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.
- 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

1. The SK-combinator calculus
2. Overview of related work
3. Generating combinator trees
4. An evaluator for the Turing-complete language of SK-combinator trees
5. Inferring simple types for SK-combinator trees
6. Estimating the proportion of well-typed SK-combinator trees
7. The well-typed frontier of an untypable SK-expression
8. Type-directed generation of SK-combinator trees
9. Conclusions
Combinator expressions / trees

- \( \lambda \)-terms: \( \text{Term} = \text{Var} \; \lambda \text{Var}. \text{Term} \; (\text{Term} \; \text{Term}) \)
- closed terms: all variable occurrences are bound by an enclosing lambda
- \textit{combinator expressions} are lambda terms represented as binary trees having applications as internal nodes and closed lambda terms called \textit{combinators} as leaves
- a \textit{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 \texttt{genSK} generates SK-combinator trees with a limited number of internal nodes. Note that we use “*” for application. It is left associative.

\begin{verbatim}
\texttt{genSK(k)-->[].}
\texttt{genSK(s)-->[].}
\texttt{genSK(X*Y)-->down,genSK(X),genSK(Y).}
\end{verbatim}

\begin{verbatim}
\texttt{down(From,To):-From>0,To is From-1.}
\end{verbatim}

\begin{verbatim}
\texttt{genSK(N,X):-genSK(X,N,0). \% with exactly N internal nodes}
\texttt{genSKs(N,X):-genSK(X,N,\_). \% with up to N internal nodes}
\end{verbatim}

Prolog’s DCG preprocessor transforms a clause defined with “-->” like
\begin{verbatim}
a0 --> a1,a2,...,an.
\end{verbatim}
into a clause where predicates have two extra arguments expressing a chain of state changes as in
\begin{verbatim}
a0(S0,Sn):-a1(S0,S1),a2(S1,S2),... ,an(Sn-1,Sn).
\end{verbatim}
A Turing-complete evaluator for SK-combinator trees

eval(k, k).
eval(s, s).
eval(F \ast G, R) :- eval(F, F1), eval(G, G1), app(F1, G1, R).

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

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

?- app(s \ast k \ast k, s, R).
R = s.

?- app(s \ast k \ast 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:

\[ \text{?– genSK(1,X),simpleTypeOf(X,T).} \]
\[ X = k*k, T = (o->o->o->o) ; \]
\[ X = k*s, T = (o-> (o->o->o)-> (o->o)->o->o) ; \]
\[ X = s*k, T = ((o->o)->o->o) ; \]
\[ X = s*s, mT = (((o->o->o)->o->o)->(o->o->o)->o->o) . \]

\( C_n \) 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

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

- Higher density of simply typed terms than for general \( \lambda \)-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 $\lambda$-terms as intermediate code
Generating simple types

- Our types are just binary trees of a given size

\[
\text{genType}(o) \rightarrow [] .
\]

\[
\text{genType}( (X \rightarrow Y) ) \rightarrow \text{down} , \text{genType}(X) , \text{genType}(Y) .
\]

\[
\text{genType}(N,X) :\neg \text{genType}(X,N,0) . \quad \% \text{types with exactly N arrows}
\]

\[
\text{genTypes}(N,X) :\neg \text{genType}(X,N,\_). \quad \% \text{types with up to N arrows}
\]

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

\[
?- \text{genTypes}(2,T) .
\]

\[
T = o ; \\
T = (o\rightarrow o) ; \\
T = (o\rightarrow o\rightarrow o) ; \\
T = ((o\rightarrow o)\rightarrow o) .
\]
Generating SK-trees by increasing type sizes

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

\begin{verbatim}
\texttt{genByTypeSK(L,X,T):-}
\hspace{1em} \texttt{genType(L,T),}
\hspace{1em} \texttt{genSKs(L,X),}
\hspace{1em} \texttt{simpleTypeOf(X,T).
}\end{verbatim}

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.

?- \texttt{genByTypeSK(2,B,T).}
\texttt{B = k, T = (o->o->o) ;}
\texttt{B = k*k*k, T = (o->o->o) ;}
\texttt{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

```prolog
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).
```
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].
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 \(\text{eval}\) terminates
- once evaluated, we can graft back the results to the typeless trunk

?- \(\text{Term} = s*s*s* (s*s)*s* (k*s*k)\), simplifySK(\(\text{Term}\), \(\text{Trunk}\)).

\(\text{Term} = s*s*s* (s*s)*s* (k*s*k),\)
\(\text{Trunk} = s*s*s* (s*s)*s*s.\)

?- \(\text{Term} = k* (s*s*s* (s*s)*s* (k*s*k))\), simplifySK(\(\text{Term}\), \(\text{Trunk}\)).

\(\text{Term} = k* (s*s*s* (s*s)*s* (k*s*k)),\)
\(\text{Trunk} = k* (s*s*s* (s*s)*s*s).\)
Comparison of sizes of the typeless trunk and the well-typed frontier of SK-terms, by size

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

- 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
- 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