lambda terms [1] and their variable free variants, combinators. Lambda terms provide a foundation to modern functional languages, type theory and proof assistants and have been lately incorporated into mainstream programming languages including Java 8, C# and Apple's Swift. While possibly one of the most heavily researched computational objects, lambda terms and combinators offer an endless stream of surprises to anyone digging just deep enough below their intriguingly simple surface.

This paper tries to follow some of the consequences of a very simple idea: what can happen if combinators, their types, their computationally equivalent lambda terms would all share the same basic representation as the natural numbers that have been used as encodings of formulas and proofs in such important fundamental results as Gödel's incompleteness theorems, as well as for mundane purposes like doing arithmetic operations in a programming language.

We have shown in the past [25, 27–29] that arithmetic operations and encodings of various data structures can be performed with tree based numbering systems in average time and space complexity that is comparable with the traditional binary numbers. One of the properties that singles out such numbering systems is their ability to favor objects with a regular structure on which representation size and complexity of operations can be significantly better than with the usual bitstring representations.

At the same time, we have observed that Rosser's X-combinator expressions [7] (a 1-point basis for combinatory logic) can also be hosted, together with function application nodes on top of our ubiquitous binary tree representation. While the representation of X-combinator expressions and their types collapses with that of binary trees representing natural numbers, with a few additional steps, we can also derive size proportionate ranking and unranking algorithms for general lambda terms.

This results in a shared representation of combinators, simple types, natural numbers and general lambda terms defining a common declarative playground for experiments connecting their computational properties.

The paper is organized as follows. Section 2 introduces X-combinator trees together with a generator and an evaluation algorithm. Section 3 defines simple types for X-combinator expressions via their equivalent lambda terms, describes type inference algorithms on de Bruijn terms and X-combinator expressions. It also explores consequences of expressions and types sharing the same binary tree representation. Section 4 describes a normalization algorithm for de Bruijn terms, derives from it one for X-combinator trees and compares the two algorithms. Section 5 interprets X-combinator trees as natural numbers on which it defines arithmetic operations. Section 6 describes mappings from lambda terms to binary trees that lead to size-proportionate ranking and unranking algorithms for lambda terms. Section 7 explores consequences that emerge from interactions between such heterogeneous computational objects sharing the same binary tree representation. Section 8 discusses related work and section 9 concludes the paper.

The paper is structured as a literate Prolog program. The code, tested with SWI-Prolog 6.6.6 and YAP 6.3.4. is also available at `http://www.cse.unt.edu/~tarau/research/2015/xco.pro`.

PPDP'15, July 14 - 16, 2015, Siena, Italy.

ACM 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790526

2. X-Combinator Trees

Combinator expressions are lambda terms represented as binary trees having applications as internal nodes and closed lambda terms called *combinators* as leaves. A *combinator basis* is a set of combinators in terms of which any other combinators can be expressed.

The most well known basis for combinator calculus consists of $K = \lambda x0.\ \lambda x1.x0$ and $S = \lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))$. SK-combinator expressions can be seen as binary trees with leaves labeled with symbols S and K, having function applications as internal nodes. Together with the primitive operation of application, K and S can be used as a 2-point basis to define a Turing-complete language.

It is shown in [9] that a countable number of (somewhat artificially constructed) 1-point bases exist for combinator calculi, but we will focus here on *Rosser's X-combinator*, one of the simplest 1-point bases that is naturally connected through mutual definitions to the combinators K and S.

2.1 Rosser's X-combinator

A derivation of Rosser's X-combinator is described in [7].

Defined as $X = \lambda f.fKSK$, this combinator has the nice property of expressing both K and S in a symmetric way.

$$K = (XX)X \tag{1}$$

$$S = X(XX) \tag{2}$$

Moreover, as shown in [7] the following holds.

$$KK = XX = \lambda x0.\ \lambda x1.\ \lambda x2.x1 \tag{3}$$

As a result, X-combinator expressions are within a (small, see Prop. 1) constant factor of their equivalent SK-expressions.

Denoting "`x`" the empty leaf corresponding to the X-combinator and "`>`" the (non-associative, infix) constructor for the binary tree's internal nodes, the predicates `sT`, `kT` and `xxT` define the Prolog expressions for the S, K and $KK = XX$ combinators, respectively.

```
sT(x>(x>x)).
kT((x>x)>x).
xxT(x>x).
```

This symmetry is part of the motivation for choosing the X-combinator basis, rather than any of the more well-known ones (see [13]).

2.2 Generating the combinator trees

Prolog is an ideal language to define in a few lines generators for various classes of combinatorial objects. The predicate `genTree` generates X-combinator trees with a limited number of internal nodes.

```
genTree(x)-->[].
genTree(X>Y)-->down,genTree(X),genTree(Y).

down(From,To):-From>0,To is From-1.
```

Note the use of Prolog's definite clause grammars (DCGs) and the predicate `down/2` that counts downward the number of available internal nodes. The predicate `genTree/3` provides two interfaces: `genTree/2` that generates trees with exactly N internal nodes and `genTrees/2` that generates trees with N or less internal nodes.

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

genTrees(N,X):-genTree(X,N,_).
```

The predicate `tsize` defines the size of an X-combinator tree in terms of the number of its internal nodes.

```
tsize(x,0).
tsize((X>Y),S):-tsize(X,A),tsize(Y,B),S is 1+A+B.
```

EXAMPLE 1. *X-combinator trees with up to 3 internal nodes (and up to 4 leaves).*

```
?- genTrees(3,T).
T = x ;
T = (x>x) ;
T = (x> (x>x)) ;
T = (x> (x> (x>x))) ;
T = (x> ((x>x)>x)) ;
T = ((x>x)>x) ;
T = ((x>x)> (x>x)) ;
T = ((x> (x>x))>x) ;
T = (((x>x)>x)>x) .
```

2.3 An evaluator for the Turing-complete language of X-combinator trees

We can derive an evaluator for X-combinator trees from a well-known evaluator for SK-combinator trees.

```
eval((F>G),R):-!,eval(F,F1),eval(G,G1),app(F1,G1,R).
eval(X,X).
```

In the predicate `app/3` handling the application of the first argument to the second, we describe in the first two clauses the actions corresponding to `K` and `S`. The final clause returns the unevaluated application as its third argument.

```
app((((x>x)>x)>X),_Y,R):-!,R=X. % K
app((((x>(x>x))>X)>Y),Z,R):-!,  % S
  app(X,Z,R1),
  app(Y,Z,R2),
  app(R1,R2,R).
%app(((x>x)>_X)>Y),_Z,R):-!,R=Y.
%app((x>x)>x,(x>x)>x,R):-!,app(x,x,R).
app(F,G,(F>G)).
```

Note also the commented out clauses, that can shortcut some evaluation steps, using the identity (3).

EXAMPLE 2. *Evaluation of SKK and SKX, equivalent implementations of the identity combinator $I = \lambda x.x$.*

```
?- SKK=(((x>(x>x))>((x>x)>x))>((x>x)>x)),eval(SKK>x,R).
SKK = (((x>(x>x))>((x>x)>x))>((x>x)>x)),
R = x.

?- SKX=(((x> (x>x))> ((x>x)>x))>x),eval(SKX>x,R).
SKX = (((x> (x>x))> ((x>x)>x))>x),
R = x.
```

2.4 De Bruijn equivalents of X-combinator expressions

De Bruijn indices [6] provide a *name-free* representation of lambda terms. All terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation. Variables following lambda abstractions are omitted and their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* is found on the way up to the root of the term. We represent them using the constructor `a/2` for application, `l/1` for lambda abstractions (that we will call shortly *binders*) and `v/1` for marking the integers corresponding to the de Bruijn indices. For instance, $\lambda x0.(\ \lambda x1.(x0\ (x1\ x1))\ \ \lambda x2.(x0\ (x2\ x2)))$ becomes `l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0))))))`, corresponding to the fact that `v(1)` is bound by the outermost lambda (two steps away, counting from 0) and the occurrences of `v(0)` are bound each by the closest lambda, represented by the constructor `l/1`.

The predicates `kB` and `sB` define the K and S combinators in de Bruijn form.

```
kB(l(l(v(1)))).

sB(l(l(l(a(a(v(2),v(0)),a(v(1),v(0)))))).
```

We obtain the X-combinator's definition in terms of S and K, in de Bruijn form, by using the equation $Xf = fKSK$ derived from its lambda expression $\lambda f.fKSK$. The predicate `xB` implements it.

```
xB(X):-F=v(0),kB(K),sB(S),X=l(a(a(a(F,K),S),K)).
```

The predicate `t2b` transforms an X-combinator tree in its lambda expression form, in de Bruijn notation, by replacing leaves with the de Bruijn form of the X-combinator and replacing recursively the constructor "`>`"/2 with the application nodes "`a`"/2.

```
t2b(x,X):-xB(X).
t2b((X>Y),a(A,B)):-t2b(X,A),t2b(Y,B).
```

EXAMPLE 3. *Expansion of small X-combinator trees to de Bruijn forms.*

```
?- t2b(x,X).
X = l(a(a(a(v(0), l(l(v(1)))), l(l(l(a(a(v(2), v(0)),
                     a(v(1), v(0)))))), l(l(v(1))))).

?- t2b(x>x,XX).
XX=a(
     l(a(a(a(v(0),l(l(v(1)))),l(l(l(a(a(v(2),v(0)),
                     a(v(1),v(0)))))),l(l(v(1))))),
     l(a(a(a(v(0),l(l(v(1)))),l(l(l(a(a(v(2),v(0)),
                    a(v(1),v(0)))))),l(l(v(1))))
   ).
```

Clearly their de Bruijn equivalents are significantly larger than the corresponding combinator trees, but we will show that this is only by a constant factor. We will also see that often normalization can bring down significantly the size of such expressions, given that nodes like `x>x` are equivalent to smaller lambda expressions like $\lambda x0.\ \lambda x1.\ \lambda x2.x1$.

Similarly to the case of X-combinator trees, one can define the size of a lambda expression in de Bruijn form as the number of its internal nodes.

```
bsize(v(_),0).
bsize(l(A),R):-bsize(A,RA),R is RA+1.
bsize(a(A,B),R):-bsize(A,RA),bsize(B,RB),R is 1+RA+RB.
```

PROPOSITION 1. *The size of the lambda term equivalent to an X-combinator tree with N internal nodes is 15N+14.*

Proof Note that the an X-combinator tree with N internal nodes has N+1 leaves. The de Bruijn tree built by the predicate `t2b` has also N application nodes, and is obtained by having leaves replaced in the X-combinator term, with terms bringing 14 internal nodes each, corresponding to `x`. Therefore it has a total of $N + 14(N + 1) = 15N + 14$ internal nodes.

A lambda term is called *closed* if it contains no free variables. The predicate `isClosedB` defines this property for de Bruijn terms.

```
isClosedB(T):-isClosed1(T,0).

isClosed1(v(N),D):-N<D.
isClosed1(l(A),D):-D1 is D+1,isClosed1(A,D1).
isClosed1(a(X,Y),D):-isClosed1(X,D),isClosed1(Y,D).
```

PROPOSITION 2. *The lambda terms equivalent to X-combinators computed by* `t2b` *are closed.*

Proof As the lambda term equivalent of the X-combinator is clearly a closed expression, the proposition follows from the definition of `t2b`, as it builds terms that apply closed terms to closed terms.

Besides being closed, lambda terms interesting for functional languages and proof assistants are also well-typed. While the K and S combinators are known to be well-typed, we would like to see how this property extends to X-combinator trees. In particular, we would like to have an idea on the asymptotic density of well-typed X-combinator tree expressions. We will take advantage of Prolog's sound unification algorithm to define a type inferrer directly on de Bruijn terms.

3. Inferring simple types for X-combinator trees

A natural way to define types for combinator expressions is to borrow them from their lambda calculus equivalents. This makes sense, as they represent the same function i.e., they are extensionally the same.

We will start with an algorithm inferring types on the de Bruijn equivalents of X-combinator trees.

3.1 A type inference algorithm for Bruijn terms

Simple types will be defined here as binary trees built with the constructor "`>/2`" with empty leaves, representing the unique primitive type "`x`". Clearly, this is exactly the same representation as our X-combinator trees! Simple types can be seen as as a "binary tree approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization), as the following well known property states (e.g., [2]).

PROPOSITION 3. *Lambda terms that have simple types are strongly normalizing.*

We will say a that a term is *well-typed* if we can infer a simple type for it. While in a functional language inferring types requires implementing unification with occur check, as shown for instance in the appendix of [10], this is readily available in Prolog. We will closely follow here the Prolog implementation given in [30] but we will also give in subsection 3.3 an simpler equivalent algorithm working directly on X-combinator trees.

The predicate `btype/3` works by associating the same logic variable, denoting its type, to each of its occurrences. As logic variable bindings propagate between binders, this ensures that types are consistently inferred. Note that unification with occurs check needs to be used to avoid cycles in the inferred type formulas.

```
btype(v(I),V,Vs):-nth0(I,Vs,V0),
  unify_with_occurs_check(V,V0).
btype(a(A,B),Y,Vs):-btype(A,X>Y,Vs),btype(B,X,Vs).
btype(l(A),X>Y,Vs):-btype(A,Y,[X|Vs]).
```

Note also the use of the built-in `nth0(I,Vs,V0)` that unifies `V0` with the `I`-th element of the list `Vs`.

At this point, most general types are inferred by `btype` as fresh variables, somewhat similar to polymorphic types in functional languages, if one interprets logic variables as universally quantified.

EXAMPLE 4. *Type inferred for the* `S` *combinator* $\lambda x0.\ \lambda x1.\ \lambda x2.\ ((x0\ x2)\ (x1\ x2))$ *in de Bruijn form.*

```
?- btype(l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),T,[]),
   numbervars(T,0,_).
T = ((A> (B>C))> ((A>B)> (A>C))).
```

However, as we are only interested in simple types with only one basic type, we will bind uniformly the leaves of our type tree to the constant "`x`" representing our only primitive type, by using the predicate `bindType/1`.

```
btype(A,T):-btype(A,T,[]),bindType(T).

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

EXAMPLE 5. *Simple type inferred for the* S *combinator and failure to assign a type to the* Y *combinator* $\lambda x0.(\ \lambda x1.(x0\ (x1\ x1))\ \lambda x2.(x0\ (x2\ x2)))$.

```
?- btype(l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),T).
T = ((x> (x>x))> ((x>x)> (x>x))).
?- btype(l(a(l(a(v(1), a(v(0), v(0)))),
                        l(a(v(1), a(v(0), v(0)))))),T).
false.
```

3.2 Type trees as combinator trees

We can define the type of a combinator expression as the type of its lambda expression translation. The predicate `xtype` defines a function from binary trees to binary trees mapping an X-combinator expression to its type, as inferred on its equivalent lambda term in de Bruijn notation.

```
xtype(X,T):-t2b(X,B),btype(B,T).
```

Observe that this only makes sense if the combinator basis is well-typed. Fortunately this is the case of the X-combinator $\lambda x0.(((x0\ \lambda x1.\ \lambda x2.x1)\ \lambda x3.\ \lambda x4.\ \lambda x5.((x3\ x5)\ (x4\ x5)))\ \lambda x6.\ \lambda x7.x6)$.

EXAMPLE 6. *The X-combinator is well-typed.*

```
?- xtype(x,T).
T = (((x> (x>x))> (((x> (x>x))>
      ((x>x)> (x>x)))> ((x> (x>x))>x)))>x).
```

3.3 Inferring types of X-combinator trees directly

The predicate `xt`, that can be seen as a "partially evaluated" version of `xtype`, infers the type of the combinators directly.

```
xt(X,T):-poly_xt(X,T),bindType(T).

xT(T):-t2b(x,B),btype(B,T,[]).

poly_xt(x,T):-xT(T).
poly_xt(A>B,Y):-poly_xt(A,T),poly_xt(B,X),
  unify_with_occurs_check(T,(X>Y)).
```

It proceeds by first borrowing the type of `x` from its de Bruijn equivalent. Then, after calling `poly_xt` to infer polymorphic types, it binds them to our simple-type representation by calling `bindType`.

EXAMPLE 7. *Simple type inferred directly on X-combinator trees.*

```
?- skkT(X),xt(X,DirectT),xtype(X,BorrowedT).
X = (((x> (x>x))> ((x>x)>x))> ((x>x)>x)),
DirectT = BorrowedT, BorrowedT = (x>x).
```

3.4 Estimating the proportion of well-typed X-combinator trees

An interesting question arises at this point: what proportion of X-combinator trees of a given size are well-typed? While the analytic study of the *asymptotic density* has been successfully performed on several families of lambda terms [4, 10, 11], it is considered an open problem for well-typed terms. We will limit ourselves here to empirically estimate it, as it is done in [10] for general lambda terms, where experiments indicate extreme sparsity for very large terms.

We can use our generator `genTree` to enumerate X-combinator trees among which we can then count the number of well-typed ones.

EXAMPLE 8. *Types inferred for terms with 2 internal nodes.*

```
?- genTree(2,X),xtype(X,T).
X = (x> (x>x)),
T = ((x> (x>x))> ((x>x)> (x>x))) ;
X = ((x>x)>x),
T = (x> (x>x))
```

Term size	Well-typed	Total	Ratio
0	1	1	1
1	1	1	1
2	2	2	1
3	5	5	1
4	12	14	0.8571
5	38	42	0.9047
6	113	132	0.8560
7	357	429	0.8321
8	1148	1430	0.8027
9	3794	4862	0.7803
10	12706	16796	0.7564
11	43074	58786	0.7327
12	147697	208012	0.7100

Figure 1. Proportion of well-typed X-combinator terms

Figure 1 shows the counts for well-typed X-combinator expressions among the total binary trees of given size. Note that the total column is given by the Catalan numbers (entry A000108 in [22]), as binary trees are a member of the Catalan family of combinatorial objects [23]. Somewhat surprisingly, a large proportion of well-typed X-combinator terms is present among the binary trees of a given size, indicating the possible existence of a lower bound that might be easier to determine analytically than in the case of general lambda terms.

3.5 Generating closed well-typed terms of a given size

One can derive, from the type inferrer `btype`, a generator for de Bruijn terms with a given number of internal nodes, by controlling their creation with the predicate `down/2` in a way similar to the binary tree generator `genTree`.

Like the predicate `genTree`, the predicate `genTypedB/5` relies on Prolog's DCG notation to thread together the steps controlled by the predicate `down`. Note also the nondeterministic use of the built-in `nth0` that enumerates values for both `I` and `V` ranging over the list of available variables `Vs`, as well as the use of `unify_with_occurs_check` to ensure that unification of candidate types does not create cycles.

```
genTypedB(v(I),V,Vs)--> {
   nth0(I,Vs,V0),
   unify_with_occurs_check(V,V0)
  }.
genTypedB(a(A,B),Y,Vs)-->down,
  genTypedB(A,X>Y,Vs),
  genTypedB(B,X,Vs).
genTypedB(l(A),X>Y,Vs)-->down,
  genTypedB(A,Y,[X|Vs]).
```

Like in the case of `genTree`, two interfaces are offered: `genTypedB` that generates de Bruijn terms of with exactly `L` internal nodes and `genTypedBs` that generates terms with `L` internal nodes or less.

```
genTypedB(L,B,T):-genTypedB(B,T,[],L,0),bindType(T).

genTypedBs(L,B,T):-genTypedB(B,T,[],L,_),bindType(T).
```

As expected, the number of solutions, $1, 2, 9, 40, 238, 1564, \ldots$ for sizes for sizes $1, 2, 3, \ldots$ matches entry `A220471` in [22].

EXAMPLE 9. *Generation of well-typed closed de Bruijn terms of size 3.*

```
?- genTypedB(3,Term,Type).
Term = a(l(v(0)), l(v(0))),Type = (x>x) ;
Term = l(a(v(0), l(v(0)))),Type = (((x>x)>x)>x) ;
```

```
Term = l(a(l(v(0)), v(0))),Type = (x>x) ;
Term = l(a(l(v(1)), v(0))),Type = (x>x) ;
Term = l(l(a(v(0), v(1)))),Type = (x> ((x>x)>x)) ;
Term = l(l(a(v(1), v(0)))),Type = ((x>x)> (x>x)) ;
Term = l(l(l(v(0)))),Type = (x> (x> (x>x))) ;
Term = l(l(l(v(1)))),Type = (x> (x> (x>x))) ;
Term = l(l(l(v(2)))),Type = (x> (x> (x>x))) .
```

3.6 Querying the generator for specific types

Coming with Prolog's unification and non-deterministic search, is the ability to make more specific queries by providing a type pattern, that selects only terms of a given type.

EXAMPLE 10. *Terms of type* `x>x` *of size 4.*

```
?- genTypedB(4,Term,(x>x)).
Term = a(l(v(0)), l(v(0))) ;
Term = l(a(l(v(1)), l(v(0)))) ;
Term = l(a(l(v(1)), l(v(1)))) .

?- genTypedBs(12,T,(x>x)>x).
false.
```

Note that the last query, taking about a minute, shows that no closed terms of type `(x>x)>x` exist up to size 12.

We can make use of our generator's efficient specialization to a given type to explore empirical estimates for some interesting function types.

Contrary to the total absence of type `(x>x)>x` among terms of size up to 12, "binary operations" of type `x>(x>x)` turn out to be quite frequent, giving, by increasing sizes, the sequence [0, 2, 0, 14, 12, 201, 445, 4632, 17789, 158271, 891635].

Transformers of type `x>x`, by increasing sizes, give the sequence [1, 0, 3, 3, 31, 78, 596, 2500, 18474, 110265, 888676]. While type `(x>x)>x` turns our to be absent up to size 12, the type `(x>x)>(x>x)` describing *transformers of transformers* turns out to be quite popular, as shown by the sequence [1,1, 4, 11, 55, 227, 1315, 7066, 46731, 309499, 2358951]. The same turns out to be tree also for `(x>x)>((x>x)>(x>x))`, giving [0, 2, 1, 16, 29, 272, 940, 7594, 39075, 312797, 2115374] and `((x>x)>(x>x)) > ((x>x)>(x>x))` giving [1, 1, 5, 13, 73, 300, 1846, 10130, 69336, 469217, 3640134]. One might speculate that homotopy type theory [31], that focuses on such transformations and transformations of transformations etc. has a rich population of lambda terms from which to chose interesting inhabitants of such types!

3.7 Generating terms with free variables

Another interface, generating closed simply-typed terms of given size and with at most a given number of free de Bruijn indices is given by the predicate `genTypedWithSomeFree`.

```
genTypedWithSomeFree(Size,NbFree,B,T):-
   between(0,NbFree,NbVs),
   length(FreeVs,NbVs),
   genTypedB(B,T,FreeVs,Size,0),
   bindType(T).
```

The first 9 numbers counting closed simply-typed terms with at most one free variables (not yet in [22]), are [3, 10, 45, 256, 1688, 12671, 105743, 969032, 9639606].

Note that, as our generator integrates filtering out closed and untypable terms early the generation process rather than as a post-processing step, enumeration and counting of these terms happens in a few seconds.

3.8 Generating closed typable lambda terms by types

In [19] generation of random terms is guided by their types, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some bugs in the Glasgow Haskell Compiler (GHC).

We can organize in a similar way the interface of our combined generator and type inferrer. The predicate `genByTypeB` first generates types (seen simply as binary trees) with `genTree` and then uses the unification-based querying mechanism to generate all the de Bruijn terms terms with fewer internal nodes then their binary tree type.

```
genByTypeB(L,B,T):-
  genTree(L,T),
  genTypedBs(L,B,T).
```

EXAMPLE 11. *Enumeration of closed simply-typed de Bruijn terms with types of size 3 and less then 3 internal nodes.*

```
?- genByTypeB(3,B,T).
B = l(l(l(v(0)))),T = (x> (x> (x>x))) ;
B = l(l(l(v(1)))),T = (x> (x> (x>x))) ;
B = l(l(l(v(2)))),T = (x> (x> (x>x))) ;
B = l(l(v(1))),T = (x> ((x>x)>x)) ;
B = l(l(a(v(0), v(1)))),T = (x> ((x>x)>x)) ;
B = a(l(v(0)), l(v(0))),T = ((x>x)> (x>x)) ;
B = l(v(0)),T = ((x>x)> (x>x)) ;
B = l(a(l(v(0)), v(0))),T = ((x>x)> (x>x)) ;
B = l(a(l(v(1)), v(0))),T = ((x>x)> (x>x)) ;
B = l(l(v(0))),T = ((x>x)> (x>x)) ;
B = l(l(a(v(1), v(0)))),T = ((x>x)> (x>x)) ;
B = l(a(v(0), l(v(0)))),T = (((x>x)>x)>x).
```

Given that various constraints are naturally interleaved by our generator we obtain in a few seconds the sequence counting these terms having types up to size 8, [1, 2, 12, 72, 702, 7970, 111573, 1867256]. It indicates that the number of terms sharing the same type grows very fast, despite of the experimentally observed (see [10]) quickly decreasing density among the set of all (closed) lambda terms.

3.9 Iterated types

EXAMPLE 12. *As an interesting coincidence, one might note that the binary tree representation of the type of the K combinator is nothing but the S combinator itself.*

```
?- kT(K),xtype(K,T),sT(S).
K = ((x>x)>x),
T = S, S = (x> (x>x)).
```

Given that X-combinator expressions and their inferred simple types are both represented as binary trees of often comparable sizes, one might be curious about what happens if we iterate this process.

By interpreting a type as its identically represented X-combinator expression, one can ask the question: is the type expression itself well-typed? If so, is the set of distinct iterated types starting from an X-combinator finite?

The predicate `iterType` applies the type inference operation at most K-times, until an untypable term or a fixpoint is reached.

```
iterType(K,X, Ts, Steps):-
  iterType(K,FinalK,X,[],Rs),
  reverse(Rs,Ts),
  Steps is K-FinalK.

iterType(K,FinalK,X,Xs,Ys):-K>0,K1 is K-1,
  xtype(X,T),
  \+(member(T,Xs)),
  !,
  iterType(K1,FinalK,T,[T|Xs],Ys).
iterType(FinalK,FinalK,_,Xs,Xs).
```

EXAMPLE 13. *Iterated types for K and S and I=SKK combinators.*

```
?- kT(K),iterType(100,K,Ts,Steps).
K = ((x>x)>x),
```

Initial term size	Average steps	Average size
0	1	7
1	4	3
2	3	3.25
3	2.4	7.2799
4	2.5714	4.9476
5	2.8333	5.5087
6	2.5075	6.1571
7	2.4405	6.6171
8	2.3832	7.0235
9	2.3290	7.4627
10	2.2547	7.9913
11	2.1831	8.5392
12	2.1174	9.1143

Figure 2. Average steps and term sizes of iterated types

```
Ts = [x> (x>x), (x> (x>x))> ((x>x)> (x>x)), (x>x)> (x>x)],
Steps = 3.

?- sT(S),iterType(100,S,Ts,Steps).
S = (x>(x>x)),
Ts = [(x> (x>x))>((x>x)> (x>x)),(x>x)>(x>x),x> (x>x)],
Steps = 3.

?- skkT(XX),iterType(100,XX,Ts,Steps).
XX = (((x> (x>x))> ((x>x)>x))> ((x>x)>x)),
Ts = [x>x, x> (x> (x>x)), x> (x>x),
      (x> (x>x))> ((x>x)> (x>x)), (x>x)> (x>x)],
Steps = 5.
```

Figure 2 shows the average number of steps until a non-typable term is found or a fixpoint is reached as well as the average size of the terms in the sequence of iterated types.

This matches the intuition that types are (smaller) approximations of programs and suggests that the following holds.

Conjecture. The set of iterated types is finite for any X-combinator tree.

4. Evaluation via de Bruijn terms

While, as shown in subsection 2.3, X-combinator trees can be evaluated directly, it makes sense to investigate if more compact equivalent normal forms can be obtained for them via their mapping to lambda terms.

4.1 Normalization of de Bruijn terms

Evaluation of lambda terms involves β*-reduction*, a transformation of a term like `a(l(X,A),B)` by replacing every occurrence of `X` in `A` by `B`, (with possible variable renamings to avoid variable capture) and η*-conversion*, the transformation of an application term `l(X,a(A,X))` into `A`, under the assumption that `X` does not occur in `A`. For terms in de Bruijn form, this occurs check is not needed as variables are indirectly represented as offsets indicating where their binders are.

The first tool we need to implement normalization of lambda terms is a safe substitution operation. While logic variables offer a fast and easy way to perform *substitutions*, they do not offer any elegant mechanism to ensure that substitutions are *capture-free*. Moreover, no HOAS-like mechanism exists in Prolog for borrowing anything close to *normal order reduction* from the underlying system, as Prolog would provide, through meta-programming, only a *call-by-value* model.

Normalization algorithms for of lambda terms are well known (e.g., [21]). To be able to compare it with direct evaluation of combinator trees, we will describe here, following [30], a Prolog implementation of an interpreter for lambda terms supporting normal order β-reduction using de Bruijn terms. This also ensures that terms are unique up to α-equivalence. As usual, we will omit η-conversion, known to interfere with things like type inference, as the redundant argument(s) that it removes might carry useful type information.

The predicate `beta/3` implements the β-conversion operation corresponding to the binder `l(A)`. It calls `subst/4` that replaces in `A` occurrences corresponding the the binder `l/1`.

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

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

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

When the right occurrence `v(N)` is reached, the term substituted for it is shifted such that its variables are marked with the new, incremented distance to their binders. The predicate `shift_var/4` implements this operation.

```
shift_var(I,K,a(A,B),a(RA,RB)):-K>=0,I>=0,
  shift_var(I,K,A,RA),
  shift_var(I,K,B,RB).
shift_var(I,K,l(A),l(R)):-K>=0,I>=0,K1 is K+1,
  shift_var(I,K1,A,R).
shift_var(I,K,v(N),v(M)):-K>=0,I>=0,N>=K,M is N+I.
shift_var(I,K,v(N),v(N)):-K>=0,I>=0,N<K.
```

We first compute the *weak head normal form* using `wh_nf/2`.

```
wh_nf(v(X),v(X)).
wh_nf(l(E),l(E)).
wh_nf(a(X,Y),Z):-wh_nf(X,X1),wh_nf1(X1,Y,Z).
```

The predicate `wh_nf1/3` does the case analysis of application terms `a/2`. The key step is the β-reduction in its second clause, when it detects an "eliminator" lambda expression as its left argument, in which case it performs the substitution of its binder, with its right argument.

```
wh_nf1(v(X),Y,a(v(X),Y)).
wh_nf1(l(E),Y,Z):-beta(l(E),Y,NewE),wh_nf(NewE,Z).
wh_nf1(a(X1,X2),Y,a(a(X1,X2),Y)).
```

The predicate `evalB` implements *normal order reduction*. Normal order reduction of a lambda term, if it terminates, leads to a unique normal form, as a consequence of the Church-Rosser theorem, elegantly proven in [6] using de Bruijn terms. Termination holds, for instance, in the case of simply-typed lambda terms. The predicate `evalB` follows the same skeleton as `wh_nf`, which is called in the third clause to perform reduction to weak head normal form, starting from the outermost lambda binder. Note also that, unlike in `wh_nf`, in the second clause, lambdas are traversed and their subterms evaluated.

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

Case analysis of application terms for possible β-reduction is performed by `applyB/3`, where the second clause calls `beta/3` and recurses on its result.

```
applyB(v(E1),E2,a(v(E1),NE2)):-evalB(E2,NE2).
applyB(l(E),E2,R):-beta(l(E),E2,NewE),evalB(NewE,R).
applyB(a(A,B),E2,a(NE1,NE2)):-
  evalB(a(A,B),NE1),
  evalB(E2,NE2).
```

The predicate `evalB` provides a lambda calculus interpreter working on de Bruijn terms. It is guaranteed to compute a normal form, if it exists.

EXAMPLE 14. *Evaluation of the lambda term* $SKK = ((\lambda x0.\,\lambda x1.\,\lambda x2.((x0\,x2)\,(x1\,x2))\;\lambda x3.\,\lambda x4.x3)\;\lambda x5.\,\lambda x6.x5)$ *in de Bruijn form, resulting in the definition of the identity combinator* $I = \lambda x0.x0$.

```
?- evalB(a(a(l(l(l(a(a(v(2), v(0)),
         a(v(1), v(0))))))), l(l(v(1)))), l(l(v(1)))),R).
R = l(v(0)) .
```

4.2 Comparing the two evaluators

One can now compare the evaluation performed on X-combinator trees to that performed on their corresponding lambda expressions. The predicate `evalAsT` first evaluates and then converts while the predicate `evalAsB` first converts to a de Bruijn terms and then evaluates it, with opportunities for additional reductions.

```
evalAsT --> eval,t2b.
evalAsB --> t2b,evalB.
```

We express these two predicates as a *composition of functions* (first argument in, second out) using Prolog's DCG notation.

EXAMPLE 15. *Additional reductions obtained from a term of size 29 to a term of size 3 on the de Bruijn terms associated to an X-combinator expression.*

```
?- evalAsT(x>x,R),bsize(R,Size),write(Size),nl,fail.
29

?- evalAsB(x>x,R),bsize(R,Size).
R = l(l(l(v(1)))),
Size = 3 .
```

Note however, as predicted by the Church-Rosser theorem [1, 6], applying normalization via `evalB` to the result of `evalAsT` reaches the same final normal form. This property is called *confluence*.

EXAMPLE 16. *Confluence of evaluation as X-combinator tree and as lambda term.*

```
?- evalAsT(x>x,R),evalB(R,FinalR).
R = a(l(a(a(a(v(0),...,l(l(v(1)))))),
FinalR = l(l(l(v(1)))) .
```

5. X-combinator trees as natural numbers

Gödel numberings seen as injective mappings from formulas and proofs to natural numbers have been used for important theoretical results in the past [12] among which Gödel's incompleteness theorems are the most significant [8].

In the form of ranking and unranking functions, bijections from families of combinatorial objects to natural numbers have been devised with often practical uses in mind, like generation of random inputs for software testing.

Ensuring that such bijections are also size-proportionate, adds an additional challenge to the problem, as the fast growth of the number of combinatorial objects of a given size makes it difficult to impossible to associate to all of them comparably small unique natural numbers. As another challenge, computation of the unranking function often involves some form of binary or multiway tree search to locate the object corresponding to a given natural number [10, 25], which precludes their use on very large objects. Our solution described here consists in two steps, the second one involving an arguably surprising twist.

First, we define a bijection between natural numbers and trees. Next we define arithmetic operations directly on trees and ensure that they mimic exactly their natural number equivalents. *This turns our trees into natural numbers (they become yet another model or Peano's axioms), hence we can make them the target of ranking algorithms and the source of unranking ones.*

As we are now dealing with bijections between trees and tree-like data structures, making them size proportionate becomes surprisingly easy. We will define such a bijection to general lambda terms in section 6.

5.1 A bijection from binary trees to natural numbers

The (big-endian) binary representation of a natural number can be written as a concatenation of binary digits of the form

$$n = b_0^{k_0} b_1^{k_1} \dots b_i^{k_i} \dots b_m^{k_m} \tag{4}$$

with $b_i \in \{0, 1\}$ and the highest digit $b_m = 1$. The following hold.

PROPOSITION 4. *An even number of the form* $0^i j$ *corresponds to the operation* $2^i j$ *and an odd number of the form* $1^i j$ *corresponds to the operation* $2^i(j + 1) - 1$.

Proof It is clearly the case that $0^i j$ corresponds to multiplication by a power of 2. If $f(i) = 2i+1$, then it can be shown by induction that the i-th iterate of f, f^i is computed as in the equation (5)

$$f^i(j) = 2^i(j + 1) - 1 \tag{5}$$

Observe that each block 1^i in n, represented as $1^i j$ in equation (4), corresponds to the iterated application of f, i times, $n = f^i(j)$.

PROPOSITION 5. *A number* n *is even if and only if it contains an even number of blocks of the form* $b_i^{k_i}$ *in equation (4). A number* n *is odd if and only if it contains an odd number of blocks of the form* $b_i^{k_i}$ *in equation (4).*

Proof It follows from the fact that the highest digit (and therefore the last block in big-endian representation) is 1 and the parity of the blocks alternate.

This suggests defining a `cons` operation on natural numbers as follows.

$$cons(i, j) = \begin{cases} 2^{i+1} j & \text{if } j \text{ is odd,} \\ 2^{i+1}(j + 1) - 1 & \text{if } j \text{ is even.} \end{cases} \tag{6}$$

Note that the exponents are $i + 1$ instead of i as we start counting at 0. Note also that $cons(i, j)$ will be even when j is odd and odd when j is even.

PROPOSITION 6. *The equation (6) defines a bijection* $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+ = \mathbb{N} - \{0\}$.

Therefore `cons` has an inverse `decons`, that we will constructively define together with it.

```
cons(I,J,C) :- I>=0,J>=0,
  D is mod(J+1,2),
  C is 2^(I+1)*(J+D)-D.
```

The definition of the inverse `decons` relies on the *dyadic valuation* of a number n, $\nu_2(n)$, defined as the largest exponent of 2 dividing n, implemented as the helper predicate `dyadicVal`, which computes the least significant bit of its first argument with help from the built-in `lsb`.

```
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.

dyadicVal(KB,I,J):-I is lsb(KB),J is KB // (2^I).
```

EXAMPLE 17. *The inverse* cons *and* decons *operations.*

```
?- decons(2016,A,B),cons(A,B,N).
A = 4,
B = 63,
N = 2016.
```

We can compute a natural number from an X-combinator tree by mapping recursively the "**>**" constructor to cons.

```
n(x,0).
n((A>B),K):-n(A,I),n(B,J),cons(I,J,K).
```

Similarly, we can build an X-combinator tree from a natural number by recursing over decons.

```
t(0,x).
t(K,(A>B)):-K>0,decons(K,I,J),t(I,A),t(J,B).
```

Note the small codes corresponding to some interesting combinators.

EXAMPLE 18. *Encodings of combinators X, S, K and XX=KK.*

```
?- n(x,N).
N = 0.
?- n(x>x,N).
N = 1.
?- sT(X),n(X,N).
X = (x> (x>x)), N = 2.
?- kT(X),n(X,N).
X = ((x>x)>x), N = 3.
```

PROPOSITION 7. *The predicates* n *and* t *define inverse functions between natural numbers and X-combinator trees.*

Proof It follows from the fact that cons and decons implement inverse functions.

EXAMPLE 19. *The work of* t *and* n *on the first 8 natural numbers.*

```
?- maplist(t,[0,1,2,3,4,5,6,7],Ts),maplist(n,Ts,Ns).
Ts = [x,x>x,x> (x>x), (x>x)>x, (x>x)> (x>x),
      x> (x> (x>x)),x> ((x>x)>x), (x> (x>x))>x],
Ns = [0, 1, 2, 3, 4, 5, 6, 7].
```

5.2 Binary tree arithmetic

As we know for sure that natural numbers support arithmetic operations, we will try to mimic their behavior with binary trees built with the constructor "**>**" and empty leaves x that we have interpreted so far as X-combinator expressions and simple types.

The operations even_ and odd_ implement the observation following from of Prop. 5 that parity (staring with 1 at the highest block) alternates with each block of distinct 0 or 1 digits.

```
parity(x,0).
parity(_>x,1).
parity(_>(X>Xs),P1):-parity(X>Xs,P0),P1 is 1-P0.

even_(_>Xs):-parity(Xs,1).
odd_(_>Xs):-parity(Xs,0).
```

We will now specify successor and predecessor through two mutually recursive predicates, s and p.

They first decompose their arguments as if using decons. Then, after transforming them as a result of adding 1, they place back the results as if using the cons operation, both emulated by the use of the constructor "**>**". Note that the two functions work on trees with steps corresponding to *a block of* 0 *or* 1 *digits at a time*. They are based on arithmetic observations about the behavior of these blocks when incrementing or decrementing a binary number by 1.

```
s(x,x>x).
s(X>x,X>(x>x)):-!.
s(X>Xs,Z):-parity(X>Xs,P),s1(P,X,Xs,Z).
```

After computing parity, the successor predicate s delegates the transformation of the blocks of 0 and 1 digits to predicate s1 handling both the even_ and odd_ cases.

```
s1(0,x,X>Xs,SX>Xs):-s(X,SX).
s1(0,X>Ys,Xs,x>(PX>Xs)):-p(X>Ys,PX).
s1(1,X,x>(Y>Xs),X>(SY>Xs)):-s(Y,SY).
s1(1,X,Y>Xs,X>(x>(PY>Xs))):-p(Y,PY).
```

The predecessor function p inverts the work of s

```
p(x>x,x).
p(X>(x>x),X>x):-!.
p(X>Xs,Z):-parity(X>Xs,P),p1(P,X,Xs,Z).
```

After computing parity, the predecessor predicate p delegates the transformation of the blocks of 0 and 1 digits to p1 handling separately the even_ and odd_ cases.

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

PROPOSITION 8. *Assuming parity information is kept explicitly, the operations* s *and* p *work on a binary tree of size N in time constant on average and and $O(log^*(N))$ in the worst case*

Proof See [26].

PROPOSITION 9. *The operations* s *and* p *implement successor and predecessor operations such that their results correspond to the same operations on natural numbers,i.e., the following hold.*

$$t(A,X), s(X,Y), B \; is \; A+1, n(Y,C) \rightarrow B = C \qquad (7)$$

$$t(A,X), p(X,Y), B \; is \; A-1, n(Y,C) \rightarrow B = C \qquad (8)$$

Proof See [26].

EXAMPLE 20. s *and* p *implement arithmetic correctly.*

```
?- A=10,t(A,X),s(X,Y),B is A+1,n(Y,C).
A = 10,X = (x> (x> (x> (x>x)))),Y = ((x>x)> (x> (x>x))),
B = C, C = 11 .
```

Our binary trees can be seen as a model of *Peano Arithmetic*, in the same sense as unary or binary arithmetic. Note also, that while any enumeration would provide unary arithmetic, our representation implements the equivalent (or better) of *binary arithmetic*. We refer to [27] and [29] for the description of algorithms covering all the usual arithmetic operations with equivalent representations working on other members of the Catalan family and to [26] for a generic implementation using Haskell type classes. Hence our X-combinator trees can provide an implementation of arithmetic operations (including extension to integers and rational numbers). Moreover, they can also become the target of ranking and unranking functions that associate unique natural number codes to various combinatorial objects. In section 6 they will play this role for general lambda terms.

We refer to [26] for the development of a complete arithmetic system for the Catalan family of combinatorial objects, of which binary trees are the most well known instance.

6. A size-proportionate Gödel numbering bijection for lambda terms

We are finally ready to define our simple, linear time, size-proportionate bijection between tree-represented natural numbers and general lambda terms in de Bruijn notation.

6.1 Ranking and unranking de Bruijn terms to binary-tree represented natural numbers

The predicate `rank` defines a bijection from lambda expressions in de Bruijn notation to binary trees, seen here as implementing natural numbers. Variables `v/1` are represented as trees with the left `x` as their left branch, lambdas `l/1` as trees with `x` as their right branch. To avoid ambiguity, ranks for application nodes will be incremented by one using the successor predicate `s/2`.

```
rank(v(0),x).
rank(l(A),x>T):-rank(A,T).
rank(v(K),T>x):-K>0,t(K,T).
rank(a(A,B),X1>Y1):-rank(A,X),s(X,X1),rank(B,Y),s(Y,Y1).
```

The predicate `unrank` defines the inverse bijection from binary trees, seen as natural numbers, to lambda expressions in de Bruijn notation. It works by case analysis on trees with branches marked with `x` and decrements branches using predicate `p/2` to ensure it inverts the action of `rank` on application nodes. Note also that both predicates use the bijections `t` and respective `n` to convert between tree-based naturals and their standard natural number equivalents.

```
unrank(x,v(0)).
unrank(x>T,l(A)):-!,unrank(T,A).
unrank(T>x,v(N)):-!,n(T,N).
unrank(X>Y,a(A,B)):-
  p(X,X1),unrank(X1,A),
  p(Y,Y1),unrank(Y1,B).
```

PROPOSITION 10. *Assuming variable indices are small (word-size) integers,* `rank` *and* `unrank` *define a size-proportionate bijection between lambda terms in de Bruijn form and X-combinator trees. Their runtime is proportional to the size of their input.*

Proof If variable indices are fixed sized small integers, one can assume that `t` and `n` work in constant time. Then, observe that each step of both predicates works in time proportional to `s` or `p` for a total proportional to the number of internal nodes.

As an interesting variation, for very large terms, one could actually *use binary tree-based natural numbers for the indices of* `v/1` *in de Bruijn terms*, and completely bypass the use of `t` and `n`, and thus lifting the assumption about variable indices being fixed size integers.

EXAMPLE 21. *Ranking and unranking of K and S combinators in de Bruijn form.*

```
?- kB(K),rank(K,B),unrank(B,K1).
K = K1, K1 = l(l(v(1))),
B = (x> (x> ((x>x)>x))) .

?- sB(S),rank(S,B),unrank(B,S1).
S = S1, S1 = l(l(l(a(a(v(2), v(0)), a(v(1), v(0))))))),
B = (x> (x> (x> ((x> (((x> (x>x))>x)>
    (x>x)))> (x> (((x>x)>x)> (x>x)))))))) .
```

7. Playing with the playground

The following quote from Donald Knuth, in answering a question of Frank Ruskey about the short term economics behind research (`http://www.informit.com/articles/article.aspx?p=2213858`) and prominently displayed at Mayer Goldberg's home page at `http://www.little-lisper.org/website/`, summarizes our motivation behind building this declarative playground:

> Everybody seems to understand that astronomers do astronomy because astronomy is interesting. Why don't they understand that I do computer science because computer science is interesting?

This being said, we will sketch here a few use cases, some of possible practical significance.

7.1 Self-typed terms

As X-combinator trees and their types share the same representation, it makes sense to generate and count terms that are equal to their types. The predicate `genSelfTypedT` generates such "self-typed" terms.

```
genSelfTypedT(L,T):-genTree(L,T),xtype(T,T).
```

EXAMPLE 22. *Self-typed X-combinator trees of size 6.*

```
?- genSelfTypedT(6,T).
T = (x> ((x>x)> ((x>x)> (x>x)))) ;
T = (x> (((x> (x>x))> (x>x))>x)) ;
T = ((x>x)> ((x> (x>x))> (x>x))) ;
T = ((x>x)> (((x>x)>x)> (x>x))).
```

The sequence [0, 0, 0, 1, 2, 4, 14, 34, 101, 315, 1017, 3325, 11042] counts the number of self-typed terms by increasing sizes, up to size 13.

7.2 Two size-inflating injective functions from terms to terms

By composing transformations of X-combinator trees to their equivalent lambda expressions two interesting (but injective only) mappings can be defined from X-combinator trees to a subset of them (`t2t`) and from lambda terms to a subset of them (`b2b`).

```
b2b --> rank,t2b.
t2t --> t2b,rank.
```

EXAMPLE 23. *The injective mappings* `t2t` *and* `b2b` *can be used to generate significantly larger X-combinator trees and lambda expressions.*

```
?- between(0,3,N),t(N,T),t2t(T,NewT),tsize(T,S1),
   tsize(NewT,S2),write(S1<S2),write(' '),fail;nl.
0<27 1<57 2<86 2<86

?- skkB(B),bsize(B,S1),b2b(B,BB),bsize(BB,S2),
          write(S1<S2),nl,fail.
12<374
```

It is interesting at this point to see what happens to our building block – the X-combinator – when going through some of these transformations.

EXAMPLE 24. *Transformations of the X-combinator via* `b2b`, `evalB`, `btype` *and* `n`.

```
?- xB(X),b2b(X,XX),evalB(XX,R),btype(R,T),n(T,N).
X = l(a(a(a(...(l(v(1))))),
XX = a(l(a(a(a....l(l(v(1))))))))),
R = l(l(l(l(a(a(a(v(3), v(2)), v(0)), a(v(1), v(0))))))),
T = ((x> (x> (x>x)))> (x> ((x>x)> (x>x)))) .
N = 576
```

While `b2b` significantly inflates the de Bruijn term corresponding to the X-combinator, normalization reduces it to a small, well-typed term. This suggests the use of our shared representation for experiments with dynamic systems or genetic programming where applications of arithmetic, type inference and normalization operations are likely to create interesting trajectories of evolution.

7.3 Evolution of a multi-operation dynamic system

Normalization, as the lambda calculus is is Turing-complete, is subject to non-termination. However, simply-typed terms are strongly normalizing so it makes sense to play with combinations of arithmetic operations, type inference operations and normalization involving X-term combinator trees as well as their lambda term equivalents.

For instance, the predicate `evalOrNextB` ensures that evaluation only proceeds on lambda terms for which we are sure it terminates with a new term and applies the successor predicate "`s`" otherwise, borrowed via the `rank` and `unrank` operations.

```
evalOrNextB(B,EvB):-btype(B,_),evalB(B,EvB),EvB\==B,!.
evalOrNextB(B,NextB):-
  rank(B,T),
  s(T,NextT),
  unrank(NextT,NextB).
```

We can observe the orbits of these dynamic systems [15] starting from a given lambda term in de Bruijn notation, for a given number of steps with the predicate `playWithB`.

```
playWithB(Term,Steps,Orbit):-
  playWithB(Term,Steps,Orbit,[]).

playWithB(Term,Steps,[NewTerm|Ts1],Ts2):-Steps>0,!,
  Steps1 is Steps-1,
  evalOrNextB(Term,NewTerm),
  playWithB(NewTerm,Steps1,Ts1,Ts2).
playWithB(Term,_,[Term|Ts],Ts).
```

Note that ranking these terms to usual bitstring-represented integers would be intractable given their super-exponential growth with depth. On the other hand, all the underlying operations are linear time with ranking and unranking to natural numbers represented as binary trees. These terms are rather large, but by computing the sizes of the terms one can have a good guess on their evolution.

Figure 3 illustrates the evolution of this dynamic system starting from the X-combinator's lambda equivalent by plotting the tree sizes of the terms in its orbit. The plot indicates that it is very likely that a repetitive pattern has developed.

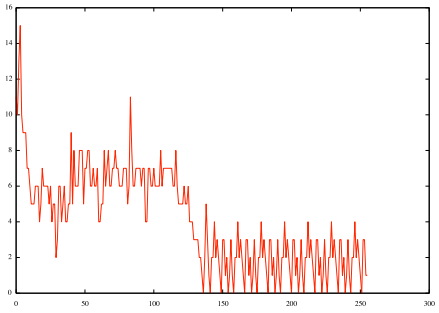

Figure 3. Term sizes in the orbit starting from the X-combinator

Figure 4 illustrates the evolution of this dynamic system starting from the term $\omega = SII(SII)$ by plotting the tree sizes of the terms in its orbit. The plot indicates that it is very unlikely that a repetitive pattern will develop.

Besides theoretical curiosity, one might use such operations for implementing genetic programming algorithms.

7.4 Memory savings through shared representations

Given that the ranking and unranking operations work in time proportional to the size of our lambda terms, we will explore some of the memory management consequences of a shareable

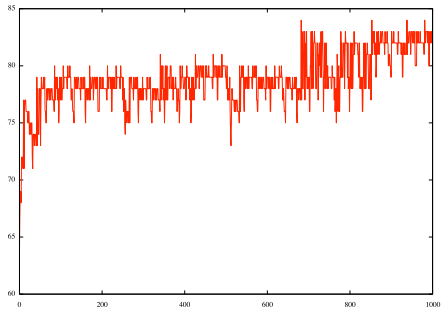

Figure 4. Term sizes in the orbit starting from the term ω

representation of combinators, simple types, natural numbers and lambda expressions.

We will look first at a well-known isomorphism that brings us a significantly more compact memory representation.

7.4.1 A succinct representation of binary trees

Binary trees are in a well-known bijection with the language of of balanced parentheses, both being a member of the Catalan family of combinatorial objects [23]. The reversible predicate `t2p/2` transforms between binary trees and lists of balanced parentheses.

```
t2p(T,Ps):-t2p(T,0,1,Ps,[]).

t2p(X,L,R) --> [L],t2ps(X,L,R).

t2ps(x,_,R) --> [R].
t2ps((X>Xs),L,R) --> t2p(X,L,R),t2ps(Xs,L,R).
```

EXAMPLE 25. *The work of the reversible predicate* `t2p/2`.

```
?- skkT(X),t2p(X,Ps),t2p(NewX,Ps).
X = NewX, NewX = (((x> (x>x))> ((x>x)>x))> ((x>x)>x)),
Ps = [0,0,0,0,1,0,1,1,0,0,1,1,1,0,0,1,1,1].

?- kB(B),rank(B,T),t2p(T,Ps).
B = l(l(v(1))),
T = (x> (x> ((x>x)>x))),
Ps = [0,0,1,0,1,0,0,1,1,1] .
```

Seen as a bitstring, the mapping to a list of balanced parentheses is a succinct representation for our binary trees, if one wants to trade time complexity for space complexity. It is also a self-delimiting prefix-free representation, uniquely decodable when read from left to right. As one might notice, it is actually is a bifix code, i.e., it is also prefix-free when read from right to left.

7.4.2 A practical shared memory representation

In a practical implementation, given the high frequency of small objects of any of our kinds – numbers, lambda expressions, types and combinators, one might consider a hybrid representation where small trees are represented within a machine word as balanced 0,1-parentheses sequences and larger ones as cons-cells. 2-bit-tagged pointers could be used to disambiguate interpretation as numbers, combinators types or lambda expressions but their targets could be shared if structurally identical. Besides sharing static data or code objects, a shared representation is likely to also facilitate memory management by recycling fragments of computations like β-reductions or arithmetic operations.

Graph-based representation of lambda terms has been used as early as [18] to avoid redundant evaluation of redexes. In a similar way, one could fold our tree-based representations into DAGs, providing uniform savings for combinators, types and tree-based natural numbers.

8. Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [2]. One-point combinator bases, together with a derivation of the X-combinator are described in [7]. In [9] the existence of a countable number of 1-point bases is proven. While esoteric programming languages exist based on similar 1-point bases [24], we have not seen any such development centered around Rosser's X-combinator, or type inference and normalization algorithms designed specifically for it, as described in this paper.

Originally introduced in [6], the de Bruijn notation makes terms equivalent up to α-conversion and facilitates their normalization [14]. Their use in this paper is motivated by their comparative simplicity rather than by efficiency considerations, for which several abstract machines, used in the implementation of functional languages, have been designed [20].

Combinatorics of lambda terms, including enumeration, random generation and asymptotic behavior has seen an increased interest recently (see for instance [4, 5, 10, 11]), partly motivated by applications to software testing, given the widespread use of lambda terms as an intermediate language in compilers for functional languages and proof assistants. In [19], types are used to generate random terms for software testing. The same naturally goal-oriented effect is obtained in the generator/type inferrer for de Bruijn terms in subsection 3.6, by taking advantage simply of Prolog's ability to backtrack over possible terms while filtering against unification with a specific pattern. In fact one can see the the "query-by-type" algorithm in subsection 3.6 as similar in purpose, via the Curry-Howard correspondence, to algorithms for finding inhabitants of intuitionistic propositional tautologies like [3].

Of particular interest are the results of [10] where recurrence relations and asymptotic behavior are studied for several families of lambda terms. Empirical evaluation of the density of closed simply-typed general lambda terms described in [10] indicates extreme sparsity for large sizes. However, the problem of their exact asymptotic behavior is still open. This has motivated our interest in the empirical evaluation of the density of simply-typed X-combinator trees, where we observed significantly higher initial densities and where there's a chance that the also open problem of their asymptotic behavior might be easier to tackle.

Ranking and unranking algorithms for several classes of lambda terms are also described in [10],together with a type inference algorithm for de Bruijn terms. Ranking and unranking of lambda terms can be seen as a building block for bijective serialization of practical data types [33] as well as for Gödel numbering schemes [12] of theoretical relevance. In fact, ranking functions for sequences can be traced back to Gödel numberings [8] associated to formulas.

Injective Gödel numbering schemes for lambda terms in de Bruijn notation have been described in the context of binary lambda calculus [32] and as a mechanism to encode datatypes in [17, 33]. Both these use prefix-free codes, ensuring unique decoding. A bijective Gödel numbering scheme is associated to the esoteric programming language Jot [24], where every bitstring is considered a valid executable expression. This is similar to ours in the sense that every binary tree representing an X-combinator expression is executable. However, the use of a binary tree based model of Peano's axioms, playing the role of the set of natural numbers, and the corresponding ranking and unranking algorithms as described in this paper are novel.

The binary-tree based numbering system defined here is isomorphic to the ones in [27, 29], where a similar treatment of arithmetic operations is specialized to the language of balanced parentheses and multiway trees. In fact, such an encoding can be used as a prefix-free succinct representation for our binary trees, if one wants to trade space complexity for time complexity. Any enumeration of combinatorial objects (e.g., [16, 23]) can be seen as providing unary Peano arithmetic operations implicitly. By contrast, the tree-based arithmetic operations used in this paper have efficiency comparable to the usual binary numbers, as shown in [26]. Note also that while [27, 29] focus exclusively on arithmetic operations with members of the Catalan family of combinatorial objects, of which our binary trees are an instance, their use in this paper, as a target for ranking/unranking of lambda expressions, relies exclusively on the successor and predecessor operations, adapted here to work on binary trees.

Univalent foundations of type theory [31] have recently emphasized isomorphism paths between objects as a means to unify equality and equivalence between heterogenous data types sharing essential properties and behaviors under transformations. While informal, our executable equivalences between combinators, lambda terms, types and numbers might be useful as practical illustrations of these concepts.

Some of the algorithms used in the paper, like type inference and normalization of combinators and lambda terms, are common knowledge [1, 14, 21], although we are not aware, for instance, of Prolog implementations of type inference working directly on de Bruijn terms or X-combinator trees. In [30] a type inference algorithm for standard terms using Prolog's logic variables is given. To make the paper self-contained, we have closely followed the normalization algorithm of [30] using a de Bruijn representation of lambda terms. We refer to [30] for a compressed de Bruijn representation and several Prolog algorithms that complement our playground with generators for closed, linear, linear affine, binary lambda terms as well as lambda terms of bounded binary height.

9. Conclusion and future work

By sharing the representation of the Turing-complete language of X-combinator expressions, natural numbers, lambda terms and their types, interesting synergies become available.

While the main focus of the paper is the creation of a logic programming based declarative playground for experiments with various classes of lambda terms, under the assumption of a shared representation, the paper introduces several new concepts among which we mention:

- X-combinator trees playing the role of both natural numbers and types in subsections 2.1, 3.2 and 5.1
- a bijection between natural numbers and binary trees (predicates `t/2` and `n/2` in subsection 5.1) that is works consistently with their isomorphic arithmetic operations
- a concept of "iterated types" in subsection 3.9
- two size-inflating injective functions from terms to terms in subsection 7.2
- a multi-operation dynamic system combining normalization and arithmetic operations in subsection 7.3

The paper also describes algorithms that, at our best knowledge, are novel, at least in terms of their logic programming implementation:

- direct type inference for X-combinator trees in subsection 3.3
- integrated generation and type inference algorithm for closed simply-typed de Bruijn terms in subsection 3.5
- successor and predecessor and arithmetic operations on binary trees in subsection 5.2
- ranking and unranking de Bruijn terms to/from binary-tree represented natural numbers in subsection 6.1

Future work is planned along the following lines. Enumeration or random generation of binary trees can be extended to general lambda expressions and various data types expressed in terms of them. Functional languages like Scheme and Lisp, based on `cons` operations might be able to improve memory footprint of symbolic and numerical data through shared representations of arithmetic operations and list or tree data structures. Small steps in the normalization of combinator expressions or lambda trees can be mapped to possibly interesting number sequences. Open problems related to the asymptotic density of typable combinators and lambda terms might benefit from empirical estimates computable within our framework for very large terms.

In combination with unification with occurs-check, the backtracking mechanism in logic-based languages like Prolog automates combinatorial generation, by contrast to the need to write wrapper code in a functional or procedural language. We hope that the techniques described in this paper, taking advantage of this unique combination of strengths, recommend logic programming as a convenient meta-language for the manipulation of various families of lambda terms and the study of their combinatorial and computational properties.

Acknowledgement

We thank the anonymous reviewers of PPDP'15 for their constructive criticism and valuable suggestions that have helped improving the paper. This research has been supported by NSF grant `1423324`.

References

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

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

[3] C.-B. Ben-Yelles. *Type assignment in the lambda-calculus: Syntax and semantics*. PhD thesis, University College of Swansea, 1979.

[4] O. Bodini, D. Gardy, and B. Gittenberger. Lambda-terms of bounded unary height. In *ANALCO*, pages 23–32. SIAM, 2011.

[5] 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.

[6] 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.

[7] J. Fokker. The systematic construction of a one-combinator basis for lambda-terms. *Formal Aspects of Computing*, 4:776–780, 1992.

[8] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[9] M. Goldberg. A construction of one-point bases in extended lambda calculi. *Inf. Process. Lett.*, 89(6):281–286, 2004.

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

[11] K. Grygiel, P. M. Idziak, and M. Zaionc. How big is BCI fragment of BCK logic. *J. Log. Comput.*, 23(3):673–691, 2013.

[12] J. Hartmanis and T. P. Baker. On Simple Goedel Numberings and Translations. In J. Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 301–316, Berlin Heidelberg, 1974. Springer. ISBN 3-540-06841-4.

[13] J. R. Hindley and J. P. Seldin. *Lambda-calculus and combinators: an introduction*, volume 13. Cambridge University Press Cambridge, 2008.

[14] F. Kamareddine. Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems. *Journal of Logic and Computation*, 11(3):363–394, 2001.

[15] A. Katok and B. Hasselblatt. *Introduction to the modern theory of dynamical systems*, volume 54 of *Ency. of Math. and its App.* Cambridge Univ. Press, 1995.

[16] 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. ISBN 0321335708.

[17] N. Kobayashi, K. Matsuda, and A. Shinohara. Functional Programs as Compressed Data. *ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, January 2012. ACM Press.

[18] J. Lamping. An Algorithm for Optimal Lambda Calculus Reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 16–30, 1990.

[19] 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.

[20] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., NJ, USA, 1987.

[21] P. Sestoft. Demonstrating lambda calculus reduction. In T. A. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[22] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. 2014. Published electronically at https://oeis.org/.

[23] R. P. Stanley. *Enumerative Combinatorics*. Wadsworth Publ. Co., Belmont, CA, USA, 1986. ISBN 0-534-06546-5.

[24] M. Stay. Very simple chaitin machines for concrete AIT. *CoRR*, abs/cs/0508056, 2005. URL `http://arxiv.org/abs/cs/0508056`.

[25] P. Tarau. Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings) . *Theory and Practice of Logic Programming*, 13(4-5):847–861, 2013.

[26] P. Tarau. A Generic Numbering System based on Catalan Families of Combinatorial Objects. *CoRR*, abs/1406.1796, 2014.

[27] P. Tarau. Computing with Catalan Families. In A.-H. Dediu, C. Martin-Vide, J.-L. Sierra, and B. Truthe, editors, *Proceedings of Language and Automata Theory and Applications, 8th International Conference, LATA 2014*, pages 564–576, Madrid, Spain,, Mar. 2014. Springer, LNCS.

[28] P. Tarau. Bijective Collection Encodings and Boolean Operations with Hereditarily Binary Natural Numbers. In *PPDP '14: Proceedings of the 16th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, New York, NY, USA, 2014. ACM.

[29] P. Tarau. Arithmetic and boolean operations on recursively run-length compressed natural numbers. *Scientific Annals of Computer Science*, 24(2):287–323, 2014. .

[30] 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.

[31] The Univalent Foundations Program. *Homotopy Type Theory*. Institute of Advanced Studies, Princeton, 2013. `http://homotopytypetheory.org/2013/06/20/the-hott-book/`.

[32] J. Tromp. Binary lambda calculus and combinatory logic, 2014. URL `https://tromp.github.io/cl/LC.pdf`.

[33] D. Vytiniotis and A. Kennedy. Functional Pearl: Every Bit Counts. *ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming*, September 2010. ACM Press.