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

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

CLA’2016

Research supported by NSF grant 1423324
An Overview of our Playground

- 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
- transformers to/from a compressed de Bruijn form
- an algorithm combining term generation and type inference
- a normal order reduction algorithm for lambda terms relying on their de Bruijn representation
- generators and evaluation algorithms for SK and (Rosser’s) X-combinator expressions
- type inference algorithms for SK and X-combinator expressions
- a discussion of what happens when expressions and types sharing the same binary tree representation
size-proportionate bijective encodings of lambda terms and combinators
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 size-proportionate ranking and unranking algorithms for lambda terms and their inferred types
an interpretation of X-combinator trees as natural numbers on which it defines arithmetic operations
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
• some uses of our combined term generation and type inference algorithm to discover frequently occurring type patterns
• a type-directed algorithm for the generation of closed typable lambda terms
• the well-typed frontier of an untypable SK-expression
• its application a (partial) normalization-based simplification algorithm that terminates on all SK-expressions

this talk:

a few samples of the playground at work – after a short introduction to Prolog
Horn Clause Prolog in three slides
Prolog: Unification, backtracking, clause selection

?- X=a, Y=X.  % variables uppercase, constants lower
X = Y, Y = a.

?- X=a, X=b.
false.

?- f(X,b)=f(a,Y).  % compound terms unify recursively
X = a, Y = b.

% clauses

a(1). a(2). a(3).  % facts for a/1
b(2). b(3). b(4).  % facts for b/1

c(0).
c(X):-a(X),b(X).  % a/1 and b/1 must agree on X

?-c(R).  % the goal at the Prolog REPL
R=0; R=2; R=3.  % the stream of answers
Prolog: Definite Clause Grammars

Prolog’s DCG preprocessor transforms a clause defined with “-->” like
\[ a_0 --> a_1, a_2, \ldots, a_n. \]
into a clause where predicates have two extra arguments expressing a chain of state changes as in
\[ a_0(S_0, S_n) :\neg a_1(S_0, S_1), a_2(S_1, S_2), \ldots, a_n(S_{n-1}, S_n). \]

- work like “non-directional” attribute grammars/rewriting systems
- they can used to compose relations (functions in particular)
- with compound terms (e.g. lists) as arguments they form a Turing-complete embedded language

\[ f --> g, h. \]
\[ f(In, Out) :\neg f(In, Temp), g(Temp, Out). \]
Prolog: the two-clause meta-interpreter

The meta-interpreter \texttt{metaint/1} uses a (difference)-list view of prolog clauses.

\begin{verbatim}
metaint([]). % no more goals left, succeed
metaint([G|Gs]):- % unify the first goal with the head of a clause
    cls([G|Bs],Gs), % build a new list of goals from the body of the
    % clause extended with the remaining goals as tail
    metaint(Bs). % interpret the extended body

% clauses are represented as facts of the form \texttt{cls/2}
% the first argument representing the head of the clause + a list of body goals
% clauses are terminated with a variable, also the second argument of \texttt{cls/2}.

cls([ add(0,X,X) |Tail],Tail).
cls([ add(s(X),Y,s(Z)), add(X,Y,Z) |Tail],Tail).
cls([ goal(R), add(s(s(0)),s(s(0)),R) |Tail],Tail).
\end{verbatim}

?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
Lambda terms in Prolog: canonical, de Bruijn, compressed
Lambda Terms in Prolog

- 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.
- The term $\lambda a.((\lambda b.(a(b\ b)))(\lambda c.(a(c\ c))))$ is represented as $l(A, a(l(B, a(A, a(B, B))), l(C, a(A, a(C, C))))).
- "Canonical" names - each lambda binder is mapped to a distinct logic variable.
- Scoping of logic variables is "global" to a clause - they are all universally quantified.
De Bruijn Indices

- *De Bruijn Indices* provide a name-free representation of lambda terms
- terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation
  - variables following lambda abstractions are omitted
  - their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* on the way up to the root of the term
- term with canonical names: \( l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C))))) \) ⇒
- de Bruijn term: \( l(a(l(a(v(1),a(v(0),v(0)))))),l(a(v(1),a(v(0),v(0)))))) \)
- note: we start counting up from 0
- closed terms: every variable occurrence belongs to a binder
- open terms: otherwise
Should we compress $\lambda$-terms in de Bruijn notation?

Figure: Random $\lambda$-terms can have long necks: $l(l(l(l(l(\ldots a(\ldots$

Figure: Iterated “$1$”s are unary arithmetic! So they can be compressed!

$\lambda$-term $\Rightarrow$ compressed $\lambda$-term
Compressed de Bruijn terms

- Iterated λs (represented as a block of constructors $1/1$ in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them.
- It makes sense to represent that number more efficiently in the usual binary notation!
- In de Bruijn notation, blocks of λs can wrap either applications or variable occurrences represented as indices.
- We need only two constructors:
  - $v/2$ indicating in a term $v(K,N)$ that we have $K$ λs wrapped around the de Bruijn index $v(N)$
  - $a/3$ indicating in a term $a(K,X,Y)$ that $K$ λs 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* – they can be seen as labeled binary trees.
Generating binary trees

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

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

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

generating trees with given number of internal nodes

generateTree(N, T) :- generateTree(T, N, 0).

generateTree(x) --> [].
generateTree((X > Y)) --> down, generateTree(X), generateTree(Y).
Generating lambda terms
Generating Motzkin trees

- Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2
- ⇒ like lambda term trees, for which we ignore the de Bruijn indices that label their leaves

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).
Generating closed de Bruijn terms

- we can derive a generator for closed lambda terms in de Bruijn form by extending the Motzkin-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.

\[
\begin{align*}
\text{genDBterm}(v(X), V) &\rightarrow \{ \text{down}(V, V0), \text{between}(0, V0, X) \}. \\
\text{genDBterm}(l(A), V) &\rightarrow \text{down}, \ \{ \text{up}(V, \text{NewV}) \}, \\
&\quad \text{genDBterm}(A, \text{NewV}). \\
\text{genDBterm}(a(A, B), V) &\rightarrow \text{down}, \\
&\quad \text{genDBterm}(A, V), \\
&\quad \text{genDBterm}(B, V). \\
\end{align*}
\]
Generating closed de Bruijn terms – continued

`genDB(L, T) :- genDB(T, 0, L, 0).`  % terms of size L
`genDBs(L, T) :- genDB(T, 0, L, _).`  % terms of size up to L

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)))`. 
Generation of linear lambda terms

- *linear lambda terms* restrict binders to *exactly one* occurrence
- `linLamb/4` uses logic variables both as leaves and as lambda binders and generates terms in standard form
- Binders accumulated on the way down from the root, must be split between the two branches of an application node
- `subset_and_complement_of/3` achieves this by generating all such possible splits of the set of binders
  
  \[
  \begin{align*}
  &\text{linLamb}(X, [X]) \rightarrow [].
  
  &\text{linLamb}(l(X,A), Vs) \rightarrow \text{down, linLamb}(A, [X \mid Vs]).
  
  &\text{linLamb}(a(A,B), Vs) \rightarrow \text{down,}
  
  &\{ \text{subset_and_complement_of}(Vs, As, Bs) \},
  
  &\text{linLamb}(A, As), \text{linLamb}(B, Bs).
  \end{align*}
  \]

- 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
Generating lambda terms of bounded unary height

- a bound on the number of lambda binders from a de Bruijn index to the root of the term

\[
\text{boundedUnary}(v(X), V, \_D) \rightarrow \{ \text{down}(V, V0), \text{between}(0, V0, X) \}.
\]

\[
\text{boundedUnary}(l(A), V, D1) \rightarrow \text{down},
\]

\[
\{ \text{down}(D1, D2), \text{up}(V, \text{NewV}) \},
\]

\[
\text{boundedUnary}(A, \text{NewV}, D2).
\]

\[
\text{boundedUnary}(a(A, B), V, D) \rightarrow \text{down},
\]

\[
\text{boundedUnary}(A, V, D),
\]

\[
\text{boundedUnary}(B, V, D).
\]

- the predicate \text{boundedUnary}/5 generates lambda terms of size \( L \) in compressed de Bruijn form with unary height \( D \)

\[
\text{boundedUnary}(D, L, T) :\neg \text{boundedUnary}(B, 0, D, L, 0), \text{b2c}(B, T).
\]

\[
\text{boundedUnarys}(D, L, T) :\neg \text{boundedUnary}(B, 0, D, L, \_), \text{b2c}(B, T).
\]
Combining term generation and type inference
Type Inference

```prolog
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).

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

slightly more complex for de Bruijn terms

?- 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).

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

?- 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.
```
Generating well typed de Bruijn terms of a given size

- we can interleave generation and type inference in one program
- DCG grammars control size of the terms with predicate `down/2`
- in terms of the Curry-Howard correspondence, the size of the generated term corresponds to the size of the (Hilbert-style) proof of the intuitionistic formula defining its type

```prolog
\begin{verbatim}
genTypedTerm(v(I),V,Vs) --> 
  nth0(I,Vs ,V0), % pick binder and ensure types match
  unify_with_occurs_check(V,V0)
).
genTypedTerm(a(A,B),Y,Vs) --> down, % application node
  genTypedTerm(A, (X->Y),Vs),
genTypedTerm(B, X,Vs).
genTypedTerm(l(A),(X->Y),Vs) --> down, % lambda node
  genTypedTerm(A,Y,[X|Vs]).
\end{verbatim}
```

?- 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) ;
```

Paul Tarau (University of North Texas)  Lambda Terms, Types and Tree-Arithmetic  CLA’2016  24 / 56
Some algorithms for SK and X combinator trees
Generating SK-combinator trees

• the most well known basis for combinator calculus consists of
  \( K = \lambda x_0. \lambda x_1. x_0 \) and \( S = \lambda x_0. \lambda x_1. \lambda x_2.((x_0 \ x_2) \ (x_1 \ x_2)) \)

• the predicate \( \text{genSK} \) generates SK-combinator trees with a limited number of internal nodes. Note that we use “*” for application. It is left associative.

\[
\begin{align*}
\text{genSK}(k) & \rightarrow [ ] . \\
\text{genSK}(s) & \rightarrow [ ] . \\
\text{genSK}(X \ast Y) & \rightarrow \text{down}, \text{genSK}(X), \text{genSK}(Y) . \\
\end{align*}
\]

\[
\begin{align*}
\text{genSK}(N, X) & \leftarrow \text{genSK}(X, N, 0) . \ % \ with \ exactly \ N \ internal \ nodes \\
\text{genSKs}(N, X) & \leftarrow \text{genSK}(X, N, _) . \ % \ with \ up \ to \ N \ internal \ nodes
\end{align*}
\]
Inferring types for SK-combinator trees

\[
\text{skTypeOf}(k, (A \to (> A))). \quad \% \text{K is well typed}
\]
\[
\text{skTypeOf}(s, (((A \to B) \to C) > (A \to B) > A \to C))). \quad \% \text{S is well-typed}
\]
\[
\text{skTypeOf}(A \ast B, Y) :- \% \text{recursion on application trees}
\]
\[
\quad \text{skTypeOf}(A, T),
\quad \text{skTypeOf}(B, X),
\quad \text{unify_with_occurs_check}(T, (X \to Y)). \quad \% \text{types must unify} !!!
\]

- Intuition: e.g., if defined in Haskell: \( s (+) \text{succ 5} = 11, \ k 10 20 = 10 \)
- type inferred for some SK-combinator expressions

\[
?- \text{skTypeOf}(k \ast k \ast k \ast k \ast k, T) .
\]
\[
T = (A \to B \to A).
\]

\[
?- \text{skTypeOf}(k \ast s \ast k, T) .
\]
\[
T = ((A \to B \to C) > (A \to B) \to A \to C).
\]

- failure to infer a type for \( SSI = SS(SKK) \).

\[
?- \text{skTypeOf}(s \ast s \ast (s \ast k \ast k), T) .
\]
\[
\text{false}.
\]
Rosser’s X-combinator

- defined as \( X = \lambda f. fKSK \), the X-combinator has the nice property of expressing both \( K \) and \( S \) in a symmetric way

\[
K = (XX)X \quad (1)
\]
\[
S = X(XX) \quad (2)
\]

- another useful property is

\[
KK = XX = \lambda x_0. \lambda x_1. \lambda x_2. x_1 \quad (3)
\]

- if we denote application with “\( > \)” and the X-combinator with “\( x \)”, this gives, in Prolog:

\[
\text{sT} (x > (x > x)) \quad \% \text{tree for the S combinator}
\text{kT} ((x > x) > x) \quad \% \text{tree for the K combinator}
\text{xxT} (x > x) \quad \% \text{tree for} \ (X X) = (K K)
\]
De Bruijn equivalents of X-combinator expressions

- **kB and sB** define the $K$ and $S$ combinators in de Bruijn form
  
  $$\begin{align*}
  kB(l(l(v(1)))). \\
  sB(l(l(l(a(a(v(2),v(0)),a(v(1),v(0))))))).
  \end{align*}$$

- the X-combinator’s definition in terms of $S$ and $K$, in de Bruijn form, is derived from $X \ f = f \ K \ S \ K$ and then $\lambda f. f \ K \ S \ K$
  
  $$xB(X) :- F = v(0), kB(K), sB(S), X = l(a(a(F,K),S),K).$$

- **t2b** transforms an X-combinator tree in its lambda expression form, in de Bruijn notation

  $$\begin{align*}
  t2b(x,X) :- xB(X). \\
  t2b((X \succ Y), a(A,B)) :- t2b(X,A), t2b(Y,B).
  \end{align*}$$
Inferring types of X-combinator trees directly

- in the paper: inferring via translation to λ-terms
- the predicate \( \text{xt} \), that can be seen as a “partially evaluated” version of \( \text{xtype} \), infers the type of the combinators directly

\[
\text{xt}(X, T) :- \text{poly_xt}(X, T), \text{bindType}(T).
\]

\[
\text{xt}(T) :- \text{t2b}(x, B), \text{btype}(B, T, []).
\]

\[
\text{poly_xt}(x, T) :- \text{xt}(T). \quad \% \text{borrowing the type of the X combinator}
\]

\[
\text{poly_xt}(A \rightarrow B, Y) :-
\text{poly_xt}(A, T),
\text{poly_xt}(B, X),
\text{unify_with_occurs_check}(T, (X \rightarrow Y)).
\]

- we proceed by first borrowing the type of \( x \) from its de Bruijn equivalent
- then, after calling \( \text{poly_xt} \) to infer polymorphic types, we bind them to our simple-type representation by calling \( \text{bindType} \)
An (injective) size proportional encoding of X-combinator expressions as $\lambda$-terms

**Proposition**

*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 $\texttt{t2b}$ 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.
Binary tree arithmetic
Blocks of digits in the binary representation of natural numbers

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

\[ n = b_0^{k_0} b_1^{k_1} \ldots b_i^{k_i} \ldots b_m^{k_m} \]  \hspace{1cm} (4)

with \( b_i \in \{0, 1\} \), \( b_i \neq b_{i+1} \) and the highest digit \( b_m = 1 \).

Proposition

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 \( 2^i (j + 1) - 1 \).

Proposition

A number \( n \) is even if and only if it contains an even number of blocks of the form \( b_i^{k_i} \) in equation (4). A number \( n \) is odd if and only if it contains an odd number of blocks of the form \( b_i^{k_i} \) in equation (4).
The constructor $c$: prepending a new block of digits

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

- the exponents are $i + 1$ instead of $i$ as we start counting at 0
- $c(i,j)$ will be even when $j$ is odd and odd when $j$ is even

**Proposition**

The equation (5) defines a bijection $c : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}^+ = \mathbb{N} - \{0\}$. 
The DAG representation of 2014, 2015 and 2016

- A more compact representation is obtained by folding together shared nodes in one or more trees
- Integers labeling the edges are used to indicate their order
Binary tree arithmetic

- parity (inferred from assumption that largest bloc is made of 1s)
- as blocks alternate, parity is the same as that of the number of blocks
- several arithmetic operations, with Haskell type classes at
- complete code at: http://www.cse.unt.edu/~tarau/research/2014/Cats.hs

**Proposition**

*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 $O(\log^*(N))$ in the worst case*
Successor \((s)\) and predecessor \((p)\)

\[
\begin{align*}
  s(x, x^>x) & . \\
  s(x^>x, x^>(x^>x)) & : -! . \\
  s(x^>Xs, Z) & : \neg \text{parity}(X^>Xs, P), s1(P, X, Xs, Z) . \\
  s1(0, x, X^>Xs, SX^>Xs) & : -s(X, SX) . \\
  s1(0, X^>Ys, Xs, x^>(PX^>Xs)) & : -p(X^>Ys, PX) . \\
  s1(1, X, x^>(Y^>Xs), X^>(SY^>Xs)) & : -s(Y, SY) . \\
  s1(1, X, Y^>Xs, x^>(x^>(PY^>Xs))) & : -p(Y, PY) . \\
  p(x^>x, x) & . \\
  p(x^>(x^>x), x^>x) & : -! . \\
  p(x^>Xs, Z) & : \neg \text{parity}(X^>Xs, P), p1(P, X, Xs, Z) . \\
  p1(0, X, x^>(Y^>Xs), X^>(SY^>Xs)) & : -s(Y, SY) . \\
  p1(0, X, (Y^>Ys)^>Xs, x^>(x^>(PY^>Xs))) & : -p(Y^>Ys, PY) . \\
  p1(1, x, X^>Xs, SX^>Xs) & : -s(X, SX) . \\
  p1(1, X^>Ys, Xs, x^>(PX^>Xs)) & : -p(X^>Ys, PX) .
\end{align*}
\]
Size-proportionate ranking/unranking for lambda terms
A size-proportionate Gödel numbering bijection for λ-terms

- injective encodings are easy: encode each symbol as a small integer and use a separator
- in the presence of a bijection between two infinite sets of data objects, it is possible that representation sizes on one side are exponentially larger than on the other side
- e.g., Ackerman’s bijection from hereditarily finite sets to natural numbers
  \[ f(\{\}\}) = 0, \quad f(x) = \sum_{a \in x} 2^{f(a)} \]
- however, if natural numbers are represented as binary trees, size-proportionate bijections from them to “tree-like” data types (including λ-terms) is (un)surprisingly easy!
- some terminology: “bijective Gödel numbering” (for logicians), same as “ranking/unranking” (for combinatorialists)
Ranking and unranking de Bruijn terms to binary-tree represented natural numbers

- **variables $v/1$:** as trees with $x$ as their left branch
- **lambda $l/1$:** as trees with $x$ as their right branch
- To avoid ambiguity, the rank for application nodes will be incremented by one, using the successor predicate $s/2$

```prolog
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).
```

- **unrank** simply reverses the operations – note the use of predecessor $p/2$

```prolog
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).
```
What can we do with this bijection?

- a size proportional bijection between de Bruijn terms and binary trees with empty leaves
- Rémy’s algorithm directly applicable to lambda terms
- a different but possibly interesting distribution
- “plain” natural number codes

?- t(666,T),unrank(T,LT),rank(LT,T1),n(T1,N).
T = T1, T1 = (x> (x> (x> ((x>x>) ((x>x>) (x> (x> (x>x))))))))),
LT = l(l(l(a(v(0), a(v(0), v(1)))))))
N = 666.
The well-typed frontier
What is the well-typed frontier?

Definition

We call well-typed frontier of a combinator tree the set of its maximal well-typed subtrees.

- contrary to general lambda terms, SK-terms are hereditarily closed i.e., every subterm of a SK-expression is closed
- the concept is well-defined for combinator expressions as all their subtrees are closed terms

Definition

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.
Computing the well-typed frontier

- We separate the trunk from the frontier and mark with fresh logic variables the replaced subtrees.
- These variables are added as left sides of equations with the frontiers as their right sides.

\[
\text{wellTypedFrontier}(\text{Term, Trunk, FrontierEqs}) :-
\text{wtf}(\text{Term, Trunk, FrontierEqs}, []).
\]

\[
\text{wtf}(\text{Term, X}) \rightarrow \{\text{typable}(\text{Term})\}, !, [X=\text{Term}].
\]

\[
\text{wtf}(A \ast B, X \ast Y) \rightarrow \text{wtf}(A, X), \text{wtf}(B, Y).
\]
Example

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

\[
? - \text{wellTypedFrontier}(s*s*(s*k*k)*(s*s*(s*k*k))),
\]

\[
\text{Trunk,FrontierEqs}.
\]

\[
\text{Trunk} = A*B* (C*D),
\]

\[
\text{FrontierEqs} = [A=s*s, B=s*k*k, C=s*s, D=s*k*k].
\]
Full reversibility: grafting back the frontier

- 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
- 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: 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)), % now the same as the term

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] .

- after grafting back the frontier, the trunk becomes equal to the term that we have started with
Simplification as normalization of the well-typed frontier

- 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 \texttt{eval} terminates
- once evaluated, we can graft back the results to the typeless trunk

?- 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 = \( k* (s*s*s* (s*s)*s*s) \).
Comparison of sizes of the typeless trunk and the well-typed frontier of SK-terms, by size

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

- while the size of the frontier dominates for small terms, it decreases progressively
- open problem: *does the average ratio of the frontier and the trunk converge to a limit as the size of the terms increases?*
Playing with the playground
Querying a generator for specific types (efficiently!)

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

Figure: logical inferences when querying with type patterns given in advance

\[- \text{queryTypedTerms}(12, (x \times x) > x, T).\]  
false.

- no closed terms of type \((x \times x) > x\) exist up to size 12
- we expect that, also as the corresponding logic formula is not a tautology in minimal logic!
Some “popular” type patterns

<table>
<thead>
<tr>
<th>Count</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>23095</td>
<td>x&gt;(x&gt;x)</td>
</tr>
<tr>
<td>22811</td>
<td>(x&gt;x)&gt;(x&gt;x)</td>
</tr>
<tr>
<td>22514</td>
<td>x&gt;x&gt;(x&gt;x)</td>
</tr>
<tr>
<td>21686</td>
<td>x&gt;x</td>
</tr>
<tr>
<td>18271</td>
<td>x&gt; ((x&gt;x)&gt;x)</td>
</tr>
<tr>
<td>14159</td>
<td>(x&gt;x)&gt;(x&gt;(x&gt;x))</td>
</tr>
<tr>
<td>13254</td>
<td>((x&gt;x)&gt;x)&gt;( (x&gt;x)&gt;x)</td>
</tr>
<tr>
<td>12921</td>
<td>x&gt; (x&gt;x)&gt;(x&gt;x)</td>
</tr>
<tr>
<td>11541</td>
<td>(x&gt;x)&gt;( (x&gt;x)&gt;x)&gt;x</td>
</tr>
<tr>
<td>10919</td>
<td>(x&gt; (x&gt;x))&gt;(x&gt; (x&gt;x))</td>
</tr>
</tbody>
</table>

Figure: Most frequent types, out of a total of 33972 distinct types, of 1016508 closed well-typed terms up to size 9.

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

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

Figure: Proportion of well-typed SK-combinator terms
Estimating the proportion of well-typed X-combinator trees

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

Figure: Proportion of well-typed X-combinator terms

Somewhat surprisingly, a large proportion of well-typed X-combinator terms is present among the binary trees of a given size.
More details in a series of papers:

- PADL’15: generation of various families of lambda terms
- PPDP’15: a uniform representation of combinators, arithmetic, lambda terms, ranking/unranking to tree-based numbering systems
- CIKM/Calculemus’15: size-proportionate ranking using a generalization of Cantor’s pairing functions to k-tuples
- ICLP’15: type-directed generation of lambda terms
- SYNASC’15: SK-combinators, well-typed frontiers
- PADL’16: the underlying tree arithmetic in terms of Catalan families of combinatorial objects (Haskell type-class) + tree arithmetic for random term generation

All Prolog-based work (70 pages paper+code) is now merged together at:
https://github.com/ptarau/play

And also at
http://arxiv.org/abs/1507.06944
Conclusions

- Prolog (and other logic and constraint programming languages) are an ideal tool for term and type generation and as well as type-inference algorithms for lambda terms and combinator expressions.
- A few new concepts: well-typed frontiers of combinator expressions, compressed deBruijn terms.
- Possible applications: compilation and test generation for lambda-calculus based languages and proof assistants.
- 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 enables the discovery of interesting patterns about the type expressions occurring in well-typed programs.
- SK and X-combinator expressions: terms and their types can share the same representation.
- Ranking/unranking to natural numbers represented as binary trees is naturally size-proportionate.