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

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

PPDP’2015

Research supported by NSF grant 1423324.
Outline

1. X-combinator trees and de Bruijn terms
2. Types as X-combinator trees
3. X-combinator trees as natural numbers
4. A size-proportionate Gödel numbering bijection for lambda terms
5. Playing with the playground – possible applications
6. Conclusion
Combinator bases

- closed terms: all variable occurrences are bound by an enclosing lambda
- **combinator expressions** are lambda terms represented as binary trees having applications as internal nodes and closed lambda terms called **combinators** as leaves
- a **combinator basis** is a set of combinators in terms of which any other combinators can be expressed
- the most well known basis for combinator calculus consists of $K = \lambda x_0. \lambda x_1. x_0$ and $S = \lambda x_0. \lambda x_1. \lambda x_2. ((x_0 \ x_2) \ (x_1 \ x_2))$
- together with the primitive operation of application, $K$ and $S$ can be used as a 2-point basis to define a Turing-complete language

Our metalanguage: a subset of Prolog, with occasional use of some built-ins, Horn clauses of the form $a_0 : - a_1, a_2 \ldots a_n$. 
Rosser’s X-combinator

- defined as $X = \lambda f. fKS\!K$, the X-combinator has the nice property of expressing both $K$ and $S$ in a symmetric way
  
  \[
  K = (XX)X 
  \]
  \[
  S = X(XX) 
  \]

- another useful property is
  
  \[
  KK = XX = \lambda x_0. \lambda x_1. \lambda x_2. x_1 
  \]

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

  \[
  sT(x>(x>x)). \ % \text{tree for the S combinator} \\
  kT((x>x)>x). \ % \text{tree for the K combinator} \\
  xxT(x>x). \ % \text{tree for (X X) = (K K)} 
  \]
Generating X-combinator trees of given size

- **genTree** generates X-combinator trees with a limited number of internal nodes
  
  \[
  \text{genTree}(x) \rightarrow [].
  \]
  
  \[
  \text{genTree}(X \succ Y) \rightarrow \text{down}, \text{genTree}(X), \text{genTree}(Y).
  \]

  \[
  \text{down}(\text{From}, \text{To}) :\neg \text{From} > 0, \text{To is From-1}.
  \]

- **Definite clause grammars (DCGs)** and the predicate \( \text{down}/2 \) (that counts downward the number of available internal nodes) specify the generation algorithm

- **Two interfaces:** \( \text{genTree}/2 \) that generates trees with exactly \( N \) and

  \[ \text{genTrees}/2 \] that generates trees with \( N \) or less internal nodes

  \[
  \text{genTree}(N,X) :\neg \text{genTree}(X,N,0).
  \]

  \[
  \text{genTrees}(N,X) :\neg \text{genTree}(X,N,\_).
  \]
Examples

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

```
?- genTrees(3,T). % up to size 3
T = x ;
T = (x>x) ;
T = (x> (x>x)) ;
T = (x> (x> (x>x))) ;
T = (x> ((x>x)>x)) ;
T = ((x>x)>x) ;
T = ((x>x> (x>x)) ;
T = ((x> (x>x)>x)) ;
T = (((x>x)>x)>x) .
```

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

```
tsize(x,0).
tsize((X>Y),S):-tsize(X,A),tsize(Y,B),S is 1+A+B.
```
X-combinator expressions as a Turing-complete language

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

```prolog
eval(X,X).
```

- **in app/3** the first two clauses mimic the rewriting corresponding to K and S
- the final clause returns the unevaluated application as its third argument

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

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

SKK = (( (x>(x>x))>((x>x)>x))>((x>x)>x)),
R = x.

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

SKX = (( (x> (x>x))>((x>x)>x))>x),
R = x.
De Bruijn Indices

- a lambda term: \( \lambda a. (\lambda b. (a (b b)) \lambda c. (a (c c))) \)
- in Prolog: \( l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))) \)

**De Bruijn Indices** provide a name-free representation of lambda terms

- terms that can be transformed by a renaming of variables (\( \alpha \)-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
De Bruijn equivalents of X-combinator expressions

- **kB and sB** define the K and S combinators in de Bruijn form
  
  \[
  \text{kB}(l(l(v(1)))).
  \]
  
  \[
  \text{sB}(l(l(l(a(a(v(2),v(0)),a(v(1),v(0))))))).
  \]

- 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 \(\lambda f. f K S K\)
  
  \[
  \text{xB}(X) :- F = v(0), \text{kB}(K), \text{sB}(S), X = l(a(a(F,K),S),K).
  \]

- **t2b** transforms an X-combinator tree in its lambda expression form, in de Bruijn notation
  
  \[
  \text{t2b}(x,X) :- \text{xB}(X).
  \]
  
  \[
  \text{t2b((X>Y),a(A,B)) :- t2b(X,A),t2b(Y,B)}.
  \]
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 $t\ 2\ b$ 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.
Inferring types of X-combinator trees directly

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

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

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

$$\text{poly_xt}(x, T) :- \text{xT}(T). \% \text{borrowing the type of the X combinator}$$

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

- 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 $\text{bindType}$
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 – larger than for \( \lambda \)-terms
Generating simply typed de Bruijn terms of a given size

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

```prolog
genTypedB(v(I),V,Vs) --> {  
nth0(I,Vs,V0), % pick binder and ensure types match  
unify_with_occurs_check(V,V0)  
}.

genTypedB(a(A,B),Y,Vs) --> down, % application node  
  genTypedB(A,X\>Y,Vs),  
  genTypedB(B,X,Vs).

genTypedB(l(A),X\>Y,Vs) --> down, % lambda node  
  genTypedB(A,Y, [X|Vs]).
```

Paul Tarau (University of North Texas)  
On combinators, arithmetic, λ-terms and types  
PPDP'2015 Research supported by NSF grant 1423324
Generating all simply-typed BCK(p) terms of given size

**BCK(p): at most p occurrences for each lambda binder (p>1: Turing-complete)**

\[
\text{genTBCK}(K, L, X, T) :- \text{genTBCX}(X, T, K, _, 0, [], [], L, 0).
\]

\[
\text{genTBCX}(v(X), T, _K1, _K2, V, Vs1, Vs2) \rightarrow \begin{cases} 
\text{selsub}(V, X:C1:T0, X:C2:T, Vs1, Vs2), \text{down}(C1, C2), \\
\text{unify_with_occurs_check}(T, T0)
\end{cases}.
\]

\[
\text{genTBCX}(l(A), (X\rightarrow Y), K1, K2, V, Vs1, Vs2) \rightarrow \text{down,} \\
\begin{cases} 
\text{up}(V, \text{NewV}) \\
\text{genTBCX}(A, Y, K1, K2, \text{NewV}, [V:K1:X|Vs1], [V:NewK:_|Vs2]), \\
\{+ \ (\text{NewK}=K2) \}
\end{cases}.
\]

\[
\text{genTBCX}(a(A, B), Y, K1, K2, V, Vs1, Vs3) \rightarrow \text{down,} \\
\text{genTBCX}(A, (X\rightarrow Y), K1, K2, V, Vs1, Vs2), \\
\text{genTBCX}(B, X, K1, K2, V, Vs2, Vs3).
\]

\[
\text{selsub}(I, X, Y, [X|Xs], [Y|Xs]) :- \text{down}(I, _). \\
\text{selsub}(I, X, Y, [Z|Xs], [Z|Ys]) :- \text{down}(I, Il), \text{selsub}(Il, X, Y, Xs, Ys).
\]

**we interleave generation and constraints – code not in the paper**
Querying the generator for specific types

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

?- genTypedBs(12, T, (x>=x)>x).
false.
Some interesting facts about simple types and their inhabitants

- total absence of type $(x > x) > x$ among terms of size up to 12
- **Transformers** of type $x > x$, by increasing sizes, give the sequence $[1, 0, 3, 3, 31, 78, 2500, 18474, 110265, 888676]$
- 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 is true for $(x > x) > ((x > x) > (x > x))$, giving $[0, 2, 1, 16, 29, 272, 940, 7594, 39075, 312797, 2115374]$
- also $(x > x) > (x > x) > ((x > x) > (x > x))$ giving $[1, 1, 5, 13, 73, 300, 1846, 10130, 69336, 469217, 3640134]$
Iterated types

- 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.
- for instance, 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)).

- a fixpoint is reached after a few steps - code and table in the paper.

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

\[ n = b_0^{k_0} b_1^{k_1} \ldots b_i^{k_i} \ldots b_m^{k_m} \tag{4} \]

with \( b_i \in \{0, 1\} \) 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 \( 2^i (j + 1) - 1 \).

**Proof.**

\( 0^i j \) corresponds to multiplication by a power of 2. If \( f(i) = 2i + 1 \), then, by induction, the \( i \)-th iterate of \( f \), \( f^i \) is computed as in the equation (5)

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

Each block \( 1^i \) in \( n \), represented as \( 1^i j \) in (4), corresponds to the iterated application of \( f \), \( i \) times, \( n = f^i(j) \).
The bijection between $\mathbb{N}$ and binary trees with empty leaves

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

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).
```

Encodings of combinators $X$, $S$, $K$ and $XX=KK$ – code for encoder $n/2$ and decoder $t/2$ in the paper

?- 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.
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 http://arxiv.org/pdf/1406.1796.pdf
- 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 \((p)\)

\[
s(x, x) .
\]
\[
s(x, x(x)) : - !.
\]
\[
s(x, x, X) : - \text{parity}(X, P), s1(P, X, X, X).
\]
\[
s1(0, x, x, X, S) : - s(X, S).
\]
\[
s1(0, x, Y, X, S, X) : - p(X, Y, P).
\]
\[
s1(1, x, Y, X, S) : - s(Y, S).
\]
\[
s1(1, x, Y, X, S, X, X) : - p(Y, P).
\]
\[
p(x, x) .
\]
\[
p(x, x, x) : - !.
\]
\[
p(x, x, x) : - \text{parity}(x, P), p1(P, x, x, x, x).
\]
\[
p1(0, x, x, Y, X, S, X) : - s(Y, S).
\]
\[
p1(0, x, (Y, S), X, x, (P, Y, X, S)) : - p(Y, S, P).
\]
\[
p1(1, x, (Y, S), S, X, S) : - s(X, S).
\]
\[
\]
in 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

\[ f(\{\}) = 0, f(x) = \sum_{a \in x} 2^{f(a)} \]

however, *if natural numbers are represented as binary trees*, size-proportionate bijections from them to “tree-like” data types (including \( \lambda \)-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** \( \nu/1 \): as trees with \( \times \) as their left branch
- **lambdas** \( l/1 \): as trees with \( \times \) 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(\nu(0), x).
rank(l(A), x > T) :- rank(A, T).
rank(\nu(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, \nu(0)).
unrank(x > T, l(A)) :- !, unrank(T, A).
unrank(T > x, \nu(N)) :- !, n(T, N).
unrank(X > Y, a(A, B)) :- p(X, X1), unrank(X1, A), p(Y, Y1), unrank(Y1, B).
```
Playing with the playground – possible applications

- size-inflating injections from \( \lambda \)-terms to \( \lambda \)-terms
- evolution of a multi-operation dynamic system
- a succinct representation of binary trees via their bijection to the language of balanced parentheses
- a possible “real” application:
  - as we have size-proportionate bijections between \( \lambda \)-terms, natural numbers, \( X \)-combinator trees and simple types, it makes sense to think about sharing their memory representation
  - a hybrid representation:
    - small trees are represented within a machine word as balanced 0,1-parentheses sequences
    - larger ones as cons-cells
  - 2-bit-tagged pointers could be used to disambiguate interpretation as numbers, combinators types or lambda expressions
  - their targets could be shared if structurally identical!
Conclusion

Prolog code at:
http://www.cse.unt.edu/~tarau/research/2015/xco.pro

- logic programming was used as a meta-language for a “declarative playground” for lambda terms and combinators
- we have explored some of the consequences of having a uniform representation for combinators, types, lambda terms and arithmetic
- size-proportionate bijections between these lead to possible practical applications

Compactness and simplicity of the code is coming from a combination of:
- logic variables / unification with occurs check / acyclic term testing
- Prolog’s backtracking – and occasional CUTs :-(
- DCGs for size testing in generators and for relation composition

The same is doable in functional programming - but with a much richer “language ontology” needed for managing state, backtracking, unification.
Future work (and work from the close enough past)

Extending our “declarative playground” for lambda terms and combinators:

- PADL’15: generation of various families of lambda terms
- CICM’15: compressed de Bruijn terms and a bijective Gödel numbering scheme using the generalized Cantor bijection from $\mathbb{N}^k$ to $\mathbb{N}$
- ICLP’15: type-directed generation of lambda terms
- plans to release it all together as a large arxiv draft + Github code