A Logic Programming Playground for Lambda Terms, Combinators, Types and Tree-based Arithmetic Computations

PAUL TARAU
Department of Computer Science and Engineering
(*e-mail:* `paul.tarau@unt.edu`)

submitted ?; revised ?; accepted ?

Abstract

With sound unification, Definite Clause Grammars 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.

Compact combinatorial generation algorithms are given for several families of lambda terms, including open, closed, simply typed and linear terms as well as type inference and normal order reduction algorithms. We describe a Prolog-based combined lambda term generator and type-inferrer for closed well-typed terms of a given size, in de Bruijn notation.

We introduce a compressed de Bruijn representation of lambda terms and define its bijections to standard representations. Our compressed terms facilitate derivation of size-proportionate ranking and unranking algorithms of lambda terms and their inferred simple types.

By taking advantage of Prolog's unique bidirectional execution model and sound unification algorithm, our generator can build "customized" closed terms of a given type. This relational view of terms and their types enables the discovery of interesting patterns about frequently used type expressions occurring in well-typed functional programs. Our study uncovers the most "popular" types that govern function applications among a about a million small-sized lambda terms and hints toward practical uses to combinatorial software testing.

The S and K combinator expressions form a well-known Turing-complete subset of the lambda calculus. We specify evaluation, type inference and combinatorial generation algorithms for SK-combinator trees. In the process, we unravel properties shedding new light on interesting aspects of their structure and distribution. We study the proportion of well-typed terms among size-limited SK-expressions as well as the type-directed generation of terms of sizes smaller then the size of their simple types. We also introduce the *well-typed frontier of an untypable term* and we use it to design a simplification algorithm for untypable terms taking advantage of the fact that well-typed terms are normalizable.

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. Using this shared representation, ranking/unranking algorithm of lambda terms to tree-based natural numbers are described.

Our algorithms, expressed as an incrementally developed literate Prolog program, implement a declarative playground for exploration of representations, encodings and computations with uniformly represented lambda terms, types, combinators and tree-based arithmetic.

KEYWORDS: lambda calculus, de Bruijn notation, generation of lambda terms, type inference, combinatorics of lambda terms, ranking and unranking of lambda terms, normalization of de Bruijn terms, SK-combinator calculus, generation of well-typed combinator expressions Rosser's X-combinator, tree-based numbering systems, bijective Gödel-numberings, logic programming as meta-language.

1 Introduction

This paper is an extended synthesis of (Tarau 2015b; Tarau 2015c; Tarau 2015a). It is organized as literate Prolog program and provides a comprehensive playground for both classic algorithms working on lambda terms and combinators as well as a large number of new algorithms and data representations covering their combinatorial generation, type inference and their ranking/unranking to both standard and tree-represented natural numbers.

Logic programming provides a convenient metalanguage for modeling data types and computations 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 generating terms for various calculi. This holds in particular for lambda terms and combinators (Barendregt 1984).

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. Lambda terms provide a foundation to modern functional languages, type theory and proof assistants and, as a sign of their lasting relevance, they have been lately incorporated into mainstream programming languages including Java 8, C# and Apple's newly designed programming language Swift.

Generation of lambda terms has practical applications to testing compilers that rely on lambda calculus as an intermediate language, as well as to the generation of random tests for user-level programs and data types. At the same time, several instances of lambda calculus are of significant theoretical interest given their correspondence with logic and proofs.

The use of modern Prologs' unification with occurs-check, backtracking mechanisms and Definite Clause Grammars (DCGs) are instrumental in designing compact algorithms for inferring simple types or for generating linear, linear affine or lambda terms with bounded unary height as well as in implementing normalization algorithms.

Of particular interest are representations that are canonical up to alpha-conversion (variable renamings), among which the most well-known ones are de Bruijn's indices (de Bruijn 1972), representing bound variables as the number of binders to traverse to the lambda abstraction binding them. As a sequence of binders, in de Bruijn notation, can be seen as a natural number expressed in unary notation, we introduce a compressed representation of the binders that puts in a new light the underlying combinatorial structure of lambda terms and highlights their connection to the *Catalan family* of combinatorial objects (Stanley 1986), among which binary trees are the most well known. The proposed compressed de Bruijn notation also simplifies generation of some families of lambda terms.

A joke about the de Bruijn indices representation of lambda terms is that it can be used to tell apart Cylons from humans (McBride 2010). Arguably, the compressed de Bruijn representation that we introduce in here is taking their fictional use one step further. To alleviate the legitimate fears of our (most likely, for now, human) reader, these representations will be mapped bijectively to conventional ones. To be able to use the most natural representation for each of the proposed algorithms, we implement bijective transformations between lambda terms in standard as well as de Bruijn and compressed de Bruijn representation.

A merit of our compressed representation is to simplify the underlying combinatorial structure of lambda terms, by exploiting their connection to the Catalan family of combinatorial objects (Stanley 1986). This leads to algorithms that focus on their (bijective) natural number encodings - known to combinatorialists as *ranking/unranking* functions (Kreher and Stinson 1999) and to logicians as *Gödel-numberings* (Gödel 1931). Among the most obvious practical applications, such encodings can be used to generate random terms for testing tools like compilers or source-to-source program transformers. At the same time, as our encodings are "size-proportionate", they provide a compact serialization mechanism for lambda terms.

To derive a bijection to $\mathbb{N}$ (seen as made of conventional bitstring-represented numbers), that is size-proportionate, we will first extract a "Catalan skeleton" abstracting away the recursive structure of the compressed de Bruijn term, then implement a bijection from it to $\mathbb{N}$. The "content" fleshing out the term, represented as a list of natural numbers, will have its own bijection to $\mathbb{N}$ by using a generalized Cantor tupling / untupling function, that will also help pairing / unpairing the code of the skeleton and the code of the content of the term.

Combinators are closed lambda terms without bound variables. They actually predate lambda calculus , being discovered in the 1920s by Schönfinkel and then independently by Curry. With *function application* as their unique operation, and a convenient *base*, (for instance, $K = \lambda x.\ \lambda y.\ x$ and $S = \lambda f.\ \lambda g.\ \lambda x.(f\ x)\ (g\ x)$), they form a Turing-complete subset of the lambda calculus. We will focus, using some essential Prolog ingredients, on synergies between generation and type inference on the language of S and K combinators. SK-combinator expressions are binary trees with leaves labeled S or K and internal nodes representing function application. While working with a drastically simplified model of computation, interesting patterns emerge, some of which may extend to the richer combinator languages used in compilers for functional languages like Haskell and ML and proof assistants like Coq and Agda.

Of particular interest are type inference algorithms and the possibility to melt them together with generators of terms of a limited size. Evaluation of SK-combinator expressions has the nice property of always terminating on terms that have simple types. This suggests looking into how this property (called *strong normalization*) can be used to simplify untypable (and possibly not normalizable) terms through normalization of their maximal typable subterms. At the same time, this suggests trying to discover some empirical hints about the distribution of well-typed terms and the structure induced by typable subterms of untypable terms.

We also 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 (Tarau 2014a; Tarau 2014d; Tarau 2014c; Tarau 2013) 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 will take advantage of the fact that Rosser's X-combinator expressions (Fokker 1992) (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 time having tree-based natural numbers as targets.

Ranking and unranking of lambda terms (i.e., their bijective mapping to unique natural number codes) has practical applications to testing compilers that rely on lambda calculus as an intermediate language, as well as in generation of random tests for user-level programs and data types. At the same time, several instances of lambda calculus are of significant theoretical interest given their correspondence with logic and proofs. 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. Prolog's ability to support "relational" queries enables us to easily explore the population of de Bruijn terms up to a given size and answer questions like the following:

1. How many distinct types occur for terms up to a given size?
2. What are the most popular types?
3. What are the terms that share a given type?
4. What is the smallest term that has a given type?
5. What smaller terms have the same type as this term?

The remaining of the paper (with the content of the most salient subsections also pointed out) is organized as follows.

Section 2 introduces the compressed de Bruijn terms (2.5) and bijective transformations from them to standard lambda terms.

Section 3 describes a type inference algorithms for lambda terms.

Section 4 describes generators for several classes of lambda terms, including closed, simply typed, linear, affine as well as terms with bounded unary height and terms in the binary lambda calculus encoding. Subsection 4.4 introduces a generator for lambda terms in de Bruijn form. Subsection 4.13 introduces an algorithm combining term generation and type inference.

Section 5 describes a normal order reduction algorithm for lambda terms relaying on their de Bruijn representation.

Section 6 describes combinators with emphasis on the SK and X combinator bases. SK-combinator trees together with a generator and an evaluation algorithm. Subsection 6.2 defines simple types for SK-combinator expressions and describes a type inference algorithm on for SK-combinator trees. Subsection 6.3 introduces X-combinator trees together with a generator and an evaluation algorithm. Subsection 6.5 defines simple types for X-combinator expressions via their equivalent lambda terms, describes type inference algorithms for X-combinator expressions. It also explores consequences of expressions and types sharing the same binary tree representation.

Section 7 describes size-proportionate bijective encodings of lambda terms and combinators. Subsection 7.1 builds mappings from lambda terms to Catalan families of combinatorial objects, with focus on binary trees representing their inferred types and their applicative skeletons. These mappings lead in subsection 7.2 to size-proportionate ranking and unranking algorithms for lambda terms and their inferred types. Subsection 7.3 interprets X-combinator trees as natural numbers on which it defines arithmetic operations. Subsection 7.4 describes a bijection from lambda terms to binary trees implementing tree-based arithmetic operations that leads to a different mechanism for size-proportionate ranking and unranking algorithms for lambda terms.

Section 8 shows examples of applications of our declarative playground. Subsection 8.3 uses our combined term generation and type inference algorithm to discover frequently occurring type patterns. Subsection 8.4 describes a type-directed algorithm for the generation of closed typable lambda terms. We also explore consequences that emerge from interactions between such heterogeneous computational objects sharing the same binary tree representation. Subsection 8.6 introduces the well-typed frontier of an untypable SK-expression and describes a partial normalization-based simplification algorithm that terminates on all SK-expressions.

Section 9 discusses related work and section 10 concludes the paper.

The paper is structured as a literate Prolog program. The code has been tested with SWI-Prolog 6.6.6 and YAP 6.3.4.

It is available `http://www.cse.unt.edu/~tarau/research/2015/play.pro`.

2 Morphing between representations of lambda terms

Logic variables can be used in Prolog for connecting a lambda binder and its related variable occurrences. This representation can be made canonical by ensuring that each lambda binder is marked with a distinct logic variable. For instance, the lambda term $\lambda a.((\lambda b.(a(b\ b)))(\lambda c.(a(c\ c))))$ will be represented as `l(A,a(l(B, a(A,a(B,B))), l(C, a(A,a(C,C)))))`. It is however convenient in most algorithms to avoid any confusion between variables in our meta-language (Prolog) and lambda variables. We will achieve this by using the *de Bruijn representation of lambda terms*.

2.1 De Bruijn Indices

De Bruijn indices (de Bruijn 1972) provide a *name-free* representation of lambda terms. All closed 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, the term `l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))` is represented as `l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0))))))`, given 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`.

We will also define the size of a lambda expression in de Bruijn form as the number of its internal nodes, implemented by the predicate `dbTermSize`.

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

2.2 Open and closed terms

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

```
isClosedB(T):-isClosed1B(T,0).

isClosed1B(v(N),D):-N<D.
isClosed1B(l(A),D):-D1 is D+1,
  isClosed1B(A,D1).
isClosed1B(a(X,Y),D):-
  isClosed1B(X,D),
  isClosed1B(Y,D).
```

Besides being closed, lambda terms interesting for functional languages and proof assistants, are also well-typed. We will start with an algorithm inferring types directly on the de Bruijn terms.

2.3 From de Bruijn to lambda terms with canonical names

The predicate `b2l` converts from the de Bruijn representation to lambda terms whose canonical names are provided by logic variables. We will call them terms in *standard notation*.

```
b2l(A,T):-b2l(A,T,_Vs).

b2l(v(I),V,Vs):-nth0(I,Vs,V).
b2l(a(A,B),a(X,Y),Vs):-b2l(A,X,Vs),b2l(B,Y,Vs).
b2l(l(A),l(V,Y),Vs):-b2l(A,Y,[V|Vs]).
```

Note the use of the built-in `nth0/3` that associates to an index `I` a variable `V` on the list `Vs`. As we initialize in `b2l/2` the list of logic variables as a free variable `_Vs`, free variables in open terms, represented with indices larger than the number of available binders will also be consistently mapped to logic variables. By replacing `_Vs` with `[]` in the definition pf `b2l/2` one could enforce that only closed terms (having no free variables) are accepted.

Example 1
illustrates the bijection defined by predicates `l2b` and `b2l`.

```
?- LT=l(A,l(B,l(C,a(a(A,C),a(B,C))))),l2b(LT,BT),b2l(BT,LT1),LT=LT1.
LT = LT1, LT1 = l(A, l(B, l(C, a(a(A, C), a(B, C))))),
BT = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))).
```

2.4 From lambda terms with canonical names to de Bruijn terms

Logic variables provide canonical names for lambda variables. An easy way to manipulate them at meta-language level is to turn them into special "$VAR/1" terms - a mechanism provided by Prolog's built-in `numbervars/3` predicate. Given that "$VAR/1" is distinct from the constructors lambda terms are built from (`l/2 and a/2`) this is a safe (and invertible) transformation. To avoid any side effect on the original term, we will uniformly rename its variables to fresh ones with Prolog's `copy_term/2` built-in. We will adopt this technique through the paper each time our operations would mutate an input argument otherwise.

```
l2b(A,T):-copy_term(A,CA),numbervars(CA,0,_),l2b(CA,T,_Vs).

l2b('$VAR'(V),v(I),Vs):-once(nth0(I,Vs,'$VAR'(V))).
l2b(a(X,Y),a(A,B),Vs):-l2b(X,A,Vs),l2b(Y,B,Vs).
l2b(l(V,Y),l(A),Vs):-l2b(Y,A,[V|Vs]).
```

2.5 A compressed de Bruijn representation of lambda terms

.

As a step further, we will not look into compressing blocks of lambdas. Iterated lambdas (represented as a block of constructors `l/1` in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them. So it makes sense to represent that number more efficiently in the usual binary notation. Note that in de Bruijn notation blocks of lambdas can wrap either applications or variable occurrences represented as indices. This suggests using just two constructors: `v/2` indicating in a term `v(K,N)` that we have `K` lambdas wrapped around the de Bruijn index `v(N)` or `a/3` indicating in a term `a(K,X,Y)` that `K` lambdas are wrapped around the application `a(X,Y)`.

We call the terms built this way with the constructors `v/2` and `a/3` *compressed de Bruijn terms*.

2.6 From de Bruijn to compressed

We can make precise the definition of compressed deBruijn terms by providing a bijective transformation between them and the usual de Bruijn terms.

The predicate `b2c` converts from the usual de Bruijn representation to the compressed one. It proceeds by case analysis on `v/1, a/2, l/1` and counts the binders `l/1` as it descends toward the leaves of the tree. Its steps are controlled by the predicate `up/2` that increments the counts when crossing a binder.

```
b2c(v(X),v(0,X)).
b2c(a(X,Y),a(0,A,B)):-b2c(X,A),b2c(Y,B).
b2c(l(X),R):-b2c1(0,X,R).

b2c1(K,a(X,Y),a(K1,A,B)):-up(K,K1),b2c(X,A),b2c(Y,B).
b2c1(K, v(X),v(K1,X)):-up(K,K1).
b2c1(K,l(X),R):-up(K,K1),b2c1(K1,X,R).

up(From,To):-From>=0,To is From+1.
```

2.6.1 From compressed to de Bruijn

The predicate `c2b` converts from the compressed to the usual de Bruijn representation. It reverses the effect of `b2c` by expanding the `K` in `v(K,N)` and `a(K,X,Y)` into `K+1` `l/1` binders (as counts start at `0`). The predicate `iterLam` performs this operation in both cases, and the predicate `down/2` computes the decrements at each step. We will reuse the predicates `up/2` and `down/2` that can be seen as abstracting away the successor/predecessor operation.

```
c2b(v(K,X),R):-X>=0,iterLam(K,v(X),R).
c2b(a(K,X,Y),R):-c2b(X,A),c2b(Y,B),iterLam(K,a(A,B),R).

iterLam(0,X,X).
iterLam(K,X,l(R)):-down(K,K1),iterLam(K1,X,R).

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

Example 2
illustrates the bijection defined by the predicates `b2c` and `c2b`.

```
?- BT=l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),b2c(BT,CT),c2b(CT,BT1).
BT = BT1, BT1 = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),
CT = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))) .
```

A convenient way to simplify defining chains of such conversions is by using Prolog's DCG transformation. For instance, the predicate `c2l/2` converts from compressed de Bruijn terms and standard lambda terms using de Bruijn terms as an intermediate step, while `l2c/2` works the other way around.

```
c2l --> c2b,b2l.
l2c --> l2b,b2c.
```

Closed terms can be easily identified by ensuring that the lambda binders on a given path from root outnumber the de Bruijn index of a variable occurrence ending the path. The predicate `isClosedC` does that for compressed de Bruijn terms.

```
isClosedC(T):-isClosedC(T,0).

isClosedC(v(K,N),S):-N<S+K.
isClosedC(a(K,X,Y),S1):-S2 is S1+K,isClosedC(X,S2),isClosedC(Y,S2).
```

3 Inferring simple types for lambda terms

3.1 Type Inference on standard terms with logic variables

Simple types, represented as binary trees built with the constructor “`>/2`” with empty leaves representing the unique primitive type “x”, can be seen as a “Catalan approximation” of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization).

While in a functional language inferring types requires implementing unification with occur check, as shown for instance in (Grygiel and Lescanne 2013), this operation is available in Prolog as a built-in. Also a “post-mortem” verification that unification has not introduced any cycles is provided by the built-in `acyclic_term/1`.

The predicate `extractType/2` works by seeing each logical variable `X` as denoting its type. As logic variable bindings propagate between binders and occurrences, this ensures that types are consistently inferred.

```
extractType(X,TX):-var(X),!,TX=X. % this matches all variables
extractType(l(TX,A),(TX>TA)):-extractType(A,TA).
extractType(a(A,B),TY):-extractType(A,(TX>TY)),extractType(B,TX).
```

Instead of (inefficiently) using unification with occurs-check at each step, we ensure that at the end, our term is still *acyclic*, with the built-in ISO-Prolog predicate `acyclic_term/1`.

```
polyTypeOf(LTerm,Type):-
  extractType(LTerm,Type),
  acyclic_term(LTerm).
```

At this point, most general types are inferred by `extractType` as fresh variables, similar to polymorphic types in functional languages, if one interprets logic variables as universally quantified. Such variables stand for any type expression, as schemata in the case of propositional or predicate logic axioms.

Example 3
Type inference for canonically represented standard terms. Note the need to use `copy_term` to avoid binding the object-level variables.

```
?- polyTypeOf(l(X,a(X,l(Y,Y))),T).
X = ((A>A)>B),
Y = A,
T = (((A>A)>B)>B).

?- copy_term(l(X,a(X,l(Y,Y))),LT),polyTypeOf(LT,T).
LT = l((A>A)>B, a((A>A)>B, l(A, A))),
T = (((A>A)>B)>B).
```

However, as we are only interested in simple types, we will bind uniformly the leaves of our type tree to the constant “x” representing our only primitive type, by using the predicate `bindTypeB/1`.

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

The simple type of a compressed de Bruijn term is then defined as:

```
hasType(CTerm,Type):-
  c2l(CTerm,LTerm),
  polyTypeOf(LTerm,Type),
  bindTypeB(Type).
```

We can also define the predicate `typable/1` that checks if a lambda term is well typed, by trying to infer and then ignoring its inferred type.

```
typable(Term):-hasType(Term,_Type).
```

Example 4
illustrates typability of the term corresponding to the `S` combinator $\lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\ x_2)\ (x_1\ x_2))$
and untypabilty of the term corresponding to the `Y` combinator $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$,
in de Bruijn form.

```
?- hasType(a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),T).
T = ((x> (x>x))> ((x>x)> (x>x))) .
?- hasType(
   a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),T).
false.
```

3.2 Type inference for lambda terms in de Bruijn notation

As lambda terms represent functions, inferring their types provides information on what kind of argument(s) they can be applied to. For simple types, type inference is decidable (Hindley and Seldin 2008) and it uses unification to recursively propagate type information between application sites of variable occurrences covered by a given lambda binder. We will describe next a type inference algorithm using de Bruijn indices in Prolog - a somewhat unusual choice, given that logic variables can play the role of lambda binders directly. One of the reasons we chose them is that they will be simpler to manipulate at meta-language level, as they handle object-level variables implicitly. At the same time this might be useful for other purposes, as we are not aware of any Prolog implementation of type inference with this representation of lambda terms.

Simple types will be defined here also as binary trees built with the constructor "`>/2`" with empty leaves, representing the unique primitive type "`x`". Types can be seen as as a "binary tree approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization), as it is well-known (e.g., (Barendregt 1991)) that lambda terms that have simple types are strongly normalizing. When a term `X` has a type `T` we say that the type `T` is *inhabited* by the term `X`.

While in a functional language inferring types requires implementing unification with occur check, as shown for instance in the appendix of (Grygiel and Lescanne 2013), this is readily available in Prolog.

The predicate `boundTypeOf/3` works by associating the same logical variable, denoting its type, to each of its occurrences. As a unique logic variable is associated to each leaf `v/1` corresponding via its de Bruijn index to the same binder, types are consistently inferred. This is ensured by the use of the built-in `nth0(I,Vs,V0)` that unifies `V0` with the `I`-th element of the type context `Vs`. Note that unification with occurs-check needs to be used to avoid cycles in the inferred type formulas.

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

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

Example 5
Type inferred for the `S` combinator $\lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\ x_2)\ (x_1\ x_4))$ in de Bruijn form.

```
?- X=l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),deBruijnTypeOf(X,T,0).
X = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))),
T = ((A> (B>C))> ((A>B)> (A>C))).
```

However, as we are only interested in simple types of closed terms 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 `bindTypeB/1`.

```
boundTypeOf(A,T):-deBruijnTypeOf(A,T0,[]),bindTypeB(T0),!,T=T0.
```

Example 6
Simple type inferred for the `S` combinator and failure to assign a type to the `Y` combinator $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_2))\ \lambda x_2.(x_1\ (x_2\ x_2)))$.

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

4 Generating families of lambda terms

We can see our compressed de Bruijn terms as binary trees decorated with integer labels. The binary trees provide a skeleton that describes the applicative structure of the underlying lambda terms. At the same time, types in the *simple typed lambda calculus* (Barendregt 1991) share a similar binary tree structure.

4.1 Generating binary trees

Binary trees are among the most well-known members of the Catalan family of combinatorial objects (Stanley 1986), that has at least 58 structurally distinct members, covering several data structures, geometric objects and formal languages.

We will build binary trees with the constructor `>/2` for branches and the constant `x` for its leaves. This will match the usual notation for simple types (Barendregt 1991) of lambda terms that can be represented as binary trees.

4.1.1 Generating binary trees of given depth

A generator / recognizer of binary trees of a limited depth, counted by entry `A003095` in (Sloane 2014) is defined by the predicate `genTreeByDepth/2`.

```
genTreeByDepth(_,x).
genTreeByDepth(D1,(X>Y)):-down(D1,D2),
  genTreeByDepth(D2,X),
  genTreeByDepth(D2,Y).
```

Example 7
illustrates trees of depth at most 2 generated by the predicate `genTreeByDepth`.

```
?- genTreeByDepth(2,T).
T = x ;
T = (x>x) ;
T = (x> (x>x)) ;
T = ((x>x)>x) ;
T = ((x>x)> (x>x)).
```

4.1.2 Generating binary trees of given size

A generator / recognizer of binary trees of a fixed size (seen as the number of internal nodes, counted by entry `A000108` in (Sloane 2014)) is defined by the predicate `genTree/2`.

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

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

Note the creative use of Prolog's DCG-grammar transformation. After the DCG expansion, the code for `genTree/3` becomes something like:

```
genTree(x,K,K).
genTree((X>Y),K1,K3):-down(K1,K2),
  genTree(X,K2,K3),
  genTree(K3,K4).
```

Given that down(K1,K2) unfolds to `K1>0,K2 is K1-1` it is clear that this code ensures that the total number of nodes `N` passed by `genTree/2` to `genTree/3` controls the size of the generated trees. We will reuse this pattern through the paper, as it simplifies the writing of generators for various combinatorial objects. It is also convenient to standardize on the number of *internal nodes* as defining the *size* of our terms.

Example 8
illustrates trees with 3 internal nodes (built with the constructor "`>/2`") generated by `genTree/2`.

```
?- genTree(3,BT).
BT = (x> (x> (x>x))) ;
BT = (x> ((x>x)>x)) ;
BT = ((x>x)> (x>x)) ;
BT = ((x> (x>x))>x) ;
BT = (((x>x)>x)>x) .
```

The predicate `tsize` defines the size of a binary 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.
```

4.2 Generating Motzkin trees

Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2. Thus they can be seen as an abstraction of lambda terms that ignores de Bruijn indices at the leaves. The predicate `motzkinTree/2` generates Motzkin trees with L internal and leaf nodes.

```
motzkinTree(L,T):-motzkinTree(T,L,0).

motzkinTree(u)-->down.
motzkinTree(l(A))-->down,motzkinTree(A).
motzkinTree(a(A,B))-->down,motzkinTree(A),motzkinTree(B).
```

Motzkin-trees are counted by the sequence `A001006` in (Sloane 2014). If we replace the first clause of `motzkinTree/2` with `motzkinTree(u)-->[]`, we obtain binary-unary trees with `L` internal nodes, counted by the entry `A006318` (Large Schröder Numbers) of (Sloane 2014).

4.3 Generating closed lambda terms in standard notation

With logic variables representing binders and their occurrences, one can also generate lambda terms in standard notation directly. The predicate `genLambda/2` equivalent to `genStandard/2`, builds a list of logic variables as it generates binders. When generating a leaf, it picks nondeterministically one of the binders among the list of binders available, `Vs`. As usual, the predicate `down/2` controls the number of internal nodes.

```
genLambda(L,T):-genLambda(T,[],L,0).

genLambda(X,Vs)-->{member(X,Vs)}.
genLambda(l(X,A),Vs)-->down,genLambda(A,[X|Vs]).
genLambda(a(A,B),Vs)-->down,genLambda(A,Vs),genLambda(B,Vs).
```

To generate lambda terms of a given size, we can write generators similar to the ones for binary trees in section 4.1. Moreover, we have the choice to use generators for standard, de Bruijn or compressed de Bruijn terms and then bijectively morph the resulting terms in the desired representation, as outlined is section 2.5.

4.4 Deriving a generator for lambda terms in de Bruijn form

We can derive a generator for closed lambda terms in de Bruijn form by extending a *Motzkin* or *unary-binary* tree generator to keep track of the lambda binders. When reaching a leaf `v/1`, one of the available binders (expressed as a de Bruijn index) will be assigned to it nondeterministically.

The predicate `genDBterm/4` generates closed de Bruijn terms with a fixed number of internal (non-index) nodes, as counted by entry `A220894` in (Sloane 2014).

```
genDBterm(v(X),V)-->
  {down(V,V0)},
  {between(0,V0,X)}.
genDBterm(l(A),V)-->down,
  {up(V,NewV)},
  genDBterm(A,NewV).
genDBterm(a(A,B),V)-->down,
  genDBterm(A,V),
  genDBterm(B,V).
```

The range of possible indices is provided by Prolog's built-in integer range generator `between/3`, that provides values from `0` to `V0`. Note also the use of `down/2` abstracting away the predecessor operation and `up/2` abstracting away the successor operation. Together, they control the amount of available nodes and the incrementing of de Bruijn indices at each lambda node.

Our generator of de Bruijn terms is exposed through two interfaces: `genDBterm/2` that generates closed de Bruijn terms with exactly `L` non-index nodes and `genDBterms/2` that generates terms with up to `L` non-index nodes, by not enforcing that exactly `L` internal nodes must be used.

```
genDBterm(L,T):-genDBterm(T,0,L,0).

genDBterms(L,T):-genDBterm(T,0,L,_).
```

Inserting a `down` operation in the first clause of `genDBterm/4` will enumerate terms counted by sequence `A135501` instead of `A220894`, as this would imply assuming size 1 for variables. in (Sloane 2014).

Example 9
Generation of terms with up to 2 internal nodes.

```
?- genDBterms(2,T).
T = l(v(0)) ;
T = l(l(v(0))) ;
T = l(l(v(1))) ;
T = l(a(v(0), v(0))).
```

4.5 Deriving generators for closed terms in compressed de Bruijn form

A generator for compressed de Bruijn terms can be derived by using `DCG` syntax to compose a generator for closed de Bruijn terms `genDBterm` and `genDBterms` and a transformer to compressed terms `b2c/2`.

```
genCompressed --> genDBterm,b2c.
genCompresseds--> genDBterms,b2c.
```

4.6 Generators for closed terms in standard notation

```
genStandard-->genDBterm,b2l.
genStandards-->genDBterms,b2l.
```

Example 10
illustrates generators for closed terms in compressed de Bruijn and standard notation with logic variables providing lambda variable names.

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).

?- genStandard(2,T).
T = l(_G3434, l(_G3440, _G3440)) ;
T = l(_G3434, l(_G3440, _G3434)) ;
T = l(_G3437, a(_G3437, _G3437)).
```

4.7 Generating normal forms

Normal forms are lambda terms that cannot be further reduced. A normal form should not be an application with a lambda as its left branch and, recursively, its subterms should also be normal forms. The predicate `nf/4` defines this inductively and generates all normal forms with `L` internal nodes in de Bruijn form.

```
nf(v(X),V)-->{down(V,V0),between(0,V0,X)}.
nf(l(A),V)-->down,{up(V,NewV)},nf(A,NewV).
nf(a(v(X),B),V)-->down,nf(v(X),V),nf(B,V).
nf(a(a(X,Y),B),V)-->down,nf(a(X,Y),V),nf(B,V).
```

As we standardize our generators to produce compressed de Bruijn terms, we combine `nf/4` and the converter `b2c/2` to produce normal forms of size exactly `L` (predicate `nf/2`) and with size up to `L` (predicate `nfs/2`).

```
nf(L,T):-nf(B,0,L,0),b2c(B,T).
nfs(L,T):-nf(B,0,L,_),b2c(B,T).
```

Example 11
illustrates normal forms with exactly 2 non-index nodes.

```
?- nf(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)) .
```

The number of solutions of our generator replicates entry `A224345` in (Sloane 2014) that counts closed normal forms of various sizes.

4.8 Generation of linear lambda terms

Linear lambda terms (Grygiel et al. 2013) restrict binders to *exactly one* occurrence.

The predicate `linLamb/4` uses logic variables both as leaves and as lambda binders and generates terms in standard form. In the process, binders accumulated on the way down from the root, must be split between the two branches of an application node. The predicate `subset_and_complement_of/3` achieves this by generating all such possible splits of the set of binders.

```
linLamb(X,[X])-->[].
linLamb(l(X,A),Vs)-->down,linLamb(A,[X|Vs]).
linLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  linLamb(A,As),linLamb(B,Bs).
```

At each step of `subset_and_complement_of/3`, `place_element/5` is called to distribute each element of a set to exactly one of two disjoint subsets.

```
subset_and_complement_of([],[],[]).
subset_and_complement_of([X|Xs],NewYs,NewZs):-
  subset_and_complement_of(Xs,Ys,Zs),
  place_element(X,Ys,Zs,NewYs,NewZs).
```

```
place_element(X,Ys,Zs,[X|Ys],Zs).
place_element(X,Ys,Zs,Ys,[X|Zs]).
```

As usual, we standardize the generated terms by converting them with `l2c` to compressed de Bruijn terms.

```
linLamb(L,CT):-linLamb(T,[],L,0),l2c(T,CT).
```

Example 12
illustrates linear lambda terms for `L=3`.

```
?- linLamb(3,T).
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) .
```

4.9 Generation of affine linear lambda terms

Linear affine lambda terms (Grygiel et al. 2013) restrict binders to *at most one* occurrence.

```
afLinLamb(L,CT):-afLinLamb(T,[],L,0),l2c(T,CT).

afLinLamb(X,[X|_])-->[].
afLinLamb(l(X,A),Vs)-->down,afLinLamb(A,[X|Vs]).
afLinLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  afLinLamb(A,As),afLinLamb(B,Bs).
```

Example 13
illustrates generation of affine linear lambda terms in compressed de Bruijn form.

```
?- afLinLamb(3,T).
T = v(3, 0) ;
T = a(2, v(0, 1), v(0, 0)) ;
```

```
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) ;
```

Clearly all linear terms are affine. It is also known that all affine terms are typable.

4.9.1 Generating lambda terms of bounded unary height

Lambda terms of bounded unary height are introduced in (Bodini et al. 2011) where it is argued that such terms are naturally occurring in programs and it is shown that their asymptotic behavior is easier to study.

They are specified by giving a bound on the number of lambda binders from a de Bruijn index to the root of the term.

```
boundedUnary(v(X),V,_D)-->{down(V,V0),between(0,V0,X)}.
boundedUnary(l(A),V,D1)-->down,
  {down(D1,D2),up(V,NewV)},
  boundedUnary(A,NewV,D2).
boundedUnary(a(A,B),V,D)-->down,
  boundedUnary(A,V,D),boundedUnary(B,V,D).
```

The predicate `boundedUnary/5` generates lambda terms of size `L` in compressed de Bruijn form with unary hight `D`.

```
boundedUnary(D,L,T):-boundedUnary(B,0,D,L,0),b2c(B,T).
boundedUnarys(D,L,T):-boundedUnary(B,0,D,L,_),b2c(B,T).
```

Example 14
illustrates terms of unary height 1 with size up to 3.

```
?- boundedUnarys(1,3,R).
R = v(1, 0) ;
R = a(1, v(0, 0), v(0, 0)) ;
R = a(1, v(0, 0), a(0, v(0, 0), v(0, 0))) ;
R = a(1, a(0, v(0, 0), v(0, 0)), v(0, 0)) ;
R = a(0, v(1, 0), v(1, 0)) .
```

4.10 Generating terms in binary lambda calculus encoding

Generating de Bruijn terms based on the size of their binary lambda calculus encoding (Wikipedia 2015) works by using a DCG mechanism to build the actual code as a list `Cs` of `0` and `1` digits and specifying the size of the code in advance.

```
blc(L,T,Cs):-length(Cs,L),blc(B,0,Cs,[]),b2c(B,T).

blc(v(X),V)-->{between(1,V,X)},encvar(X).
blc(l(A),V)-->[0,0],{NewV is V+1},blc(A,NewV).
blc(a(A,B),V)-->[0,1],blc(A,V),blc(B,V).
```

Note that de Bruijn binders are encoded as `00`, applications as `01` and de Bruijn indices in unary notation are encoded as `00...01`. This operation is preformed by the predicate `encvar/3`, that, in DCG notation, uses `down/2` at each step to generate the sequence of `1` terminated `0` digits.

```
encvar(0)-->[0].
encvar(N)-->{down(N,N1)},[1],encvar(N1).
```

Example 15
illustrates generation of 8-bit binary lambda terms (`Cs`) together with their compressed de Bruijn form (`T`).

```
?- blc(8,T,Cs).
T = v(3, 1),
Cs = [0, 0, 0, 0, 0, 0, 1, 0] ;
T = a(1, v(0, 1), v(0, 1)),
Cs = [0, 0, 0, 1, 1, 0, 1, 0] .
```

Note that while not bijective, the binary encoding has the advantage of being a self-delimiting code. This facilitates its use in an unusually compact interpreter.

4.11 Generating typable terms

The predicate `genTypable/2` generates closed typable terms of size `L`. These are counted by entry `A220471` in (Sloane 2014).

```
genTypable(L,T):-genCompressed(L,T),typable(T).
genTypables(L,T):-genCompresseds(L,T),typable(T).
```

Example 16
illustrates a generator for closed typable terms.

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).
```

4.12 Combining term generation and type inference

One could combine a generator for closed terms and a type inferrer in a “generate-and-test” style as follows:

```
genTypedTerm1(L,Term,Type):-
  genDBterm(L,Term),
  boundTypeOf(Term,Type).
```

Note that when one wants to select only terms having a given type this is quite inefficient. Next, we will show how to combine size-bound term generation, testing for closed terms and type inference into a single predicate. This will enable efficient querying about *what terms inhabit a given type*, as one would expect from Prolog’s multi-directional execution model.

4.13 Generating closed well-typed terms of a given size

One can derive, from the type inferrer `boundTypeOf`, a more efficient generator for de Bruijn terms with a given number of internal nodes.

The predicate `genTypedTerm/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.

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

Two interfaces are offered: `genTypedTerm` that generates de Bruijn terms of with exactly `L` internal nodes and `genTypedTerms` that generates terms with `L` internal nodes or less.

```
genTypedTerm(L,B,T):-
  genTypedTerm(B,T,[],L,0),
  bindTypeB(T).

genTypedTerms(L,B,T):-
  genTypedTerm(B,T,[],L,_),
  bindTypeB(T).
```

As expected, the number of solutions, computed as the sequence 1, 2, 9, 40, 238, 1564, 11807, 98529, 904318, 9006364, 96709332, 1110858977 ... for sizes $1,2,3,\ \dots\ ,12,\ \dots$ matches entry `A220471` in (Sloane 2014). Note that the last 2 terms are not (yet) in the `A220471` in (Sloane 2014) as the generate and filter method used in (Grygiel and Lescanne 2013) is limited by the super-exponential growth of the closed lambda terms among which the relatively few well-typed ones need to be found (e.g. more than 12 billion terms for size 12). Interestingly, by interleaving generation of closed terms and type inference in the predicate `genTypedTerm` the time to generate all the well-typed terms is actually shorter than the time to generate all closed terms of the same size, e.g.. 3.2 vs 4.3 seconds for size 9 with SWI-Prolog. As via the Curry-Howard isomorphism closed simply typed terms correspond to proofs of tautologies in minimal logic, co-generation of terms and types corresponds to co-generation of tautologies and their proofs for proofs of given length.

Example 17
Generation of well-typed closed de Bruijn terms of size 3.

```
?- genTypedTerm(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))) .
```

5 Normalization of lambda 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`, under the assumption that `X` does not occur in `B` and η*-conversion*, the transformation of an application term `a(l(X,A),X)` into `A`, under the assumption that X does not occur in `A`.

The first tool we need to implement normalization of lambda terms is a safe substitution operation. In lambda-calculus based functional languages this can be achieved through a HOAS (Higher-Order Abstract Syntax) mechanism, that borrows the substitution operation from the underlying "meta-language". To this end, lambdas are implemented as functions which get executed (usually lazily) when substitutions occur. We refer to (Pfenning and Elliot 1988) for the original description of this mechanism, widely used these days for implementing embedded domain specific languages and proof assistants in languages like Haskell or ML.

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.

We will devise here a simple and safe interpreter for lambda terms supporting normal order β-reduction by using de Bruijn terms, which 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.
```

Normal order evaluation of a lambda term, if it terminates, leads to a unique normal form, as a consequence of the Church-Rosser theorem, elegantly proven in (de Bruijn 1972) using de Bruijn terms. Termination holds, for instance, in the case of simply typed lambda terms. Its implementation is well known; we will follow here the algorithm described in (Sestoft 2002). 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 `to_nf` implements normal order reduction. It 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.

```
to_nf(v(X),v(X)).
to_nf(l(E),l(NE)):-to_nf(E,NE).
to_nf(a(E1,E2),R):-wh_nf(E1,NE),to_nf1(NE,E2,R).
```

Case analysis of application terms for possible β-reduction is performed by `to_nf1/3`, where the second clause calls `beta/3` and recurses on its result.

```
to_nf1(v(E1),E2,a(v(E1),NE2)):-to_nf(E2,NE2).
to_nf1(l(E),E2,R):-beta(l(E),E2,NewE),to_nf(NewE,R).
to_nf1(a(A,B),E2,a(NE1,NE2)):-to_nf(a(A,B),NE1),to_nf(E2,NE2).
```

Therefore, the predicate `evalDeBruijn`

```
evalDeBruijn --> to_nf.
```

provides a Turing-complete lambda calculus interpreter working on de Bruijn terms. It is guaranteed to compute a normal form, if it exists. The predicate `evalStandard/2` works on standard lambda terms, that in converts to de Bruijn terms and then back after evaluation. The predicate `evalCompressed/2` works in a similar way on compressed de Bruijn terms. We express them as a composition of functions (first argument in, second out) using Prolog's DCG notation.

```
evalStandard-->l2b,to_nf,b2l.
evalCompressed-->c2b,to_nf,b2c.
```

Example 18
illustrates evaluation of the lambda term $SKK =$
$((\ \lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\, x_2)\ (x_1\, x_2))\ \ \lambda x_3.\ \lambda x_4.x_3)\ \ \lambda x_5.\ \lambda x_6.x_5)$ in compressed de Brijn form, resulting in the definition of the identity combinator $I = \lambda x_0.x_0$.

```
?- S=a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),K=v(2,1),
     evalCompressed(a(0,a(0,S,K),K),R).
S = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
K = v(2, 1),
R = v(1, 0).
```

6 Combinators

Combinators are closed lambda terms placed exclusively as labels at the leaves of application trees. Thus combinator expressions are lambda terms represented as binary trees having applications as internal nodes and combinators as leaves. We will explore here two families of combinator expressions, one well-known (SK-combinators) and another that has been mostly forgotten for a more than a half century (Rosser's X-combinator).

6.1 SK-Combinator Trees

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 $K =\ \lambda x_0.\ \lambda x_1.x_0$ and $S = \lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\, x_2)\ (x_1\, x_2))$. 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.

6.1.1 Generating combinator trees

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

```
genSK(k)-->[].
genSK(s)-->[].
genSK(X*Y)-->down,genSK(X),genSK(Y).
```

Note the use of Prolog's definite clause grammar (DCG) notation in combination with the predicate `down/2` that counts downward the number of available internal nodes.

The predicate genSK/3 provides two interfaces: genSK/2 that generates trees with exactly *N* internal nodes and genSKs/2 that generates trees with *N* or less internal nodes.

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

```
genSKs(N,X):-genSK(X,N,_).
```

Example 19
SK-combinator trees with up to 1 internal nodes (and up to 2 leaves).

```
?- genSKs(1,T).
T = k ;
T = s ;
T = k*k ;
T = k*s ;
T = s*k ;
T = s*s .
```

The predicate csize defines the size of an SK-combinator tree in terms of the number of its internal nodes.

```
csize(k,0).
csize(s,0).
csize((X*Y),S):-csize(X,A),csize(Y,B),S is 1+A+B.
```

6.1.2 A Turing-complete evaluator for SK-combinator trees

An evaluator for SK-combinator trees recurses over application nodes, evaluates their subtrees and then applies the left one to the right one.

```
evalSK(k,k).
evalSK(s,s).
evalSK(F*G,R):-evalSK(F,F1),evalSK(G,G1),appSK(F1,G1,R).
```

In the predicate app, 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.

```
appSK((s*X)*Y,Z,R):-!,  % S
  appSK(X,Z,R1),
  appSK(Y,Z,R2),
  appSK(R1,R2,R).
appSK(k*X,_Y,R):-!,R=X. % K
appSK(F,G,F*G).
```

Example 20
Applications of SKK and SKS, both implementing the identity combinator $I = \lambda x.x$.

```
?- appSK(s*k*k,s,R).
R = s.

?- appSK(s*k*s,k,R).
R = k.
```

6.1.3 De Bruijn equivalents of SK-combinator expressions

De Bruijn indices (de Bruijn 1972) provide a *name-free* representation of lambda terms. All terms closed 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, $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$ 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))))))).
```

The predicate `sk2b` transforms an SK-combinator tree in its lambda expression form, in de Bruijn notation, by replacing leaves with their de Bruijn form of the S and K combinators and replacing recursively the constructor “*”/2 with the application nodes “a”/2.

```
sk2b(s,S):-sB(S).
sk2b(k,K):-kB(K).
sk2b((X*Y),a(A,B)):-sk2b(X,A),sk2b(Y,B).
```

Example 21
Expansion of some small SK-combinator trees to de Bruijn forms.

```
?- sk2b(k*k,R).
R = a(l(l(v(1))), l(l(v(1)))).

?- sk2b(k*s,R).
R = a(l(l(v(1))),l(l(l(a(a(v(2),v(0)),a(v(1),v(0))))))).
```

Clearly their de Bruijn equivalents are significantly larger than the corresponding combinator trees, but it is easy to see that this is only by a constant factor, i.e. at most the size of the S combinator.

A lambda term is called *closed* if it contains no free variables.

Proposition 1
The lambda terms equivalent to SK-combinators computed by `sk2b` are closed.

Proof
As the lambda term equivalent of the SK-combinator term is clearly a closed expression, the proposition follows from the definition of `sk2b`, as it builds terms that apply closed terms to closed terms. □

This well-known property holds, in fact, for all combinator expressions. It follows that combinator expressions have a stronger, *hereditary* closedness property: every subtree of a combinator tree also represents a closed expression.

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 SK-combinator trees. In particular, we would like to have an idea on the asymptotic density of well-typed SK-combinator tree expressions. We will take advantage of Prolog's sound unification algorithm to define a type inferrer directly on SK-terms.

6.2 Inferring simple types for SK-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. However, this is equivalent to just borrowing the well-known types of the `S` and `K` combinators and then recurse over application nodes.

We will next describe an algorithm for inferring types directly on SK-combinator trees.

6.2.1 A type inference algorithm for SK-terms

Simple types will be defined here also as binary trees built with the constructor "`>/2`" with empty leaves, representing the unique primitive type "`x`". For brevity, we will mean simple types when mentioning types, from now on. Types can be seen as as a "binary tree approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (called *strong normalization*), as the following well known property states (Barendregt 1991).

Proposition 2
Lambda terms (and combinator expressions, in particular) that have simple types are strongly normalizing.

When modeling lambda terms in a functional or procedural language, inferring types requires implementing unification with occurs-check, as shown for instance in the appendix of (Grygiel and Lescanne 2013). On the other hand this operation is readily available in today's Prolog systems.

```
skTypeOf(k,(A>(_B>A))).
skTypeOf(s,(((A>(B>C))> ((A>B)>(A>C) )))).
skTypeOf(A*B,Y):-
  skTypeOf(A,T),
  skTypeOf(B,X),
  unify_with_occurs_check(T,(X>Y)).
```

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

Example 22
Type inferred for some SK-combinator expressions. Note the failure to infer a type for $SSI = SS(SKK)$.

```
?- skTypeOf((((k*k)*k)*k)*k,T).
T = (A>(B>A)).
?- skTypeOf((k*s)*k,T).
T = ((A> (B>C))> ((A>B)> (A>C))).
?- skTypeOf((s*s)*((s*k)*k),T).
false.
```

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

```
simpleTypeOf(A,T):-
  skTypeOf(A,T),
  bindWithBaseType(T).

% bind all variables with type 'x'
bindWithBaseType(x):-!.
bindWithBaseType((A>B)):-
  bindWithBaseType(A),
  bindWithBaseType(B).
```

Example 23
Simple type inferred for combinators KSK, B and C.

```
?- simpleTypeOf(k*s*k,T).
T = ((x> (x>x))> ((x>x)> (x>x))).
?- B=s*(k*s)*k,C=s*(B*B*s)*(k*k),simpleTypeOf(B,TB),simpleTypeOf(C,TC).
B = s* (k*s)*k,
C = s* (s* (k*s)*k* (s* (k*s)*k)*s)* (k*k),
TB = ((x>x)> ((x>x)> (x>x))),
TC = ((x> (x>x))> (x> (x>x))).
```

It is also useful to define the predicate `typableSK` that succeeds when a type can be inferred.

```
typableSK(X):-skTypeOf(X,_).
```

6.3 Rosser's X-combinator

We will know explore expressions built with a less well-known combinator, that provides a 1-point basis for combinator calculi.

It is shown in (Goldberg 2004) 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.

6.3.1 The X-combinator in terms of S and K and vice-versa

A derivation of Rosser's X-combinator is described in (Fokker 1992).

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 (Fokker 1992) the following holds.

$$KK = XX = \lambda x_0.\ \lambda x_1.\ \lambda x_2.x_1 \tag{3}$$

As a result, X-combinator expressions are within a (small, see Prop. 3) 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 (Hindley and Seldin 2008)).

Generating the combinator trees As X-combinator trees are just plain binary trees, with leaves denoted `x` and internal nodes denoted `>`, we can reuse the predicate `genTree` defined in 4.1.2 to generate X-combinator trees with a given number of internal nodes.

6.3.2 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.

```
evalX((F>G),R):-!,evalX(F,F1),evalX(G,G1),appX(F1,G1,R).
evalX(X,X).
```

In the predicate `appX/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.

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

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

Example 24
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)),evalX(SKK>x,R).
SKK = (((x>(x>x))>((x>x)>x))>((x>x)>x)),
R = x.
```

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

6.3.3 De Bruijn equivalents of X-combinator expressions

De Bruijn indices (de Bruijn 1972) 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, $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$ 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`.

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 25
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 $\lambda x_0.\ \lambda x_1.\ \lambda x_2.x_1$.

Proposition 3
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. □

Note also that the lambda terms equivalent to X-combinators computed by `t2b` are closed, given that the lambda term equivalent to the X-combinator is a closed expression and `t2b` builds terms that apply only closed terms to closed terms.

6.4 Comparing the two evaluators

While, as shown in subsection 6.3.2, 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.

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 --> evalX,t2b.
evalAsB --> t2b,evalDeBruijn.
```

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

Example 26
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),dbTermSize(R,Size),write(Size),nl,fail.
29

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

Note however, as predicted by the Church-Rosser theorem (de Bruijn 1972; Barendregt 1984), applying normalization via `evalDeBruijn` to the result of `evalAsT` reaches the same final normal form. This property is called *confluence*.

Example 27
Confluence of evaluation as X-combinator tree and as lambda term.

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

6.5 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.

6.5.1 Type trees as combinator trees

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, that we will explore in subsection 8.7.

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),boundTypeOf(B,T).
```

Observe that this only makes sense if the combinator basis is well-typed. Fortunately this is the case of the X-combinator $\lambda x_0.(((x_0$
$\lambda x_1.\ \lambda x_2.x_1)\ \ \lambda x_3.\ \lambda x_4.\ \lambda x_5.((x_3\ x_5)\ (x_4\ x_5)))\ \ \lambda x_6.\ \lambda x7.x_6)$.

Example 28
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).
```

6.5.2 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),bindTypeB(T).

xT(T):-t2b(x,B),boundTypeOf(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 `bindTypeB`.

Example 29
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).
```

7 Size-proportionate bijective encodings of lambda terms and combinators

We will describe here two encodings. The first one, in subsections 7.1 and 7.2 does it "the hard way" by working with bitstring-represented natural number codes. The second one, in subsections 7.3 and 7.4 does it "the easy-way", by defining an alternate, tree-based natural number representation to which fairly strait-forward bijections exist from combinator expressions, lambda terms and types.

7.1 An encoding based on Cantor's $\mathbb{N}^k$ to $\mathbb{N}$ bijection

We can see our compressed de Bruijn terms as binary trees decorated with integer labels. The underlying binary trees provide a skeleton that describes the applicative structure of the terms.

7.1.1 The Catalan family of combinatorial objects

Binary trees are among the most well-known members of the Catalan family of combinatorial objects (Stanley 1986), that has at least 58 structurally distinct members, covering several data structures, geometric objects and formal languages.

7.1.2 Size-proportionate encodings

In the presence of a bijection between two, usually infinite sets of data objects, it is possible that representation sizes on one side or the other are exponentially larger that on the other. Well-known encodings like Ackermann's bijection for hereditarily finite sets to natural numbers, defined as $f(\{\}) = 0, f(x) = \sum_{a \in x} 2^{f(a)}$, fall in this category.

We will say that a bijection is *size-proportionate* if the representation sizes for corresponding terms on its two sides are "close enough" up to a constant factor multiplied with at most the logarithm of any of the sizes.

Definition 1
Given a bijection between sets of terms of two datatypes denoted M and N, $f : M \to N$, let $m(x)$ be the representation size of a term $x \in M$ and $n(y)$ be the representation size of $y \in N$. Then f is called *size-proportionate* if $|m(x) - n(y)| \in O(log(max(m(x), n(y))))$.

Informally we also assume that the constants involved are small enough such that the printed representation of two data objects connected by the bijections is about the same.

7.1.3 The language of balanced parentheses

Binary trees are in a well-known size-proportionate bijection with the language of balanced parentheses (Stanley 1986), from which we will borrow an efficient ranking/unranking bijection. The reversible predicate `catpar/2` transforms between binary trees and lists of balanced parentheses, with `0` denoting the open parentheses and `1` denoting the closing one.

```
catpar(T,Ps):-catpar(T,0,1,Ps,[]).
catpar(X,L,R) --> [L],catpars(X,L,R).

catpars(x,_,R) --> [R].
catpars((X>Xs),L,R)-->catpar(X,L,R),catpars(Xs,L,R).
```

Example 30
illustrates the work of the reversible predicate `catpar/2`.

```
?- catpar(((x>x)>(x>x)),Ps),catpar(T,Ps).
Ps = [0, 0, 0, 1, 1, 0, 1, 1], T = ((x>x)> (x>x)) .
```

Note the extra opening/closing parentheses, compared to the usual definition of Dyck words (Stanley 1986), that make the sequence self-delimiting.

7.1.4 A bijection from the language of balanced parenthesis lists to $\mathbb{N}$

This algorithm follows closely the procedural implementation described in (Kreher and Stinson 1999).

The code of the helper predicates called by `rankCatalan` and `unrankCatalan` is provided in `http://www.cse.unt.edu/~tarau/research/2015/dbr.pro`. The details of the algorithms for computing `localRank` and `localunRank` are described at `http://www.cse.unt.edu/~tarau/research/2015/dbrApp.pdf`.

The predicate `rankCatalan` uses the Catalan numbers computed by `cat` in `rankLoop` to shift the ranking over the ranks of smaller sequences, after calling `localRank`.

```
rankCatalan(Xs,R):-
  length(Xs,XL),XL>=2,
  L is XL-2, I is L // 2,
  localRank(I,Xs,N),
  S is 0, PI is I-1,
  rankLoop(PI,S,NewS),
  R is NewS+N.
```

The predicate `unrankCatalan` uses the Catalan numbers computed by `cat` in `unrankLoop` to shift over smaller sequences, before calling `localUnrank`.

```
unrankCatalan(R,Xs):-
  S is 0, I is 0,
  unrankLoop(R,S,I,NewS,NewI),
  LR is R-NewS,
  L is 2*NewI+1,
  length(As,L),
  localUnrank(NewI,LR,As),
  As=[_|Bs],
  append([0|Bs],[1],Xs).
```

The following example illustrates the ranking and unranking algorithms:

```
?- unrankCatalan(2015,Ps),rankCatalan(Ps,Rank).
Ps = [0,0,1,0,1,0,1,0,0,0,0,1,0,1,1,1,1,1],Rank = 2015
```

7.1.5 Ranking and unranking simple types

After putting together the bijections between binary trees and balanced parentheses with the ranking/unranking of the later we obtain the size-proportionate ranking/unranking algorithms for simple types.

```
rankType(T,Code):-
  catpar(T,Ps),
  rankCatalan(Ps,Code).

unrankType(Code,Term):-
  unrankCatalan(Code,Ps),
  catpar(Term,Ps).
```

Example 31
illustrates the ranking and unranking of simple types.

```
?- I=100, unrankType(I,T),rankType(T,R).
I = R, R = 100,
T = (((x>x)> ((x> (x>x))>x))>x) .
```

As there are $O(\frac{4^n}{n^{\frac{3}{2}}})$ binary trees of size n corresponding to 2^n natural numbers of bitsize up to n and our ranking algorithm visits them in lexicographic order, it follows that:

Proposition 4
The bijection between types and their ranks is size-proportionate.

7.1.6 Catalan skeletons of compressed de Bruijn terms

As compressed de Bruijn terms can be seen as binary trees with labels on their leaves and internal nodes, their "Catalan skeleton" is simply the underlying binary tree. The predicate `cskel/3` extracts this skeleton as well as the list of the labels, in depth-first order, as encountered in the process.

```
cskel(S,Vs, T):-cskel(T,S,Vs,[]).

cskel(v(K,N),x)-->[K,N].
cskel(a(K,X,Y),(A>B))-->[K],cskel(X,A),cskel(Y,B).
```

The predicates `toSkel` and `fromSkel` add conversion between this binary tree and lists of balanced parenthesis by using the (reversible) predicate `catpar`.

```
toSkel(T,Skel,Vs):-
  cskel(T,Cat,Vs,[]),
  catpar(Cat,Skel).
```

```
fromSkel(Skel,Vs, T):-
  catpar(Cat,Skel),
  cskel(T,Cat,Vs,[]).
```

Example 32
illustrates the Catalan skeleton `Skel` and the list of variable labels `Vs` extracted from a compressed de Bruijn term corresponding to the `S` combinator.

```
?- T = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
   toSkel(T,Skel,Vs),fromSkel(Skel,Vs,T1).
T = T1, T1 = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
Skel = [0,0,0,1,1,0,1,1],Vs = [3,0,0,2,0,0,0,0,1,0,0] .
```

7.1.7 The generalized Cantor k-tupling bijection

As we we have already solved the problem of ranking and unranking lists of balanced parentheses, the remaining problem is that of finding a bijection between the lists of labels collected from the nodes of a compressed de Bruijn term and natural numbers.

We will use the generalized Cantor bijection between $\mathbb{N}^n$ and $\mathbb{N}$ as the first step in defining this bijection. The formula, given in (Cegielski and Richard 1999) p.4, looks as follows:

$$K_n(x_1,\ldots,x_n) = \sum_{k=1}^{n} \binom{k-1+s_k}{k} \; where \; s_k = \sum_{i=1}^{k} x_i \tag{4}$$

Note that $\binom{n}{k}$ represents the number of subsets of k elements of a set of n elements, that also corresponds to the binomial coefficient of x^k in the expansion of $(x+y)^n$, and $K_n(x_1,\ldots,x_n)$ denotes the natural number associated to the tuple $(x_1,\ldots,x_n)$. It is easy to see that the generalized Cantor n-tupling function defined by equation (4) is a polynomial of degree n in its arguments.

7.1.8 The bijection between sets and sequences of natural numbers

We recognize in the equation (4) the *prefix sums* s_k incremented with values of k starting at 0. It represents the “set side” of the bijection between sequences of n natural numbers and sets of n natural numbers described in (Tarau 2009). It is implemented in the online Appendix as the bijection `list2set` together with its inverse `set2list`. For example, `list2set` transforms `[2,0,1,5]` to [2, 3, 5, 11] as `3=2+0+1,5=3+1+1,11=5+5+1` and `set2list` transforms it back by computing the differences between consecutive members, reduced by 1.

7.1.9 The $\mathbb{N}^n \to \mathbb{N}$ bijection

The bijection $K_n : \mathbb{N}^n \to \mathbb{N}$ is basically just summing up a set of binomial coefficients. The predicate `fromCantorTuple` implements the the $\mathbb{N}^n \to \mathbb{N}$ bijection in Prolog, using the predicate `fromKSet` that sums up the binomials in formula 4 using the predicate `untuplingLoop`, as well as the sequence to set transformer `list2set`.

```
fromCantorTuple(Ns,N):-
  list2set(Ns,Xs),
  fromKSet(Xs,N).

fromKSet(Xs,N):-untuplingLoop(Xs,0,0,N).

untuplingLoop([],_L,B,B).
untuplingLoop([X|Xs],L1,B1,Bn):-L2 is L1+1,
  binomial(X,L2,B),B2 is B1+B,
  untuplingLoop(Xs,L2,B2,Bn).
```

7.1.10 The $\mathbb{N} \to \mathbb{N}^n$ bijection

We split our problem in two simpler ones: inverting `fromKSet` and then applying `set2list` to get back from sets to lists.

We observe that the predicate `untuplingLoop` used by `fromKSet` implements the sum of the combinations $\binom{X_1}{1} + \binom{X_2}{2} + \ldots + \binom{X_K}{K} = N$, which is nothing but the representation of N in the *combinatorial number system of degree K* due to (Lehmer 1964). Fortunately, efficient conversion algorithms between the conventional and the combinatorial number system are well known, (Knuth 2005).

We are ready to implement the Prolog predicate `toKSet(K,N,Ds)`, which, given the degree `K`, indicating the number of "combinatorial digits", finds and repeatedly subtracts the greatest binomial smaller than `N`. It calls the predicate `combinatorialDigits` that returns these "digits" in increasing order, providing the canonical set representations that `set2list` needs.

```
toKSet(K,N,Ds):-combinatorialDigits(K,N,[],Ds).

combinatorialDigits(0,_,Ds,Ds).
combinatorialDigits(K,N,Ds,NewDs):-K>0,K1 is K-1,
  upperBinomial(K,N,M),M1 is M-1,
  binomial(M1,K,BDigit),N1 is N-BDigit,
  combinatorialDigits(K1,N1,[M1|Ds],NewDs).
```

```
upperBinomial(K,N,R):-S is N+K,
  roughLimit(K,S,K,M),L is M // 2,
  binarySearch(K,N,L,M,R).
```

The predicate `roughLimit` compares successive powers of 2 with binomials $\binom{I}{K}$ and finds the first `I` for which the binomial is between successive powers of 2.

```
roughLimit(K,N,I, L):-binomial(I,K,B),B>N,!,L=I.
roughLimit(K,N,I, L):-J is 2*I,
  roughLimit(K,N,J,L).
```

The predicate `binarySearch` finds the exact value of the combinatorial digit in the interval `[L,M]`, narrowed down by `roughLimit`.

```
binarySearch(_K,_N,From,From,R):-!,R=From.
binarySearch(K,N,From,To,R):-Mid is (From+To) // 2,binomial(Mid,K,B),
  splitSearchOn(B,K,N,From,Mid,To,R).
splitSearchOn(B,K,N,From,Mid,_To,R):-B>N,!,
  binarySearch(K,N,From,Mid,R).
splitSearchOn(_B,K,N,_From,Mid,To,R):-Mid1 is Mid+1,
  binarySearch(K,N,Mid1,To,R).
```

The predicates `toKSet` and `fromKSet` implement inverse functions, mapping natural numbers to canonically represented sets of `K` natural numbers.

```
?- toKSet(5,2014,Set),fromKSet(Set,N).
Set = [0, 3, 4, 5, 14], N = 2014 .
```

The efficient inverse of Cantor's N-tupling is now simply:

```
toCantorTuple(K,N,Ns):-
  toKSet(K,N,Ds),
  set2list(Ds,Ns).
```

Example 33
illustrates the work of the generalized cantor bijection, on some large numbers:

```
?- K=1000,pow(2014,103,N),toCantorTuple(K,N,Ns),fromCantorTuple(Ns,N).
K = 1000, N = 208029545585703688484419851459547264831381665...567744,
Ns = [0, 0, 2, 0, 0, 0, 0, 0, 1|...] .
```

As the image of a tuple is a polynomial of degree n it means that the its bitsize is within constant factor of the sum of the bitsizes of the members of the tuple, thus:

Proposition 5
The bijection between $\mathbb{N}^n$ and $\mathbb{N}$ is size-proportionate.

7.2 Ranking/unranking of compressed de Bruijn terms

We will implement a size-proportionate bijective encoding of compressed de Bruijn terms following the technique described in (Tarau 2013). The algorithm will split a lambda tree into its *Catalan skeleton* and the list of atomic objects labeling its nodes. In our case, the Catalan skeleton abstracts away the applicative structure of the term. It also provides the key for decoding unambiguously the integer labels in both the leaves (two integers) and internal nodes (one integer). Our ranking/unranking algorithms will rely on the encoding/decoding of the Catalan skeleton provided by the predicates `rankCatalan/2` and `unrankCatalan/2` as well as for the encoding/decoding of the labels, provided by the predicates `toCantorTuple/3` and `fromCantorTuple/2`.

The predicate `rankTerm/2` defines the bijective encoding of a (possibly open) compressed de Bruijn term.

```
rankTerm(Term,Code):-
  toSkel(Term,Ps,Ns),
  rankCatalan(Ps,CatCode),
  fromCantorTuple(Ns,VarsCode),
  fromCantorTuple([CatCode,VarsCode],Code).
```

The predicate `rankTerm/2` defines the bijective decoding of a natural number into a (possibly open) compressed de Bruijn term.

```
unrankTerm(Code,Term):-
  toCantorTuple(2,Code,[CatCode,VarsCode]),
  unrankCatalan(CatCode,Ps),
  length(Ps,L2),L is (L2-2) div 2, L3 is 3*L+2,
  toCantorTuple(L3,VarsCode,Ns),
  fromSkel(Ps,Ns,Term).
```

Note that given the unranking of `CatCode` as a list of balanced parentheses of length `2*L+2`, we can determine the number `L` of internal nodes of the tree and the number `L+1` of leaves. Then we have `2*(L+1)` labels for the leaves and `L` labels for the internal nodes, for a total of `3L+2`, value needed to decode the labels encoded as `VarsCode`.

It follows from Prop. 4 and Prop. 5 that:

Proposition 6
A compressed de Bruijn terms is size-proportionate to its rank.

Example 34
illustrates the "size-proportionate" encoding of the compressed de Bruijn terms corresponding to the combinators `S` and `Y`.

```
?- T = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
   rankTerm(T,R),unrankTerm(R,T1).
T = T1,T1 = a(3,a(0,v(0,2),v(0,0)),a(0,v(0, 1),v(0,0))),
R = 56493141 .

?- T=a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),
   rankTerm(T,R),unrankTerm(R,T1).
T=T1,T1=a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),
R = 261507060 .
```

7.2.1 Generation of lambda terms via unranking

While direct enumeration of terms constrained by number of nodes or depth is straightforward in Prolog, an unranking algorithm is also usable for term generation, including generation of random terms.

Generating open terms in compressed de Bruijn form Open terms are generated simply by iterating over an initial segment of $\mathbb{N}$ with the built-in `between/3` and calling the predicate `unrankTerm/2`.

```
ogen(M,T):-between(0,M,I),unrankTerm(I,T).
```

Reusing unranking-based open term generators for more constrained families of lambda terms works when their asymptotic density is relatively high.

Generating closed and well-typed terms in compressed de Bruijn form The extensive quantitative analysis available in the literature (Grygiel and Lescanne 2013; David et al. 2009; David et al. 2010) indicates that density of closed and typed terms decreasing very quickly with size, making generation by filtering impractical for very large terms.

The predicate `cgen/2` generates closed terms by filtering the results of `ogen/2` with the predicate `isClosedC` and `tgen` generates typable terms by filtering the results of `cgen/2` with `typable/2`.

```
cgen(M,IT):-ogen(M,IT),isClosedC(IT).

tgen(M,IT):-cgen(M,IT),typable(IT).
```

Example 35
Generation of well-typed terms via unranking.

```
?- tgen(200,T).
T = v(1, 0) ;
T = v(2, 0) ;
T = v(2, 1) ;
T = v(3, 0) ;
T = v(3, 1) ;
T = v(4, 0) ;
T = a(0, v(1, 0), v(1, 0)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = v(3, 2) ;
T = v(4, 1) .
false.
```

7.3 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 (Hartmanis and Baker 1974) among which Gödel's incompleteness theorems are the most significant (Gödel 1931).

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 (Grygiel and Lescanne 2013; Tarau 2013), 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 7.4.

7.3.1 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

$$n = b_0^{k_0} b_1^{k_1} \dots b_i^{k_i} \dots b_m^{k_m} \tag{5}$$

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

Proposition 7
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 (6)

$$f^i(j) = 2^i(j+1)-1 \tag{6}$$

Observe that each block 1^i in n, represented as $1^i j$ in equation (5), corresponds to the iterated application of f, i times, $n = f^i(j)$. □

Proposition 8
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 (5). 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 (5).

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.

$$cons(i,j) = \begin{cases} 2^{i+1} j & \text{if } j \text{ is odd,} \\ 2^{i+1}(j+1)-1 & \text{if } j \text{ is even.} \end{cases} \tag{7}$$

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 9
The equation (7) 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 36
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 37
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 10
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 38
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].
```

7.3.2 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. 8 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 11
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 (Tarau 2014b). □

Proposition 12
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.

$$t(A,X),s(X,Y),B\ is\ A+1,n(Y,C) \to B=C \tag{8}$$

$$t(A,X),p(X,Y),B\ is\ A-1,n(Y,C) \to B=C \tag{9}$$

Proof
See (Tarau 2014b). □

Example 39
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 (Tarau 2014d) and (Tarau 2014c) for the description of algorithms covering all the usual arithmetic operations with equivalent representations working on other members of the Catalan family and to (Tarau 2014b) 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 7.4 they will play this role for general lambda terms.

We refer to (Tarau 2014b) 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.

7.4 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.

7.4.1 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 13
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 40
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)))))) .
```

8 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.`

Size	Slow `x>x`	Slow `x>(x>x)`	Fast `x>x`	Fast `x>(x>x)`	Fast `x`
1	39	39	38	27	15
2	126	126	60	109	36
3	552	552	240	200	88
4	3,108	3,108	634	1,063	290
5	21,840	21,840	3,213	3,001	1,039
6	181,566	181,566	12,721	19,598	4,762
7	1,724,131	1,724,131	76,473	81,290	23,142
8	18,307,585	18,307,585	407,639	584,226	133,554
9	213,940,146	213,940,146	2,809,853	3,254,363	812,730

Fig. 1. Number of logical inferences as counted by SWI-Prolog for our algorithms when querying generators with type patterns given in advance

`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.

8.1 Querying a 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 that match it, while generating terms and inferring their types.

The predicate `queryTypedTerm` finds closed terms of a given type of size exactly `L`.

```
queryTypedTerm(L,QueryType,Term):-
  genTypedTerm(L,Term,QueryType),
  boundTypeOf(Term,QueryType).
```

Similarly, the predicate `queryTypedTerm` finds closed terms of a given type of size `L` or less.

```
queryTypedTerms(L,QueryType,Term):-
  genTypedTerms(L,Term,QueryType),
  boundTypeOf(Term,QueryType).
```

Note that giving the query type ahead of executing `genTypedTerm` would unify with more general "false positives", as type checking, contrary to type synthesis, proceeds bottom-up. This justifies filtering out the false positives simply by testing with the deterministic predicate `boundTypeOf` at the end. Despite the extra call to `boundTypeOf`, the performance improvements are significant, as shown in Figure 1. The figure also shows that when the slow generate-and-test predicate `genTypedTerm1` is used, the result (in "logical-inferences-per-second") does not depend on the pattern, contrary to the fast `queryTypedTerm` that prunes mismatching types while inferring the type of the terms as it generates them.

Example 41
Terms of type x>x of size 4.

```
?- queryTypedTerm(3,(x>x),Term).
Term = a(l(v(0)), l(v(0))) ;
Term = l(a(l(v(0)), v(0))) ;
Term = l(a(l(v(1)), v(0))) .

?- queryTypedTerms(12,(x>x)>x,T).
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. In fact, it is known that no such terms exist, as the corresponding logic formula is not a tautology in minimal logic.

8.2 Same-type siblings

Given a closed well-typed lambda term, we can ask what other terms of the same size or smaller share the same type. This can be interesting for finding possibly alternative implementations of a given function or for generation of similar siblings in genetic programming.

The predicate typeSiblingOf lists all the terms of the same or smaller size having the same type as a given term.

```
typeSiblingOf(Term,Sibling):-
  dbTermSize(Term,L),
  boundTypeOf(Term,Type),
  queryTypedTerms(L,Type,Sibling).
```

Example 42
```
?- typeSiblingOf(l(l(a(v(0),a(v(0),v(1))))),T).
T = l(l(a(v(0), v(1)))) ; % <= smaller sibling
T = l(l(a(v(0), a(v(0), v(1))))) .
```

8.3 Discovering frequently occurring type patterns

The ability to run "relational queries" about terms and their types extends to compute interesting statistics, giving a glimpse at their distribution.

8.3.1 The "Popular" type patterns

As types can be seen as an approximation of their inhabitants, we expect them to be shared among distinct terms. As we can enumerate all the terms for small sizes and infer their types, we would like to know what are the most frequently occurring ones. This can be meaningful as a comparison base for types that are used in human-written programs of comparable size. In approaches like (Palka et al. 2011), where types are used to direct the generation of random terms, focusing on the most frequent types might help with generation of more realistic random tests.

Figure 2 describes counts for terms and their types for small sizes. It also shows the first two most frequent types with the count of terms they apply to.

Term size	Types	Terms	Ratio	1-st frequent	2-nd frequent
1	1	1	1.0	1: `x>x`	
2	1	2	0.5	2: `x>(x>x)`	
3	5	9	0.555	3: `x>(x>(x>x))`	3: `x>x`
4	16	40	0.4	14: `x>(x>x)`	4: `x>x>(x>(x>x))`
5	55	238	0.231	38: `x>(x>(x>x))`	31: `x>x`
6	235	1564	0.150	201: `x>(x>x)`	80: `x>x>(x>(x>x))`
7	1102	11807	0.093	732: `x>(x>(x>x))`	596: `x>x`
8	5757	98529	0.058	4632: `x>(x>x)`	2500: `x>x`
9	33251	904318	0.036	20214: `x>(x>(x>x))`	19855: `(x>x)>(x>x)`

Fig. 2. Counts for terms and types for sizes 1 to 9 and the first two most frequent types

Count	Type
23095	`x>(x>x)`
22811	`(x>x)>(x>x)`
22514	`x>x>(x>x)`
21686	`x>x`
18271	`x> ((x>x)>x)`
14159	`(x>x)>(x>(x>x))`
13254	`((x>x)>x)> ((x>x)>x)`
12921	`x> (x>x)>(x>x)`
11541	`(x>x)> ((x>x)>x)>x`
10919	`(x>(x>x))>(x>(x>x))`

Fig. 3. Most frequent types, out of a total of `33972` distinct types, of `1016508` terms up to size 9.

Figure 3 shows the "most popular types" for the about 1 million closed well-typed terms up to size 9 and the count of their inhabitants.

We can observe that, like in some human-written programs, functions representing binary operations of type `x>(x>x)` are the most popular. Ternary operations `x>(x>(x>x))` come third and unary operations `x>x` come fourth. Somewhat surprisingly, a higher order function type `(x>x)>(x>x)` applying a function to an argument to return a result comes second and multi-argument variants of it are also among the top 10.

8.3.2 Growth sequences of some popular types

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

Contrary to the total absence of the 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]. 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 [0, 0, 1, 1, 18, 52, 503, 2381, 19855, 125599]. The same turns out to be true also for `(x>x)>((x>x)>(x>x))`, giving [0, 0, 0, 0, 2, 6, 96, 505, 5287, 36769] and `((x>x)>(x>x)) > ((x>x)>(x>x))` giving [0, 0, 0, 0, 0, 6, 23, 432, 2450, 29924]. One might speculate that homotopy type theory (The Univalent Foundations Program 2013), 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!

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

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

The first 9 numbers counting closed simply-typed terms with at most one free variable (not yet in (Sloane 2014)), are [3, 10, 45, 256, 1688, 12671, 105743, 969032, 9639606].

Note that, as our generator performs the early pruning of untypable terms, rather than as a post-processing step, enumeration and counting of these terms happens in a few seconds.

8.4 Generating closed typable lambda terms by types

In (Palka et al. 2011) a "type-directed" mechanism for the generation of random terms is introduced, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some GHC bugs.

We can organize in a similar way the interface of our combined generator and type inferrer.

8.4.1 Generating type trees

The predicate `genType` generates binary trees representing simple types with a single base type `‘‘x’’`. As we represent types as binary trees with leaves `x` and internal nodes `>` we can reuse the predicate `genTree`.

```
genType --> genTree.
genTypes --> genTrees.
```

Like `genTree`, it provides two interfaces, for generating types of exactly size N or up to size N.

Next, we will combine this type generator with the generator that efficiently produces terms matching each type pattern.

8.4.2 Generating lambda terms by increasing type sizes

The predicate `genByType` first generates types (seen simply as binary trees) with `genType` and then uses the unification-based querying mechanism to generate all closed well-typed de Bruijn terms with fewer internal nodes then their binary tree type.

```
genByType(L,B,T):-
  genType(L,T),
  queryTypedTerms(L,T,B).
```

Example 43
Enumeration of closed simply-typed de Bruijn terms with types of size 3 and terms of a given type with at most 3 internal nodes.

```
?- genByType(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(a(v(0), v(1)))),
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, 6, 18, 84, 376, 2344, 15327]. Intuitively this means that despite of their growing sizes, types have an increasingly large number of inhabitants of sizes smaller than their size. This is somewhat contrary to what we see in human-written code, where types are almost always simpler and smaller than the programs inhabiting them.

8.4.3 Generation of random lambda terms

Generation of random lambda terms, resulting from the unranking of random integers of a give bit-size, is implemented by the predicate `ranTerm/3`, that applies the predicate `Filter` repeatedly until a term is found for which the predicate `Filter` holds.

```
ranTerm(Filter,Bits,T):-X is 2^Bits,N is X+random(X),M is N+X,
  between(N,M,I),
   unrankTerm(I,T),call(Filter,T),
  !.
```

Random open terms are generated by `ranOpen/2`, random closed terms by the predicate `ranClosed`, random typable term by `ranTyped` and closed typable terms by `closedTypable/2`.

```
ranOpen(Bits,T):-ranTerm(=(_),Bits,T).

ranClosed(Bits,T):-ranTerm(isClosedC,Bits,T).

ranTyped(Bits,T):-ranTerm(closedTypable,Bits,T).

closedTypable(T):-isClosedC(T),typable(T).
```

Open terms based on unranking random numbers of 3000 bits of size above 1000, closed terms of size above 55 for 150 bits and closed typable terms of size above 13 for 30 bits can be generated within a few seconds. The limited scalability for closed and well-typed terms is a consequence of their low asymptotic density, as shown in (David et al. 2009; Grygiel and Lescanne 2013). We refer to (Grygiel and Lescanne 2013) for algorithms supporting random generation of large lambda terms.

Example 44
illustrates generation of some closed and well-typed terms in compressed de Bruijn form.

```
?- ranClosed(10,T).
T = a(1, a(0, v(0, 0), v(0, 0)), a(0, a(0, v(0, 0), v(0, 0)), v(1, 0))).

?- ranTyped(20,T).
T = a(3, v(3, 1), v(2, 0)).
```

8.5 *Estimating the proportion of well-typed SK-combinator trees*

Given the low density of closed well-typed lambda terms, an interesting question arises at this point: *what proportion of SK-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 (Bodini et al. 2011; David et al. 2010; Grygiel et al. 2013; Grygiel and Lescanne 2013), it is considered an open problem for well-typed terms. We will limit ourselves here to empirically estimate it, as it is done in (Grygiel and Lescanne 2013) for general lambda terms, where experiments indicate the extreme sparsity for very large terms.

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

Example 45
Types inferred for terms with 2 internal nodes.

```
?- genSK(1,X),simpleTypeOf(X,T).
X = k*k,
T = (x> (x> (x>x))) ;
X = k*s,
T = (x> ((x> (x>x))> ((x>x)> (x>x)))) ;
X = s*k,
T = ((x>x)> (x>x)) ;
X = s*s,
T = (((x> (x>x))> (x>x))> ((x> (x>x))> (x>x))) .
```

Similarly, we can use it also to enumerate untypable terms.

Example 46
The smallest two untypable SK-expressions.

```
?- genSKs(2,X), \+typableSK(X).
X = s*s*k ;
X = s*s*s .
```

We can implement a generator for well-typed SK-trees, to be used to compute the ratio between the number of well-typed SK-trees and the total number of SK-trees of size *n*, as well as one for the untypable SK-trees.

```
genTypedSK(L,X,T):-genSK(L,X),simpleTypeOf(X,T).

genUntypableSK(L,X):-genSK(L,X),\+skTypeOf(X,_).
```

To compute the proportion of well-typed terms among terms of a given size we will also need to count the number of SK-trees with *n* internal nodes.

Term size	Well-typed	Total	Ratio
0	2	2	1
1	4	4	1
2	14	16	0.875
3	67	80	0.8375
4	337	448	0.752
5	1867	2688	0.694
6	10699	16896	0.633
7	63567	109824	0.578
8	387080	732160	0.528
9	2401657	4978688	0.482

Fig. 4. Proportion of well-typed SK-combinator terms

Proposition 14
There are $2^{n+1}C_n$ SK-trees with n nodes, where C_n is the n-th Catalan number.

Proof
If follows from the fact that C_n counts the number of binary trees with n internal nodes, each of which has $n+1$ leaves, each of which can be either S or K. □

The predicate `cat/2` computes the nth-Catalan number efficiently using the recurrence $C_0 = 1, C_n = \frac{2(2n-1)}{n+1}C_{n-1}$ (Stanley 1986).

```
cat(0,1).
cat(N,R):-N>0,
  PN is N-1,
  cat(PN,R1),
  R is 2*(2*N-1)*R1//(N+1).
```

Figure 4 shows the counts for well-typed SK-combinator expressions and their ratio to the total number of SK-trees of given size.

Somewhat surprisingly, a large proportion of well-typed SK-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.

8.5.1 Generating typed SK-combinator trees by types

In (Palka et al. 2011) generation of random terms is guided by their types, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some GHC bugs.

8.5.2 Generating SK-trees by increasing type sizes

The predicate `genByType` first generates simple types with `genType` and then uses the unification-based querying mechanism to generate, for each of the types, its inhabitant SK-trees with fewer internal nodes then their their type.

```
genByTypeSK(L,X,T):-
  genType(L,T),
  genSKs(L,X),
  simpleTypeOf(X,T).
```

The number of such terms grows quite fast, the sequence describing the number of terms with sizes smaller or equal than the size of their types up to 7 is `0, 3, 29, 250, 3381, 48968, 809092`.

Example 47
Enumeration of closed simply-typed de SK combinator trees with types of size 2 and less then 2 internal nodes.

```
?- genByTypeSK(2,B,T).
B = k,
T = (x> (x>x)) ;
B = k*k*k,
T = (x> (x>x)) ;
B = k*k*s,
T = (x> (x>x)) .
```

8.6 The well-typed frontier of an untypable SK-expression

As in the case of lambda terms, untypable SK-expressions become the majority as soon as the size of the expression reaches some threshold, 9 in this case. This actually turns out to be a good thing, from a programmer's perspective: types help with bug-avoidance partly because being "accidentally well-typed" becomes a low probability event for larger programs.

Driven by a curiosity somewhat similar to that about distribution and density properties of prime numbers, one would want to decompose an untypable SK-expression into a set of maximal typable ones. This makes sense, as, contrary to lambda expressions, SK-trees are uniquely built with application operations as their internal nodes.

Definition 2
We call *well-typed frontier* of a combinator tree set of its maximal well-typed subtrees.

Note also, that contrary to general lambda terms, SK-terms are *hereditarily closed* i.e., every subterm of a SK-expression is closed. Consequently, the well-typed frontier is made of closed terms.

Definition 3
We call *typeless trunk* of a combinator tree the subtree starting from the root from which the members of its well-typed frontier have been removed and replaced with logic variables.

8.6.1 Computing the well-typed frontier

The well-typed frontier of a combinator tree and its typeless trunk are computed together by the predicate The predicate `wellTypedFrontier` . It actually proceeds by separating the trunk from the frontier and marking with fresh logic variables the replaced subtrees. These variables are added as left sides of equations with the frontiers as their right sides.

```
wellTypedFrontier(Term,Trunk,FrontierEqs):-
  wtf(Term, Trunk,FrontierEqs,[]).

wtf(Term,X)-->{typableSK(Term)},!,[X=Term].
wtf(A*B,X*Y)-->wtf(A,X),wtf(B,Y).
```

Example 48
Well-typed frontier and *typeless trunk* of the untypable term *SSI*(*SSI*) (with *I* represented as *SKK*).

```
?- wellTypedFrontier(s*s*(s*k*k)*(s*s*(s*k*k)),
                     Trunk,FrontierEqs).
Trunk = A*B* (C*D),
FrontierEqs = [A=s*s, B=s*k*k, C=s*s, D=s*k*k].
```

The list-of-equations representation of the frontier allows to easily reverse their separation from the trunk by a unification based “grafting” operation.

The predicate `fuseFrontier` implements this reversing process while the predicate `extractFrontier` extracts from the frontier-equations the components of the frontier without the corresponding variables marking their location in the trunk.

```
fuseFrontier(FrontierEqs):-maplist(call,FrontierEqs).

extractFrontier(FrontierEqs,Frontier):-
  maplist(arg(2),FrontierEqs,Frontier).
```

Example 49
Extracting and grafting back the well-typed frontier to the typeless trunk.

```
?- wellTypedFrontier(s*s*(s*k*k)*(s*s*(s*k*k)),
       Trunk,FrontierEqs),
   extractFrontier(FrontierEqs,Frontier),
   fuseFrontier(FrontierEqs).
Trunk = s*s* (s*k*k)* (s*s* (s*k*k)),
FrontierEqs = [s*s=s*s, s*k*k=s*k*k,
               s*s=s*s, s*k*k=s*k*k],
Frontier = [s*s, s*k*k, s*s, s*k*k] .
```

Note that after grafting back the frontier, the trunk becomes equal to the term that we have started with.

8.6.2 A comparison of the sizes of the well-typed frontier and the typeless trunk

An interesting question arises at this point: *how do the sizes of the frontier and the trunk compare*?

Figure 5 compares the average sizes of the frontier and the trunk for terms up to size 8. This indicates that, while the size of the frontier dominates for small terms, it decreases progressively. This leaves the following open problem: *does the average ratio of the frontier and the trunk converge to a limit as the size of the terms increases*? More empirical information on this can be obtained by studying what happens for randomly generated large SK-trees.

Term size	Avg. Trunk-size	Avg. Frontier-size	% Trunk	% Frontier
1	0	1	0	100
2	0.13	1.88	6.25	93.75
3	0.26	2.74	8.75	91.25
4	0.47	3.53	11.77	88.23
5	0.71	4.29	14.11	85.89
6	0.97	5.03	16.24	83.76
7	1.27	5.73	18.11	81.89
8	1.58	6.42	19.76	80.24

Fig. 5. Comparison of sizes of the typeless trunk and the well-typed frontier of SK-terms, by size.

8.6.3 Simplification as normalization of the well-typed frontier

Given that well-typed terms are strongly normalizing, we can simplify an untypable term by normalizing the members of its frontier, for which we are sure that `evalSK` terminates. Once evaluated, we can graft back the results to the typeless trunk, as implemented the predicate `simplifySK`.

```
simplifySK(Term,Trunk):-
  wellTypedFrontier(Term,Trunk,FrontierEqs),
  extractFrontier(FrontierEqs,Fs),
  maplist(evalSK,Fs,NormalizedFs),
  maplist(arg(1),FrontierEqs,Vs),
  Vs=NormalizedFs.
```

The following question arises at this point: *are there terms that are not normalizable that can be simplified by extracting and simplifying their well-typed frontier and then grafting it back*? Combinatorial search, using the `genSK` predicate finds them starting at size `8`.

Example 50
Simplifying some untypable terms for which normalization is non-terminating.

```
?- Term= s*s*s* (s*s)*s* (k*s*k),
         simplifySK(Term,Trunk).
Term = s*s*s* (s*s)*s* (k*s*k),
Trunk = s*s*s* (s*s)*s*s.

?- Term= k* (s*s*s* (s*s)*s* (k*s*k)),
   simplifySK(Term,Trunk).
Term = k* (s*s*s* (s*s)*s* (k*s*k)),
Trunk = k* (s*s*s* (s*s)*s*s).
```

Note that, as expected, while simplification does not bring termination to the normalization predicate `evalSK/2`, it shows the existence of non-terminating computations for which a terminating simplification is possible.

8.6.4 Discussion

While the well-typed (and closed) frontier does not make sense for general lambda terms where closed terms may have open subterms, it makes sense for other combinator or supercombinator languages (Peyton Jones 1987), some with practical uses in the compilation of functional languages.

Among the open problems we leave for future research, is to find out if concepts like the well-typed frontier of a richer combinator-language can be used for suggesting a fix to a program in a typed functional programming language, or to produce more precise error messages in case of type errors. For instance, it would be interesting to know if a minimal well-typed alternative can be be inferred and suggested to the programmer on a type error.

If one replaces the `unify_with_occurs_check` in predicate `skTypeOf` with the cyclic term unification (that most modern Prologs use by default), one can observe that every combinator expression passes the test! The predicate `uselessTypeOf` implements this variation.

```
uselessTypeOf(k,(A>(_B>A))).
uselessTypeOf(s,(((A>(B>C))> ((A>B)>(A>C))))).
uselessTypeOf((A*B),Y):-
  uselessTypeOf(A,(X>Y)),
  uselessTypeOf(B,X).
```

After defining the predicates `notReallyTypable` and `sameAsAny`

```
notReallyTypable(X):-uselessTypeOf(X,_).

sameAsAny(L,M):-genSK(L,M),notReallyTypable(M).
```

one can notice the identical behavior of `sameAsAny` and `genSK`, meaning that failing the occurs-check is the exclusive reason of failure to infer a type. This happens in the presence of a unique basic type "x". However, in the case of a more realistic type system with multiple basic types like `Boolean, Int, String` etc., the failure of type inference could also be a consequence of mismatched basic types. Knowing more about these two reasons for failure might suggest weakened type systems where some limited form of circularity is acceptable, provided that no basic type mismatches occur. While strong normalization would be sacrificed if such circular types were accepted, one might note that this is already the case in practical languages, where fixpoint operators or recursive data type definitions are allowed.

8.7 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 (Bodini et al. 2011; Grygiel et al. 2013; Grygiel and Lescanne 2013), it is considered an open problem for well-typed terms. We will limit ourselves here to empirically estimate it, as it is done in (Grygiel and Lescanne 2013) 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 51
Types inferred for terms with 2 internal nodes.

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

Fig. 6. Proportion of well-typed X-combinator terms

```
?- 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))
```

Figure 6 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 (Sloane 2014)), as binary trees are a member of the Catalan family of combinatorial objects (Stanley 1986).

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.

8.8 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 52
Terms of type `x>x` of size 4.

```
?- genTypedB(4,Term,(x>x)).
Term = a(l(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 (The Univalent Foundations Program 2013), 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!

8.9 Iterated types

Example 53
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 54
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

Fig. 7. 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 7 shows the average number of steps until a un-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.

8.10 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 55
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.

8.11 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 56
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),dbTermSize(B,S1),b2b(B,BB),dbTermSize(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 57
Transformations of the X-combinator via `b2b`, `evalDeBruijn`, `boundTypeOf` and `n`.

```
?- xB(X),b2b(X,XX),evalDeBruijn(XX,R),boundTypeOf(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.

8.12 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):-boundTypeOf(B,_),evalDeBruijn(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 (Katok and Hasselblatt 1995) 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 8 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.

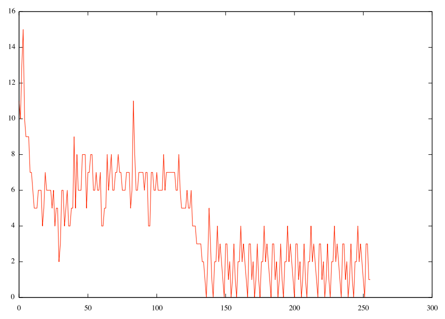

Fig. 8. Term sizes in the orbit starting from the X-combinator

Figure 9 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.

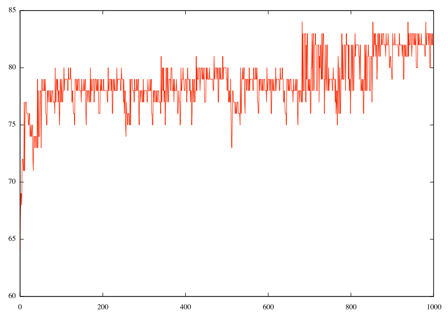

Fig. 9. Term sizes in the orbit starting from the term ω

8.13 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 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.

8.13.1 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 (Stanley 1986). 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 58
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.

8.13.2 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 (Lamping 1990) 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.

9 Related work

The classic reference for lambda calculus is (Barendregt 1984). Various instances of typed lambda calculi are overviewed in (Barendregt 1991).

Originally introduced in (de Bruijn 1972), the de Bruijn notation makes terms equivalent up to α-conversion and facilitates their normalization (Kamareddine 2001). 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 (Peyton Jones 1987). The compressed de Bruijn representation of lambda terms proposed in this paper (and (Tarau 2015c)) is novel, to our best knowledge.

Lambda terms of bounded unary height are introduced in (Bodini et al. 2011). John Tromp's binary lambda calculus is only described through online code and the Wikipedia entry at (Wikipedia 2015).

Generators for closed and well-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in (Grygiel and Lescanne 2013), derived from combinatorial recurrences. However, they are significantly more complex than the ones described here in Prolog. On the other hand, we have not found in the literature generators for linear, linear affine terms and lambda terms of bounded unary height. Normalization of lambda terms and its confluence properties are described in (Barendregt 1984) and (Kamareddine 2001) with functional programming algorithms given in (Sestoft 2002) and HOAS-based evaluation first described in (Pfenning and Elliot 1988).

In a logic programming context, unification of simply typed lambda terms has been used in as the foundation of the programming language λProlog (Miller 1991; Nadathur and Mitchell 1999) and applied to higher order logic programming (Miller and Nadathur 2012).

Various instances of typed lambda calculi are overviewed in (Barendregt 1991). Combinators originate in Moses Schönfinkel's 1924 paper, and independently, in Haskell Curry's work in 1927. A modern introduction to combinators and their relation to lambda calculus is (Hindley and Seldin 2008) and a first application of an extended set of combinators in the implementation of functional programming languages is (Turner 1979).

Combinatorics of lambda terms, including enumeration, random generation and asymptotic behavior has seen an increased interest recently (see for instance (Bodini et al. 2011; Grygiel and Lescanne 2013; David et al. 2010; Grygiel et al. 2013)), 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. Distribution and density properties of random lambda terms are described in (David et al. 2009). In (Palka et al. 2011; Fetscher et al. 2015), 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 4.13, by taking advantage of Prolog's ability to backtrack over possible terms, while filtering against unification with a specific pattern. In (Tarau 2015b) generation algorithms for several sub-families of lambda terms are given as well as a compressed deBruijn representation is introduced. In (Tarau 2015a) Rosser's X-combinator trees (Fokker 1992) are used as a uniform representation via bijections top lambda terms in de Bruijn notation, types and a tree-based number representation.

Of particular interest are the results of (Grygiel and Lescanne 2013) 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 (Grygiel and Lescanne 2013) 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.

One-point combinator bases, together with a derivation of the X-combinator are described in (Fokker 1992). In (Goldberg 2004) the existence of a countable number of 1-point bases is proven. While esoteric programming languages exist based on similar 1-point bases (Stay 2005), 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.

Ranking and unranking algorithms for several classes of lambda terms are also described in (Grygiel and Lescanne 2013),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 (Vytiniotis and Kennedy 2010) as well as for Gödel-numbering schemes (Hartmanis and Baker 1974) of theoretical relevance. In fact, ranking functions for sequences can be traced back to Gödel numberings (Gödel 1931) associated to formulas.

While Gödel-numbering schemes for lambda terms have been studied in several theoretical papers on computability, we are not aware of any size proportionate bijective encoding as the one described in this paper.

Injective Gödel-numbering schemes for lambda terms in de Bruijn notation have been described in the context of binary lambda calculus (Tromp 2014) and as a mechanism to encode datatypes in (Vytiniotis and Kennedy 2010; Kobayashi et al. 2012). Both these use prefix-free codes, ensuring unique decoding. A bijective Gödel-numbering scheme is associated to the esoteric programming language Jot (Stay 2005), 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 (Tarau 2014d; Tarau 2014c), 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., (Stanley 1986; Knuth 2006)) 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 (Tarau 2014b). Note also that while (Tarau 2014d; Tarau 2014c) 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 (The Univalent Foundations Program 2013) 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 (Kamareddine 2001; Sestoft 2002; Barendregt 1984), although we are not aware, for instance, of Prolog implementations of type inference working directly on de Bruijn terms or X-combinator trees. In (Tarau 2015b) 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 (Tarau 2015b) using a de Bruijn representation of lambda terms. We refer to (Tarau 2015b) 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.

10 Conclusions

We have described compact (and arguably elegant) combinatorial generation algorithms for several important families of lambda terms. Besides the newly introduced a compressed form of de Bruijn terms we have used ordinary de Bruijn terms as well as a canonical representation of lambda terms relying on Prolog's logic variables. In each case, we have selected the representation that was more appropriate for tasks like combinatorial generation, type inference or normalization. We have switched representation as needed, though bijective transformers working in time proportional to the size of the terms. Our combinatorial generation algorithms match the corresponding sequence of counts by size, given in (Sloane 2014) as an empirical validation of their correctness.

We have described Prolog-based term and type generation and as well as type-inference algorithms for de Bruijn terms. Among the possible applications of our techniques we mention compilation and test generation for lambda-calculus based languages and proof assistants. Our merged generation and type inference in an algorithm showed a mechanism to build "customized closed terms of a given type". This "relational view" of terms and their types has enabled the discovery of interesting patterns about the type expressions occurring in well-typed programs. We have uncovered the most "popular" types that govern function applications among a about a million small-sized lambda terms.

We have also observed some interesting phenomena about frequently occurring types, that seem to be similar to those in human-written programs and we have computed growth sequences for the number of inhabitants of some "popular" types, for which we have not found any study in the literature.

A significant contribution of this paper is the size-proportionate ranking/unranking algorithm for lambda terms and the compressed de Bruijn representation that facilitated it. The ability to encode lambda terms bijectively can be used as a "serialization" mechanism in functional programming languages and proof assistants using them as an intermediate language.

We have selected the minimalist pure combinator language built from applications of combinators S and K to explore aspects of their generation and type inference algorithms. While a draconian simplification of real-life programming languages, this well-known and well-researched subset of lambda calculus has revealed some interesting new facts about the density and distribution of their types. The new concepts of *well-typed frontier* and *typeless trunk* of an untypable term can be generalized to realistic combinator and supercombinator-based intermediate languages used by compilers for functional languages and proof assistants. As they give precise hints about the points where type inference failed, they are likely to be useful for debugging programs and give more meaningful compile-time error messages. This also results in an ability to extend (sure) termination beyond simply-typed terms, by evaluating and then grafting back their well-typed frontier. By sharing the representation of the Turing-complete language of X-combinator expressions, natural numbers, lambda terms and their types, interesting synergies became available.

The paper has introduced a number of algorithms that, at our best knowledge, are novel, at least in terms of their logic programming implementation, among which we mention the type inference for de Bruijn terms using unification with occurs-check in subsection 6.2.1 and the integrated generation and type inference algorithm for closed simply typed de Bruijn terms in section 4.13. Besides the ability to efficiently query for inhabitants of specific types, our algorithms also support a from of "query-by-example" mechanism, for finding (possibly smaller) terms inhabiting the same type as the query term. 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:

- a compressed representation of de Bruijn terms in subsection 2.5

- X-combinator trees playing the role of both natural numbers and types in subsections 6.3.1, 6.5.1 and 7.3.1
- a bijection between natural numbers and binary trees (predicates `t/2` and `n/2` in subsection 7.3.1) that is works consistently with their isomorphic arithmetic operations
- a concept of "iterated types" in subsection 8.9
- two size-inflating injective functions from terms to terms in subsection 8.11
- a multi-operation dynamic system combining normalization and arithmetic operations in subsection 8.12

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

- integrated generation and type inference algorithm for closed simply-typed de Bruijn terms in subsection 4.13
- successor and predecessor and arithmetic operations on binary trees in subsection 7.3.2
- ranking and unranking de Bruijn terms to/from binary-tree represented natural numbers in subsection 7.4.1
- direct type inference for X-combinator trees in subsection 6.5.2

While a non-strict functional language like Haskell could have been used for deriving similar algorithms, the synergy between Prolog's non-determinism, DCG transformation and the availability of unification with occurs-check made the code embedded in the paper significantly simpler and arguably clearer.

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. Future work will also focus on studying how our results extend to other families of combinators and supercombinators that occur in practical languages as well as on random SK-tree generation e.g.., by extending Rémy's algorithm (Rémy 1985) from binary trees to SK-combinator trees. This would allow fast generation of very large SK-combinator expressions that could give better empirical estimates on the asymptotic behavior of the concepts introduced in this paper and their properties. Also, as a step toward more practical uses, lifting the concept of well-typed frontier to general lambda terms (which are not hereditarily closed) seems possible by defining the frontier as being a sequence of maximal well-typed closed lambda terms.

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

This research has been supported by NSF grant 1423324.

References

BARENDREGT, H. P. 1984. *The Lambda Calculus Its Syntax and Semantics*, Revised ed. Vol. 103. North Holland.

BARENDREGT, H. P. 1991. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Vol. 2. Oxford University Press.

BODINI, O., GARDY, D., AND GITTENBERGER, B. 2011. Lambda-terms of bounded unary height. In *ANALCO*. SIAM, 23–32.

CEGIELSKI, P. AND RICHARD, D. 1999. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science 222,* 1-2, 55–75.

DAVID, R., GRYGIEL, K., KOZIK, J., RAFFALLI, C., THEYSSIER, G., AND ZAIONC, M. 2010. Asymptotically almost all λ-terms are strongly normalizing. *Preprint: arXiv: math. LO/0903.5505 v3*.

DAVID, R., RAFFALLI, C., THEYSSIER, G., GRYGIEL, K., KOZIK, J., AND ZAIONC, M. 2009. Some properties of random lambda terms. *Logical Methods in Computer Science 9,* 1.

DE BRUIJN, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae 34*, 381–392.

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

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

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

GOLDBERG, M. 2004. A construction of one-point bases in extended lambda calculi. *Inf. Process. Lett. 89,* 6, 281–286.

GRYGIEL, K., IDZIAK, P. M., AND ZAIONC, M. 2013. How big is BCI fragment of BCK logic. *J. Log. Comput. 23,* 3, 673–691.

GRYGIEL, K. AND LESCANNE, P. 2013. Counting and generating lambda terms. *J. Funct. Program. 23,* 5, 594–628.

HARTMANIS, J. AND BAKER, T. P. 1974. On Simple Goedel Numberings and Translations. In *ICALP* (2002-02-01), J. Loeckx, Ed. Lecture Notes in Computer Science, vol. 14. Springer, Berlin Heidelberg, 301–316.

HINDLEY, J. R. AND SELDIN, J. P. 2008. *Lambda-calculus and combinators: an introduction*. Vol. 13. Cambridge University Press Cambridge.

KAMAREDDINE, F. 2001. Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems. *Journal of Logic and Computation 11,* 3, 363–394.

KATOK, A. AND HASSELBLATT, B. 1995. *Introduction to the modern theory of dynamical systems*. Ency. of Math. and its App., vol. 54. Cambridge Univ. Press.

KNUTH, D. E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional.

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

KOBAYASHI, N., MATSUDA, K., AND SHINOHARA, A. 2012. Functional Programs as Compressed Data. *ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. ACM Press.

KREHER, D. L. AND STINSON, D. 1999. *Combinatorial Algorithms: Generation, Enumeration, and Search*. The CRC Press Series on Discrete Mathematics and its Applications. CRC PressINC.

LAMPING, J. 1990. 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*. 16–30.

LEHMER, D. H. 1964. The machine tools of combinatorics. In *Applied combinatorial mathematics*. Wiley, New York, 5–30.

MCBRIDE, C. 2010. I am not a number, I am a classy hack. *Blog entry:* `http://mazzo.li/epilogue/index.html%3Fp=773.html`.

MILLER, D. 1991. Unification of simply typed lambda-terms as logic programming. In *Proc. Int. Conference on Logic Programming (Paris)*. MIT Press, 255–269.

MILLER, D. AND NADATHUR, G. 2012. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA.

NADATHUR, G. AND MITCHELL, D. 1999. System Description: Teyjus A Compiler and Abstract Machine Based Implementation of λProlog. In *Automated Deduction CADE-16*. Lecture Notes in Computer Science, vol. 1632. Springer Berlin Heidelberg, 287–291.

PALKA, M. H., CLAESSEN, K., RUSSO, A., AND HUGHES, J. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. AST'11. ACM, New York, NY, USA, 91–97.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., NJ, USA.

PFENNING, F. AND ELLIOT, C. 1988. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. ACM, New York, NY, USA, 199–208.

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

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

SLOANE, N. J. A. 2014. The On-Line Encyclopedia of Integer Sequences. Published electronically at https://oeis.org/.

STANLEY, R. P. 1986. *Enumerative Combinatorics*. Wadsworth Publ. Co., Belmont, CA, USA.

STAY, M. 2005. Very simple chaitin machines for concrete AIT. *CoRR abs/cs/0508056*.

TARAU, P. 2009. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*. ACM, Coimbra, Portugal, 171–182.

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

TARAU, P. 2014a. 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*. ACM, New York, NY, USA.

TARAU, P. 2014b. A Generic Numbering System based on Catalan Families of Combinatorial Objects. *CoRR abs/1406.1796*.

TARAU, P. 2014c. Arithmetic and boolean operations on recursively run-length compressed natural numbers. *Scientific Annals of Computer Science 24,* 2, 287–323.

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

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

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

TARAU, P. 2015c. Ranking/Unranking of Lambda Terms with Compressed de Bruijn Indices. In *Proceedings of the 8th Conference on Intelligent Computer Mathematics*, M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, Eds. Springer, LNAI 9150, Washington, D.C., USA, 118–133.

THE UNIVALENT FOUNDATIONS PROGRAM. 2013. *Homotopy Type Theory*. Institute of Advanced Studies, Princeton. `http://homotopytypetheory.org/2013/06/20/the-hott-book/`.

TROMP, J. 2014. Binary lambda calculus and combinatory logic.

TURNER, D. A. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience 9,* 1, 31–49.

VYTINIOTIS, D. AND KENNEDY, A. 2010. Functional Pearl: Every Bit Counts. *ICFP 2010 : The 15th ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

WIKIPEDIA. 2015. Binary lambda calculus — wikipedia, the free encyclopedia. [Online; accessed 20-February-2015].

Appendix

Helper predicates for ranking and unranking balanced parentheses expressions

The predicate `binDif` computes the difference of two binomials.

```
binDif(N,X,Y,R):- N1 is 2*N-X,R1 is N - (X + Y) // 2, R2 is R1-1,
  binomial(N1,R1,B1),binomial(N1,R2,B2),R is B1-B2.
```

The predicate `localRank` computes, by binary search the rank of sequences of a given length.

```
localRank(N,As,NewLo):- X is 1, Y is 0, Lo is 0,
  binDif(N,0,0,Hi0),Hi is Hi0-1,
  localRankLoop(As,N,X,Y,Lo,Hi,NewLo,_NewHi).
```

After finding the appropriate range containing the rank with `binDif`, we delegate the work to the predicate `localRankLoop`.

```
localRankLoop(As,N,X,Y,Lo,Hi,FinalLo,FinalHi):-N2 is 2*N,X< N2,!,
  PY is Y-1, SY is Y+1, nth0(X,As,A),
  (0=:=A-> binDif(N,X,PY,Hi1),
     NewHi is Hi-Hi1, NewLo is Lo, NewY is SY
   ; binDif(N,X,SY,Lo1),
     NewLo is Lo+Lo1, NewHi is Hi, NewY is PY
  ), NewX is X+1,
  localRankLoop(As,N,NewX,NewY,NewLo,NewHi,FinalLo,FinalHi).
localRankLoop(_As,_N,_X,_Y,Lo,Hi,Lo,Hi).
```

```
rankLoop(I,S,FinalS):-I>=0,!,cat(I,C),NewS is S+C, PI is I-1,
  rankLoop(PI,NewS,FinalS).
rankLoop(_,S,S).
```

Unranking works in a similar way. The predicate `localUnrank` builds a sequence of balanced parentheses by doing binary search to locate the sequence in the enumeration of sequences of a given length.

```
localUnrank(N,R,As):-Y is 0,Lo is 0,binDif(N,0,0,Hi0),Hi is Hi0-1, X is 1,
  localUnrankLoop(X,Y,N,R,Lo,Hi,As).

localUnrankLoop(X,Y,N,R,Lo,Hi,As):-N2 is 2*N,X=<N2,!,
   PY is Y-1, SY is Y+1,
   binDif(N,X,SY,K), LK is Lo+K,
   ( R<LK -> NewHi is LK-1, NewLo is Lo, NewY is SY, Digit=0
   ; NewLo is LK, NewHi is Hi, NewY is PY, Digit=1
   ),nth0(X,As,Digit),NewX is X+1,
   localUnrankLoop(NewX,NewY,N,R,NewLo,NewHi,As).
localUnrankLoop(_X,_Y,_N,_R,_Lo,_Hi,_As).
```

```
unrankLoop(R,S,I,FinalS,FinalI):-cat(I,C),NewS is S+C, NewS=<R,
   !,NewI is I+1,
   unrankLoop(R,NewS,NewI,FinalS,FinalI).
unrankLoop(_,S,I,S,I).
```

The bijection between finite lists and sets

The bijection `list2set` together with its inverse `set2list` are defined as follows:

```
list2set(Ns,Xs) :- list2set(Ns,-1,Xs).

list2set([],_,[]).
list2set([N|Ns],Y,[X|Xs]) :-
  X is (N+Y)+1,
  list2set(Ns,X,Xs).

set2list(Xs,Ns) :- set2list(Xs,-1,Ns).

set2list([],_,[]).
set2list([X|Xs],Y,[N|Ns]) :-
  N is (X-Y)-1,
  set2list(Xs,X,Ns).
```

The following examples illustrate this bijection:

```
?- list2set([2,0,1,4],Set),set2list(Set,List).
Set = [2, 3, 5, 10],
List = [2, 0, 1, 4].
```

As a side note, this bijection is mentioned in (Knuth 2005) with indications that it might even go back to the early days of the theory of recursive functions.

Binomial Coefficients, efficiently

Binomial coefficients are given by the formula $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)...(n-(k-1))}{k!}$. By performing divisions as early as possible to avoid generating excessively large intermediate results, one can derive the `binomialLoop` tail-recursive predicate:

```
binomialLoop(_,K,I,P,R) :- I>=K, !, R=P.
binomialLoop(N,K,I,P,R) :- I1 is I+1, P1 is ((N-I)*P) // I1,
   binomialLoop(N,K,I1,P1,R).
```

The predicate `binomial(N,K,R)` computes $\binom{N}{K}$ and unifies the result with `R`.

```
binomial(_N,K,R):- K<0,!,R=0.
binomial(N,K,R) :- K>N,!, R=0.
binomial(N,K,R) :- K1 is N-K, K>K1, !, binomialLoop(N,K1,0,1,R).
binomial(N,K,R) :- binomialLoop(N,K,0,1,R).
```