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**

Outline

1. An Overview of our playground
2. Horn Clause Prolog in three slides
3. Lambda terms in Prolog: canonical, de Bruijn, compressed
4. Generating lambda terms
5. Combining term generation and type inference
6. Some algorithms for SK and X combinator trees
7. Binary tree arithmetic
8. Size-proportionate ranking/unranking for lambda terms
9. The well-typed frontier
10. Playing with the playground
11. Conclusions

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

– continued –

- 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

– continued –

- 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

```
a0 --> a1,a2,...,an.
```

into a clause where predicates have two extra arguments expressing a chain of state changes as in

```
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),...,an(Sn-1,Sn).
```

- 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):-f(In,Temp),g(Temp,Out).
```

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, 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))))) $\Rightarrow$
- 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

Should we compress λ-terms in de Bruijn notation?

Figure: Random λ-terms can have long necks: l(l(l(l(l(... ...a(...

Figure: Iterated "`l`"s are unary arithmetic! So they can be compressed!

λ-term $\Rightarrow$ compressed λ-term

Compressed de Bruijn terms

- iterated λs (represented as a block of constructors `l/1` in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them
- $\Rightarrow$ 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
- $\Rightarrow$ 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

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

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

?- genTreeByDepth(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

```
genTree(N,T):-genTree(T,N,0).

genTree(x)-->[].
genTree((X>Y))-->down,genTree(X),genTree(Y).
```

Generating lambda terms

Generating Motzkin trees

- Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2
- $\Rightarrow$ 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

```
genDBterm(v(X),V)--> {down(V,V0),between(0,V0,X)}.
genDBterm(l(A),V)--> down, {up(V,NewV)},
  genDBterm(A,NewV).
genDBterm(a(A,B),V)--> down,
  genDBterm(A,V),
  genDBterm(B,V).
```

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

```
linLamb(X,[X])-->[].
linLamb(l(X,A),Vs)-->down,linLamb(A,[X|Vs]).
linLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  linLamb(A,As),linLamb(B,Bs).
```

- 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

```
boundedUnary(v(X),V,_D)-->{down(V,V0),between(0,V0,X)}.
boundedUnary(l(A),V,D1)-->down,
  {down(D1,D2),up(V,NewV)},
  boundedUnary(A,NewV,D2).
boundedUnary(a(A,B),V,D)-->down,
  boundedUnary(A,V,D),
  boundedUnary(B,V,D).
```

- the predicate `boundedUnary/5` generates lambda terms of size `L` in compressed de Bruijn form with unary hight `D`

```
boundedUnary(D,L,T):-boundedUnary(B,0,D,L,0),b2c(B,T).
boundedUnarys(D,L,T):-boundedUnary(B,0,D,L,_),b2c(B,T).
```

Combining term generation and type inference

Type Inference

```
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

```
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]).
```

```
?- 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) ;
```

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 `genSK` generates SK-combinator trees with a limited number of internal nodes. Note that we use "*" for application. It is left associative.

```
genSK(k)-->[].
genSK(s)-->[].
genSK(X*Y)-->down,genSK(X),genSK(Y).

genSK(N,X):-genSK(X,N,0). % with exactly N internal nodes

genSKs(N,X):-genSK(X,N,_). % with up to N internal nodes
```

Inferring types for SK-combinator trees

```
skTypeOf(k,(A>(_>A))).                    % K is well typed
skTypeOf(s,(((A>B>C)> (A>B)>A>C))). % S is well-typed
skTypeOf(A*B,Y):-            % recursion on application trees
  skTypeOf(A,T),
  skTypeOf(B,X),
  unify_with_occurs_check(T,(X->Y)). % types must unify !!!
```

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

```
?- skTypeOf(k*k*k*k*k,T).
T = (A>B>A).

?- skTypeOf(k*s*k,T).
T = ((A>B>C)> (A>B)>A>C).
```

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

```
?- skTypeOf(s*s*(s*k*k),T).
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

$$K = (XX)X \tag{1}$$

$$S = X(XX) \tag{2}$$

- another useful property is

$$KK = XX = \lambda x_0.\, \lambda x_1.\, \lambda x_2.x_1 \tag{3}$$

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

```
sT(x>(x>x)). % tree for the S combinator
kT((x>x)>x). % tree for the K combinator
xxT(x>x).    % 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

```
kB(l(l(v(1)))).

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 $\lambda f.f\ K\ S\ K$

```
xB(X):-F=v(0),kB(K),sB(S),X=l(a(a(a(F,K),S),K)).
```

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

```
t2b(x,X):-xB(X).
t2b((X>Y),a(A,B)):-t2b(X,A),t2b(Y,B).
```

Inferring types of X-combinator trees directly

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

```
xt(X,T):-poly_xt(X,T),bindType(T).

xT(T):-t2b(x,B),btype(B,T,[]).

poly_xt(x,T):-xT(T). % borrowing the type of the X combinator
poly_xt(A>B,Y):-
  poly_xt(A,T),
  poly_xt(B,X),
  unify_with_occurs_check(T,(X>Y)).
```

- we proceed by first borrowing the type of `x` from its de Bruijn equivalent
- then, after calling `poly_xt` to infer polymorphic types, we bind them to our simple-type representation by calling `bindType`

An (injective) size proportional encoding of X-combinator expressions as λ-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 `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

$$n = b_0^{k_0} b_1^{k_1} \dots b_i^{k_i} \dots b_m^{k_m} \tag{4}$$

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

$$c(i,j) = \begin{cases} 2^{i+1}j & \text{if } j \text{ is odd,} \\ 2^{i+1}(j+1)-1 & \text{if } j \text{ is even.} \end{cases} \tag{5}$$

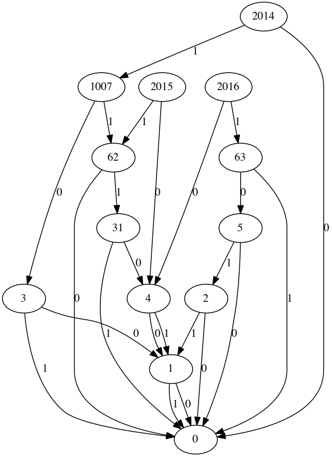
- the exponents are $i+1$ instead of i as we start counting at 0
- $c(i,j)$ will be even when j is odd and odd when j is even

Proposition

The equation (5) defines a bijection $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+ = \mathbb{N} - \{0\}$.

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/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 and* $O(\log^*(N))$ *in the worst case*

Successor (s) and predecessor (p)

```
s(x,x>x).
s(X>x,X>(x>x)):-!.
s(X>Xs,Z):-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):-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).
```

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

- variables `v/1`: as trees with `x` as their left branch
- lambdas `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`

```
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`

```
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

```
wellTypedFrontier(Term,Trunk,FrontierEqs):-
  wtf(Term, Trunk,FrontierEqs,[]).

wtf(Term,X)-->{typable(Term)},!,[X=Term].
wtf(A*B,X*Y)-->wtf(A,X),wtf(B,Y).
```

Example

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

```
?- wellTypedFrontier(s*s*(s*k*k)*(s*s*(s*k*k)),
                     Trunk,FrontierEqs).
Trunk = A*B* (C*D),
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
- $\rightarrow$ we can simplify an untypable term by normalizing the members of its frontier, for which we are sure that `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

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

- 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!)

Size	Slow `x>x`	Slow `x>(x>x)`	Fast `x>x`	Fast `x>(x>x)`	Fast `x`
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: logical inferences when querying with type patterns given in advance

```
?- queryTypedTerms(12,(x>x)>x,T).
false.
```

- no closed terms of type `(x>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

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

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

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

Figure: Proportion of well-typed SK-combinator terms

Estimating the proportion of well-typed X-combinator trees

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

Figure: Proportion of well-typed X-combinator terms

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