Lambda Terms, Types and Tree-Arithmetic

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

July 22, 2016

Research supported by NSF grant **1423324**

Outline

Overview

We overview fast generation and random sampling for simply-typable lambda terms in de Bruijn notation and ranking and unranking algorithms targeting binary-tree represented natural numbers. By interleaving constraints on term sizes with type inference steps we improve the performance of our algorithms by several orders of magnitude.

Horn Clause Prolog in two 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: the two-clause metaInterpreter

The meta-interpreter `metaint/1` uses a (difference)-list view of prolog clauses.

```
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 `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 `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).
```

```
?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```

Lambda terms in Prolog: canonical (with logic variables) and with de Bruijn indices

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))))) $\Rightarrow$
- 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

Generating lambda terms

Generating Lambda Terms (possibly open)

- we use successor arithmetic: 0, s(0),s(s(0)) ...
- possibly open term: de Bruijn indices might point higher then our lambda binders
- size definition: a/2 = 2 units, l/1 = 1 unit. s/1 = 1 unit, 0 = 0 units

```
genLambda(s(S),X):-genLambda(X,S,0).

genLambda(X,N1,N2):-nth_elem(X,N1,N2).
genLambda(l(A),s(N1),N2):-genLambda(A,N1,N2).
genLambda(a(A,B),s(s(N1)),N3):-
  genLambda(A,N1,N2),
  genLambda(B,N2,N3).

nth_elem(0,N,N).
nth_elem(s(X),s(N1),N2):-nth_elem(X,N1,N2).
```

Examples

```
?- genLambda(s(s(s(0))),Term).
Term = s(s(0)) ;
Term = l(s(0)) ;
Term = l(l(0)) ;
Term = a(0, 0) ;
false.

?- genLambda(s(s(s(s(0)))),Term).
Term = s(s(s(0))) ;
Term = l(s(s(0))) ;
Term = l(l(s(0))) ;
Term = l(l(l(0))) ;
Term = l(a(0, 0)) ;
Term = a(0, s(0)) ;
Term = a(0, l(0)) ;
Term = a(s(0), 0) ;
Term = a(l(0), 0) ;
false.
```

Generating closed terms

- a list, initially empty of variables is built
- each lambda binder pushes a variable to it
- each leaf is constrained to correspond, via its de Bruijn index to a variable
- we use the list to count binders - but it will later hold types that we infer

```
genClosed(s(S),X):-genClosed(X,[],S,0).

genClosed(X,Vs,N1,N2):-nth_elem_on(X,Vs,N1,N2).
genClosed(l(A),Vs,s(N1),N2):-genClosed(A,[_|Vs],N1,N2).
genClosed(a(A,B),Vs,s(s(N1)),N3):-
  genClosed(A,Vs,N1,N2),
  genClosed(B,Vs,N2,N3).

nth_elem_on(0,[_|_],N,N).
nth_elem_on(s(X),[_|Vs],s(N1),N2):-nth_elem_on(X,Vs,N1,N2).
```

Example

```
?- genClosed(s(s(s(0))),Term).
Term = l(l(0)) .

?- genClosed(s(s(s(s(0)))),Term).
Term = l(l(s(0))) ;
Term = l(l(l(0))) ;
Term = l(a(0, 0)) .

?- genClosed(s(s(s(s(s(0))))),Term).
Term = l(l(l(s(0)))) ;
Term = l(l(l(l(0)))) ;
Term = l(l(a(0, 0))) ;
Term = l(a(0, l(0))) ;
Term = l(a(l(0), 0)) ;
Term = a(l(0), l(0)) .
```

Combining term generation and type inference

Generating simply-typable de Bruijn terms of a given size

- type mimic function application (i.e., β-reduction of lambda terms)
- we refine our program generating closed terms by imposing constraints on the variables introduced by lambda binders
- de Bruijn indices pointing to the same variable should agree on types
- unification with "occurs-check": circular types are not allowed

```
genTypable(s(S),X,T):-genTypable(X,T,[],S,0).

genTypable(X,V,Vs,N1,N2):-genIndex(X,Vs,V,N1,N2).
genTypable(l(A),(X->Xs),Vs,s(N1),N2):-genTypable(A,Xs,[X|Vs],N1,N2).
genTypable(a(A,B),Xs,Vs,s(s(N1)),N3):-
  genTypable(A,(X->Xs),Vs,N1,N2),
  genTypable(B,X,Vs,N2,N3).

genIndex(0,[V|_],V0,N,N):-unify_with_occurs_check(V0,V).
genIndex(s(X),[_|Vs],V,s(N1),N2):-genIndex(X,Vs,V,N1,N2).
```

Generating large simply-typable random lambda terms with Boltzmann samplers

The uniform random generation mechanism

- we generate random (possibly) open terms while ensuring that closedness, type and size constraints hold at each step
- we stop when the term is resulting term is closed and simply-typed

```
ranLamb(M,X,T,N,I):-between(1,M,I),ranTypable(X,T,N),!.

min_size(100).
max_size(120).
max_steps(10000000).

ranTypable(X,T,Size):-
  max_size(Max),min_size(Min),
  random(R), ranTypable(Max,R,X,T,[],0,Size0),
  Size0>=Min,Size is Size0+1.
```

the Boltzmann sampler for simply-typed lambda terms

```
ranTypable(Max,R,X,V,Vs,N1,N2):-R<0.3524422987,!,random(NewR),
  pickIndex(Max,NewR,X,Vs,V,N1,N2).
ranTypable(Max,R,l(A),(X->Xs),Vs,N1,N3):-R<0.648039998,!,
  next(Max,NewR,N1,N2),
  ranTypable(Max,NewR,A,Xs,[X|Vs],N2,N3).
ranTypable(Max,_R,a(A,B),Xs,Vs,N1,N5):- % _R>=0.648039998
  next(Max,R1,N1,N2),
  ranTypable(Max,R1,A,(X->Xs),Vs,N2,N3),
  next(Max,R2,N3,N4),
  ranTypable(Max,R2,B,X,Vs,N4,N5).

pickIndex(_,R,0,[V|_],V0,N,N):-R<0.70440229,!,
  unify_with_occurs_check(V0,V).
pickIndex(Max,_,s(X),[_|Vs],V,N1,N3):-
  next(Max,NewR,N1,N2),
  pickIndex(Max,NewR,X,Vs,V,N2,N3).

next(Max, R,N1,N2):-N1<Max,N2 is N1+1,random(R).
```

Example

```
term: l(l(a(l(l(a(a(l(0),a(l(0),a(l(l(l(a(0,a(0,a(a(l(a(0,
      l(a(a(l(a(l(a(0,l(a(l(0),s(s(0))))))),l(0))),l(a(0,s(0)))),
      l(s(s(0))))))),a(l(s(s(0))),0)),0)))))),l(0)))),
      l(l(a(l(l(a(l(l(0)),0))),
      a(a(s(0),l(0)),a(l(a(a(s(0),l(0)),
      l(0))),l(l(l(s(s(0))))))))))),
      l(a(l(0),a(0,a(s(0),0)))))))

type: (A->((B->C)->B)->D->((((E->E)->E->E)->(E->E)->E->E)->
      ((E->E)->E->E)->(E->E)->E->E)->((E->E)->E->E)->(E->E)->E->E)

steps to find a typable term: 230284

size of the term: 102
```

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

$$n = b_0^{k_0} b_1^{k_1} \ldots b_i^{k_i} \ldots b_m^{k_m} \tag{1}$$

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 (1). 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 (1).

The constructor $\texttt{c}$: prepending a new block of digits

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

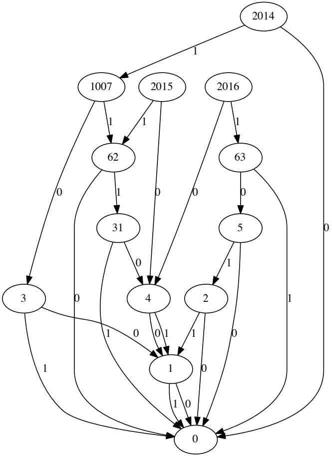
- the exponents are $i+1$ instead of i as we start counting at 0
- $c(i,j)$ will be even when j is odd and odd when j is even

Proposition

The equation (2) defines a bijection $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^{+} = \mathbb{N} - \{0\}$.

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 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/2015/GCat.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 and* $O(log^*(N))$ *in the worst case*

```
parity(0,0).
parity(_**0,1).
parity(_**(X**Xs),P1):-parity(X**Xs,P0),P1 is 1-P0.
```

Successor (s) and predecessor (p)

```
s(0,0**0).
s(X**0,X**(0**0)):-!.
s(X**Xs,Z):-parity(X**Xs,P),s1(P,X,Xs,Z).

s1(0,0,X**Xs,SX**Xs):-s(X,SX).
s1(0,X**Ys,Xs,0**(PX**Xs)):-p(X**Ys,PX).
s1(1,X,0**(Y**Xs),X**(SY**Xs)):-s(Y,SY).
s1(1,X,Y**Xs,X**(0**(PY**Xs))):-p(Y,PY).

p(0**0,0).
p(X**(0**0),X**0):-!.
p(X**Xs,Z):-parity(X**Xs,P),p1(P,X,Xs,Z).

p1(0,X,0**(Y**Xs),X**(SY**Xs)):-s(Y,SY).
p1(0,X,(Y**Ys)**Xs,X**(0**(PY**Xs))):-p(Y**Ys,PY).
p1(1,0,X**Xs,SX**Xs):-s(X,SX).
p1(1,X**Ys,Xs, 0**(PX**Xs)):-p(X**Ys,PX).
```

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 $f(\{\}) = 0, f(x) = \sum_{a \in x} 2^{f(a)}$
- 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

- after ranking, we bring it down by 1 by applying predecessor
- any other enumeration of binary trees can be used instead, that provides `s/2` and `p/2`

```
rank(X,A):-rank1(X,SA),p(SA,A).

rank1(0,0**0).
rank1(X**Y,0**(X**Y)).
rank1(l(X),SA**0):-rank1(X,SA).
rank1(a(X,Y),SA**SB):-rank1(X,SA),rank1(Y,SB).
```

- unrank simply reverses the operations

```
unrank(A,X):-s(A,SA),unrank1(SA,X).

unrank1(0**X,X).
unrank1(SA**0,l(X)):-unrank1(SA,X).
unrank1(SA**SB,a(X,Y)):-unrank1(SA,X),unrank1(SB,Y).
```

What can we do with this bijection?

- a size proportional bijection between de Bruijn terms and binary trees with empty leaves
- uniform random generation of binary trees Rémy's algorithm directly applicable to lambda terms
- a different but possibly interesting distribution
- "plain" natural number codes

```
natural number=47
tree_num=((0**0)**0)**(0**(0**0))
unranked=a(l(0),l(0))
has_type=(A->A)
```

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, simply-typable 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
- applications for generation large simply-typable random lambda terms: test generation for lambda-calculus based languages and proof assistants
- ranking/unranking to natural numbers represented as binary trees is naturally size-proportionate