On Type-directed Generation of Lambda Terms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

ICLP’2015

Research supported by NSF grant 1423324.
Motivation

- λ-calculus is possibly the most heavily researched computational mechanism, but it has a nice property: the deeper you dig, the more interesting it becomes :-(
- λ-terms provide a foundation to modern functional languages, type theory and proof assistants
- they are now part of mainstream programming languages including Java 8, C# and Apple’s Swift, C++ 11
- Prolog’s backtracking, sound unification and Definite Clause Grammars make it an “all-in-one” meta-language for generation, type inference, symbolic computation
- this is part of a Prolog-based declarative playground for λ-terms, types and combinators at http://arxiv.org/abs/1507.06944 (70 pages and growing!)
De Bruijn Indices

Our metalanguage: a subset of Prolog, with occasional use of some built-ins, Horn clauses of the form \( a_0 : - a_1, a_2 \ldots a_n \).

- a lambda term: \( \lambda a.(\lambda b.(a\,(b\,(b)))\,\lambda c.(a\,(c\,c))) \Rightarrow \)
- in Prolog: \( l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))) \)
- \textit{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 \textit{counting the number of lambdas until the one binding them} on the way up to the root of the term
- term with canonical names: \( l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))) \Rightarrow \)
- \( \text{de Bruijn term: } l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0)))))) \)
- note: we start counting up from 0
- closed terms: every variable occurrence belongs to a binder
- open terms: otherwise
Type inference algorithm for Bruijn terms

- we associate the same logical variable, denoting its type, to each variable
- each leaf \( v/1 \) corresponds via its de Bruijn index to its binder
- the built-in \( \text{nth0}(I, Vs, V0) \) unifies \( V0 \) with the \( I \)-th element of the type context \( Vs \)
- unification with occurs-check needs to be used to avoid cycles in the inferred type formulas

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

```
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]).
```
Merging term generation and type inference ⇒ improved performance

<table>
<thead>
<tr>
<th>Size</th>
<th>Slow $\circ \rightarrow \circ$</th>
<th>Slow $\circ \rightarrow \circ \rightarrow \circ$</th>
<th>Fast $\circ \rightarrow \circ$</th>
<th>Fast $\circ \rightarrow \circ \rightarrow \circ$</th>
<th>Fast $\circ$</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: Number of logical inferences as counted by SWI-Prolog for our algorithms when querying generators with type patterns given in advance.
Querying for inhabitants of a given type

\[ \text{genTypedB}(L, B, T) :- \text{genTypedB}(B, T, [], L, 0), \text{bindType}(T). \]

\[ \text{queryTypedB}(L, \text{Term}, \text{QueryType}) :- \\
    \text{genTypedB}(L, \text{Term}, \text{Type}), \\
    \text{Type} = \text{QueryType}. \]

- **Terms of type** \( x > x \) **of size 4**

  \[ \text{?- queryTypedB}(4, \text{Term}, (o->o)). \]
  Term = \( a(l(l(v(0))), l(v(0))) \);
  Term = \( l(a(l(v(1)), l(v(0)))) \);
  Term = \( l(a(l(v(1)), l(v(1)))) \).

  \[ \text{?- queryTypedB}(10, \text{Term}, ((o->o)->o)). \]
  false.

- **the last query, taking about half a minute, shows that no closed terms of type** \( (o->o)->o \) **exist up to size 10**
Discovering frequently occurring type patterns

<table>
<thead>
<tr>
<th>Term size</th>
<th>Types</th>
<th>Terms</th>
<th>Ratio</th>
<th>1-st frequent</th>
<th>2-nd frequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>1: o-&gt;o</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>2: o-&gt;o-&gt;o</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>9</td>
<td>0.555</td>
<td>3: o-&gt;o-&gt;o-&gt;o</td>
<td>3: o-&gt;o</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>40</td>
<td>0.4</td>
<td>14: o-&gt;o-&gt;o-&gt;o</td>
<td>4: ...</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>238</td>
<td>0.231</td>
<td>38: o-&gt;o-&gt;o-&gt;o-&gt;o</td>
<td>31: o-&gt;o</td>
</tr>
<tr>
<td>6</td>
<td>235</td>
<td>1564</td>
<td>0.150</td>
<td>201: o-&gt;o-&gt;o-&gt;o</td>
<td>80: ...</td>
</tr>
<tr>
<td>7</td>
<td>1102</td>
<td>11807</td>
<td>0.093</td>
<td>732: o-&gt;o-&gt;o-&gt;o-&gt;o</td>
<td>596: o-&gt;o</td>
</tr>
<tr>
<td>8</td>
<td>5757</td>
<td>98529</td>
<td>0.058</td>
<td>4632: o-&gt;o-&gt;o-&gt;o</td>
<td>2500: o-&gt;o</td>
</tr>
<tr>
<td>9</td>
<td>33251</td>
<td>904318</td>
<td>0.036</td>
<td>20214: o-&gt;o-&gt;o-&gt;o-&gt;o</td>
<td>19855: ..</td>
</tr>
</tbody>
</table>

Figure: Counts for terms and types for sizes 1 to 9 + first two most frequent types
Some “popular” types

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

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

Figure: Most frequent types, out of a total of 33972 distinct types, of 1016508 terms up to size 9.
Generating well-typed, closed BCK(p) terms of a given size

code not in the paper: we interleave generation and multiple constraints

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

genTBCK(K, L, X, T) :- genTBCX(X, T, K, _I, 0, [], [], L, 0). % for I==0, BCI(p)

genTBCX(v(X), T, _K1, _K2, V, Vs1, Vs2) --> 
  { selsub(V, X:C1:T0, X:C2:T, Vs1, Vs2), down(C1, C2),
    unify_with_occurs_check(T, T0) }

ngenTBCX(l(A), (X->Y), K1, K2, V, Vs1, Vs2) --> down,
  { up(V, NewV) },
  genTBCX(A, Y, K1, K2, NewV, [V:K1:X|Vs1], [V:NewK:_|Vs2]),
  { \+ \+ (NewK=K2) }.

genTBCX(a(A,B), Y, K1, K2, V, Vs1, Vs3) --> down,
  genTBCX(A, (X->Y), K1, K2, V, Vs1, Vs2),
  genTBCX(B, X, K1, K2, V, Vs2, Vs3).

selsub(I, X, Y, [X|Xs], [Y|Xs]) :- down(I, _).

selsub(I, X, Y, [Z|Xs], [Z|Ys]) :- down(I, I1), selsub(I1, X, Y, Xs, Ys).
Relational queries that we can answer

- How many distinct types occur for terms up to a given size?
- What are the most popular types?
- What are the terms that share a given type?
- What is the smallest term that has a given type?
- What smaller terms have the same type as this term?
Conclusion

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

- logic programming is used as a meta-language for lambda terms and types
- 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 constraints 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.