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)))))
- *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))))) $\Rightarrow$
- 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 `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

Size	Slow ○->○	Slow ○->○->○	Fast ○->○	Fast ○->○->○	Fast ○
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

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

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

queryTypedB(L,Term,QueryType):-
  genTypedB(L,Term,Type),
  Type=QueryType.
```

- Terms of type `x>x` of size 4

```
?- queryTypedB(4,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)))) .

?- queryTypedB(10,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

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

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.

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

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)
  }.
genTBCX(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.