On Synergies between Type Inference, Generation and Normalization of SK-combinator Trees

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

SYNASC'2015

Research supported by NSF grant **1423324.**

Motivation

- combinators are lambda terms of a special form that predate lambda calculus (Schönfinkel in the 1920s and then rediscovered by Curry)
- the language of SK-combinator expressions is Turing complete
- like in the case of general lambda terms, the very interesting sub-language of simply typed terms is decidable
- logic programming provides a convenient metalanguage for modeling data types and computations taken from other programming paradigms
- properties of logic variables, unification with occurs-check, and exploration of solution spaces via backtracking facilitate compact algorithms for inferring types or generating terms for various calculi
- $\rightarrow$ we want to explore, as part of a “logic programming playground” the synergies between term generation and type inference on the language of S and K combinators

Outline

Combinator expressions / trees

- λ-terms: $Term = Var \; ; \; \lambda Var.Term \; ; \; (Term \; Term)$
- 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 definite clause grammars (DCGs), all based on Horn clauses of the form a_0 :- $a_1, a_2 \ldots a_n$.

Related work

- consequences of the Curry Howard isomorphism:
  - S,K serve as axioms for minimal logic (with Modus Ponens)
  - simple types are tautologies in minimal logic
  - inhabitants of a type correspond to (Hilbert-style) proofs in minimal logic
- classic work on simple types and type inference, covering also combinators: Hindley and Seldin
- Grygiel and P. Lescanne: counting and generating lambda terms
- asymptotics: overlap with the study of classic and intuitionistic tautologies
- most relevant: 2015 paper by Bendkowski, Grygiel, and Zaionc - with focus on asymptotic density of classes of SK-combinator expressions
  - almost all weakly normalizing terms are not strongly normalizing
  - almost all strongly normalizing terms are not normal forms
  - almost all normal forms are not typable

Generating combinator trees

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

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

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

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

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

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).
```

A Turing-complete evaluator for SK-combinator trees

```
eval(k,k).
eval(s,s).
eval(F*G,R):-eval(F,F1),eval(G,G1),app(F1,G1,R).

app((s*X)*Y,Z,R):-!,  % S
  app(X,Z,R1),
  app(Y,Z,R2),
  app(R1,R2,R).
app(k*X,_Y,R):-!,R=X. % K
app(F,G,F*G).
```

Applications of SKK and SKS, both implementing the identity combinator $I = \lambda x.x$.

```
?- app(s*k*k,s,R).
R = s.

?- app(s*k*s,k,R).
R = k.
```

Inferring simple types for SK-combinator trees

```
skTypeOf(k,(A->(_B->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.
```

Estimating the proportion of well-typed SK-combinator trees

- what proportion of SK-combinator trees of a given size are well-typed?
- `simpleTypeOf`: we focus on types over a single base type "o"
- generate all terms of given size and infer their types
- types inferred for terms with 2 internal nodes:

```
?- genSK(1,X),simpleTypeOf(X,T).
X = k*k,T = (o->o->o->o) ;

X = k*s,T = (o-> (o->o->o)-> (o->o)->o->o) ;

X = s*k,T = ((o->o)->o->o) ;

X = s*s,mT = (((o->o->o)->o->o)->(o->o->o)->o->o) .
```

C_n counts the number of binary trees with n internal nodes, each of which has $n+1$ leaves, each of which can be either S or K, therefore

Proposition

There are $2^{n+1} C_n$ SK-trees with n nodes, where C_n is the n-th Catalan number.

Counts for well-typed SK-combinator expressions and their ratio to the total number of SK-trees of given size

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

- higher density of simply typed terms than for general λ-terms
- open problem: what happens asymptotically?

The well-typed frontier of an SK-expression

- untypable SK-expressions become the majority as soon as the size of the expression reaches some threshold, 9 in this case
- this actually is a good thing, from a programmer's perspective: types help with bug-avoidance partly because being "accidentally well-typed" becomes a low probability event for larger programs
- we want to decompose an untypable SK-expression into a set of maximal typable ones

Type-directed generation of SK-combinator trees

- given a type, finding a term that has that type (called an *inhabitant*) is *PSPACE*-complete
- generation of random terms is guided by their types, results in more realistic (while not uniformly random) terms
- useful for debugging compilers that use λ-terms as intermediate code

Generating simple types

- our types are just binary trees of a given size

```
genType(o)-->[].
genType((X->Y))-->down,genType(X),genType(Y).

genType(N,X):-genType(X,N,0).   % types with exactly N arrows

genTypes(N,X):-genType(X,N,_). % types with up to N arrows
```

- example: type trees with up to 2 internal nodes (and up to 3 leaves).

```
?- genTypes(2,T).
T = o ;
T = (o->o) ;
T = (o->o->o) ;
T = ((o->o)->o) .
```

Generating SK-trees by increasing type sizes

The predicate `genByType` first generates simple types with `genType` and then uses the unification-based querying mechanism to generate, for each of the types, its inhabitant SK-trees with fewer internal nodes then their their type.

```
genByTypeSK(L,X,T):-
  genType(L,T),
  genSKs(L,X),
  simpleTypeOf(X,T).
```

The number of such terms grows quite fast, the sequence describing the number of terms with sizes smaller or equal than the size of their types up to 7 is `0, 3, 29, 250, 3381, 48968, 809092`.

```
?- genByTypeSK(2,B,T).
B = k, T = (o->o->o) ;
B = k*k*k, T = (o->o->o) ;
B = k*k*s, T = (o->o->o) .
```

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*?

Conclusions

- we have selected the minimalist pure combinator language built from applications of combinators S and K to explore aspects of their generation and type inference algorithms
- $\rightarrow$ some interesting new facts about the density and distribution of their types
- new concepts of *well-typed frontier* and *typeless trunk*
- the ability to extend (sure) termination beyond simply-typed terms, by evaluating and then grafting back their well-typed frontier

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

Integrated in large (70 pages) *Logic Programming Playground for Lambda Terms, Combinators, Types and Tree-based Arithmetic* at:
`https://github.com/ptarau/play`

Future work

- random SK-tree generation e.g., by extending Rémy's algorithm from binary trees to SK-combinator trees
- $\rightarrow$ better empirical estimates on the asymptotic behavior of the concepts introduced in this paper
- lifting well-typed frontier to general lambda terms (which are not hereditarily closed) seems possible by defining the frontier as being a sequence of maximal well-typed closed lambda terms

Integrate in our declarative playground for lambda terms and combinators:

- PADL'15: generation of various families of lambda terms
- PPDP'15: type inference, X-combinators, ranking/unranking to a binary tree-based number system
- 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