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

Paul Tarau
Dept. of Computer Science and Engineering
Univ. of North Texas
Denton, USA
paul.tarau@unt.edu

***Abstract*—The S and K combinator expressions form a well-known Turing-complete subset of the lambda calculus. Using Prolog as a meta-language, we specify evaluation, type inference and combinatorial generation algorithms for SK-combinator trees. In the process, we unravel properties shedding new light on interesting aspects of their structure and distribution. We study the proportion of well-typed terms among size-limited SK-expressions as well as the type-directed generation of terms of sizes smaller then the size of their simple types. We also introduce the *well-typed frontier of an untypable term* and we use it to design a simplification algorithm for untypable terms taking advantage of the fact that well-typed terms are normalizable.**

***Keywords*-SK-combinator calculus, lambda calculus, type inference, normalization, generation of well-typed combinator expressions, logic programming as meta-language.**

I. Introduction

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. This holds in particular for lambda terms and combinators [1]. Combinators are lambda terms of a special form that predate lambda calculus. They were discovered in the 1920s by Schönfinkel and then independently by Curry. With *function application* as their unique operation, and a convenient *base*, e.g., $K = \lambda x.\ \lambda y.\ x$ and $S = \lambda f.\ \lambda g.\ \lambda x.(f\ x)\ (g\ x)$, they form a Turing-complete subset of the lambda calculus.

This paper focuses, using some essential Prolog ingredients, on synergies between term generation and type inference on the language of S and K combinators. SK-combinator expressions are binary trees with leaves labeled S or K and internal nodes representing function application. We will explore some of their combinatorial properties, while taking advantage of readily available unification (cyclical and with occurs-check) as provided by today's Prolog systems. While working with a drastically simplified model of computation, interesting patterns emerge, some of which may extend to the richer combinator languages used in compilers for functional languages like Haskell and ML and proof assistants like Coq and Agda.

Of particular interest are type inference algorithms and the possibility to melt them together with generators of terms of a limited size. Evaluation of SK-combinator expressions (called *normalization*) has the nice property of always terminating on terms that have simple types. This suggests looking into how this property (called *strong normalization*) can be used to simplify untypable (and possibly not normalizable) terms through normalization of their maximal typable subterms. At the same time, this suggests trying to discover some empirical hints about the distribution of well-typed terms and the structure induced by typable subterms of untypable terms.

The paper is organized as follows. Section II introduces SK-combinator trees together with a generator and an evaluation algorithm. Section III defines simple types for SK-combinator expressions and describes a type inference algorithm on for SK-combinator trees. Section IV introduces the well-typed frontier of an untypable SK-expression and describes a partial normalization-based simplification algorithm that terminates on all SK-expressions. Section V hosts a discussion about the benefits and limitations of our algorithms. Section VI overviews related work. Section VII suggests some future work and concludes the paper.

The paper is structured as a literate Prolog program. The code, tested with SWI-Prolog 6.6.6 and YAP 6.3.4. It is availible from http://www.cse.unt.edu/~tarau/research/2015/skt.pro.

II. SK-Combinator Trees

The most well known basis for combinator calculus consists of $K = \lambda x_0.\ \lambda x_1.x_0$ and $S = \lambda x_0.\ \lambda x_1.\ \lambda x_2.((x_0\ x_2)\ (x_1\ x_2))$. SK-combinator expressions can be seen as binary trees with leaves labeled with symbols S and K, having function applications as internal nodes. Together with the primitive operation of application, K and S can be used as a 2-point basis to define a Turing-complete language.

A. Generating combinator trees

Prolog is an ideal language to define in a few lines generators for various classes of combinatorial objects. The predicate `genSK` generates SK-combinator trees with a limited number of internal nodes.

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

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

Note the use of Prolog's definite clause grammar (DCG) notation in combination with the predicate `down/2` that counts downward the number of available 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).
```

The predicate `genSK/3` provides two interfaces: `genSK/2` that generates trees with exactly N internal nodes and `genSKs/2` that generates trees with N or less internal nodes.

```
genSK(N,X):-genSK(X,N,0).

genSKs(N,X):-genSK(X,N,_).
```

Example 1: SK-combinator trees with up to 1 internal nodes (and up to 2 leaves).

```
?- genSKs(1,T).
T = k ;
T = s ;
T = k*k ;
T = k*s ;
T = s*k ;
T = s*s .
```

The predicate `csize` defines the size of an SK-combinator tree in terms of the number of its internal nodes.

```
csize(k,0).
csize(s,0).
csize((X*Y),S):-csize(X,A),csize(Y,B),S is 1+A+B.
```

B. A Turing-complete evaluator for SK-combinator trees

An evaluator for SK-combinator trees recurses over application nodes, evaluates their subtrees and then applies the left one to the right one.

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

In the predicate `app`, handling the application of the first argument to the second, we describe in the first two clauses the actions corresponding to `K` and `S`. The final clause returns the unevaluated application as its third argument.

```
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).
```

Example 2: 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.
```

C. De Bruijn equivalents of SK-combinator expressions

De Bruijn indices [2] provide a *name-free* representation of lambda terms. A lambda term is called *closed* if it contains no free variables. All closed terms that can be transformed by a renaming of variables (α-conversion) will share a unique de Bruijn representation. The representation is based on representing occurrences as the length of the path to their binder. Variables following lambda abstractions are omitted and their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* is found on the way up to the root of the term. We represent them using the constructor `a/2` for application, `l/1` for lambda abstractions (that we will call shortly *binders*) and `v/1` for marking the integers corresponding to the de Bruijn indices.

For instance, $\lambda x_0.(\ \lambda x_1.(x_0\ (x_1\ x_1))\ \ \lambda x_2.(x_0\ (x_2\ x_2)))$ becomes `l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1), a(v(0),v(0))))))`, corresponding to the fact that `v(1)` is bound by the outermost lambda (two steps away, counting from `0`) and the occurrences of `v(0)` are bound each by the closest lambda, represented by the constructor `l/1`. The predicates `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 predicate `c2b` transforms an SK-combinator tree in its lambda expression form, in de Bruijn notation, by replacing leaves with their de Bruijn form of the S and K combinators and replacing recursively the constructor "`*`"/2 with the application nodes "`a`"/2.

```
c2b(s,S):-sB(S).
c2b(k,K):-kB(K).
c2b((X*Y),a(A,B)):-c2b(X,A),c2b(Y,B).
```

Example 3: Expansion of some small SK-combinator trees to de Bruijn forms.

```
?- c2b(k*k,R).
R = a(l(l(v(1))), l(l(v(1)))).

?- c2b(k*s,R).
R = a(l(l(v(1))),l(l(l(a(a(v(2),
         v(0)),a(v(1),v(0))))))).
```

Clearly their de Bruijn equivalents are significantly larger than the corresponding combinator trees, but it is easy to see that this is only by a constant factor, i.e. at most the size of the S combinator.

Proposition 1: The lambda terms equivalent to SK-combinators computed by `c2b` are closed.

Proof: As the lambda term equivalent of the SK-combinator term is clearly a closed expression, the proposition follows from the definition of `c2b`, as it builds terms that apply closed terms to closed terms. ■

This well-known property holds, in fact, for all combinator expressions. It follows that combinator expressions have a stronger, *hereditary* closedness property: every subtree of a combinator tree also represents a closed expression.

Besides being closed, lambda terms interesting for functional languages and proof assistants are also *well-typed*. While the K and S combinators are known to be well-typed, we would like to see how this property extends to SK-combinator trees. In particular, we would like to have an idea on the asymptotic density of well-typed SK-combinator tree expressions. We will take advantage of Prolog's sound unification algorithm to define a type inferrer directly on SK-terms.

III. Inferring simple types for SK-combinator trees

A natural way to define types for combinator expressions is to borrow them from their lambda calculus equivalents [3]. This makes sense, as they represent the same function i.e., they are extensionally the same. However, this is equivalent to just borrowing the well-known types of the `S` and `K` combinators and then recurse over application nodes.

We will next describe an algorithm for inferring types directly on SK-combinator trees.

A. A type inference algorithm for SK-terms

Simple types will be defined here also as binary trees built with the constructor "`->/2`" with leaves representing the unique primitive type "`o`". For brevity, we will mean simple types when mentioning types, from now on. Types can be seen as as a "binary tree approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (called *strong normalization*), as the following well known property states (see [3], [4]).

Proposition 2: Lambda terms (and combinator expressions, in particular) that have simple types are strongly normalizing.

When modeling lambda terms in a functional or procedural language, inferring types requires implementing unification with occurs-check, as shown for instance in the appendix of [5]. On the other hand this operation is readily available in today's Prolog systems.

```
skTypeOf(k,(A->(_B->A))).
skTypeOf(s,(((A->B->C)-> (A->B)->A->C))).
skTypeOf(A*B,Y):-
  skTypeOf(A,T),
  skTypeOf(B,X),
  unify_with_occurs_check(T,(X->Y)).
```

At this point, most general types are inferred by `skTypeOf` as fresh variables, similar to multi-parameter polymorphic types in functional languages, if one interprets logic variables as universally quantified.

Example 4: Type inferred for some SK-combinator expressions. Note the failure to infer a type for $SSI = SS(SKK)$.

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

As we are only interested in simple types with only one base type, we will bind uniformly the leaves of our type tree to the constant "`o`" representing our only primitive type, by using the predicate `bindWithBaseType/1`.

```
simpleTypeOf(A,T):-
  skTypeOf(A,T),
  bindWithBaseType(T).

% bind all variables with type 'o'
bindWithBaseType(o):-!.
bindWithBaseType((A->B)):-
  bindWithBaseType(A),
  bindWithBaseType(B).
```

Example 5: Simple type inferred for combinators KSK, B and C.

```
?- simpleTypeOf(k*s*k,T).
T = ((o->o->o)-> (o->o)->o->o).
?- B=s*(k*s)*k,C=s*(B*B*s)*(k*k),
   simpleTypeOf(B,TB),simpleTypeOf(C,TC).
B = s* (k*s)*k,
C = s* (s* (k*s)*k* (s* (k*s)*k)*s)* (k*k),
TB = ((o->o)-> (o->o)->o->o),
TC = ((o->o->o)->o->o->o).
```

It is also useful to define the predicate `typable` that succeeds when a type can be inferred.

```
typable(X):-skTypeOf(X,_).
```

B. Estimating the proportion of well-typed SK-combinator trees

An interesting question arises at this point: *what proportion of SK-combinator trees of a given size are well-typed*? While the analytic study of the *asymptotic density* has been successfully performed on several families of lambda terms [6], [7], [8], [5], it is considered an open problem for well-typed terms. We will limit ourselves here to empirically estimate it, as it is done in [5] for general lambda terms, where experiments indicate the extreme sparsity for very large terms.

We can use our generator `genSK` to enumerate SK-combinator trees among which we can then count the number of well-typed ones.

Example 6: 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,
T = (((o->o->o)->o->o)->(o->o->o)->o->o) .
```

Similarly, we can use it also to enumerate untypable terms.

Example 7: The smallest two untypable SK-expressions.

```
?- genSKs(2,X), \+typable(X).
I = 2,X = s*s*k ;
I = 2,X = s*s*s .
```

We can implement a generator for well-typed SK-trees, to be used to compute the ratio between the number of well-typed SK-trees and the total number of SK-trees of size n, as well as one for the untypable SK-trees.

```
genTypedSK(L,X,T):-genSK(L,X),simpleTypeOf(X,T).

genUntypableSK(L,X):-genSK(L,X),\+skTypeOf(X,_).
```

To compute the proportion of well-typed terms among terms of a given size we will also need to count the number of SK-trees with n internal nodes.

Proposition 3: There are $2^{n+1}C_n$ SK-trees with n nodes, where C_n is the n-th Catalan number.

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

Fig. 1. Proportion of well-typed SK-combinator terms

Proof: If follows from the fact that 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. ■

The predicate `cat/2` computes the nth-Catalan number efficiently using the recurrence $C_0 = 1, C_n = \frac{2(2n-1)}{n+1} C_{n-1}$ [9].

```
cat(0,1).
cat(N,R):-N>0,
  PN is N-1,
  cat(PN,R1),
  R is 2*(2*N-1)*R1//(N+1).
```

Figure 1 shows the counts for well-typed SK-combinator expressions and their ratio to the total number of SK-trees of given size.

Somewhat surprisingly, a large proportion of well-typed SK-combinator terms is present among the binary trees of a given size, indicating the possible existence of a lower bound that might be easier to determine analytically than in the case of general lambda terms.

C. Generating typed SK-combinator trees by types

In [10] generation of random terms is guided by their types, resulting in more realistic (while not uniformly random) terms, used successfully in discovering some bugs in the Glasgow Haskell Compiler (GHC).

1) Generating type trees: We can generate simple types as binary trees with leaves "o" and branches "->" in a way similar to our generator for SK-trees `genSK`.

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

The predicate `genType/3` provides two interfaces, `genType` for generating types of size N and `genTypes` for generating types up to size `N`.

```
genType(N,X):-genType(X,N,0).

genTypes(N,X):-genType(X,N,_).
```

Example 8: 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) .
```

2) 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`.

Example 9: Enumeration of closed simply-typed SK combinator trees with types of size 2 and less then 2 internal nodes.

```
?- 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) .
```

IV. The well-typed frontier of an untypable SK-expression

As in the case of lambda terms, untypable SK-expressions become the majority as soon as the size of the expression reaches some threshold, 9 in this case. This actually turns out to be 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.

Driven by a curiosity somewhat similar to that about distribution and density properties of prime numbers, one would want to decompose an untypable SK-expression into a set of maximal typable ones. This makes sense, as, contrary to lambda expressions, SK-trees are uniquely built with application operations as their internal nodes from their well-typed SK combinator leaves.

Definition 1: We call *well-typed frontier* of a combinator tree the set of its maximal well-typed subtrees.
Note also, that contrary to general lambda terms, SK-terms are *hereditarily closed* i.e., every subterm of a SK-expression is closed. Consequently, the well-typed frontier is made of closed terms.

Definition 2: 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.

A. Computing the well-typed frontier

The well-typed frontier of a combinator tree and its typeless trunk are computed together by the predicate `wellTypedFrontier` . It actually proceeds by separating the trunk from the frontier and marking 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 10: 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 while 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 11: 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)),
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] .
```

Note that after grafting back the frontier, the trunk becomes equal to the term that we have started with.

B. A comparison of the sizes of the well-typed frontier and the typeless trunk

An interesting question arises at this point: *how do the sizes of the frontier and the trunk compare*?

Figure 2 compares the average sizes of the frontier and the trunk for terms up to size 8. This indicates that, while the size of the frontier dominates for small terms, it decreases progressively. This leaves the following open problem: *does the average ratio of the frontier and the trunk converge to a limit as the size of the terms increases*? More empirical information on this can be obtained by studying what happens for randomly generated large SK-trees.

C. Simplification as normalization of the well-typed frontier

Given that well-typed terms are strongly normalizing, 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, as implemented the predicate `simplifySK`.

```
simplifySK(Term,Trunk):-
  wellTypedFrontier(Term,Trunk,FrontierEqs),
  extractFrontier(FrontierEqs,Fs),
  maplist(eval,Fs,NormalizedFs),
  maplist(arg(1),FrontierEqs,Vs),
  Vs=NormalizedFs.
```

The following question arises at this point: *are there terms that are not normalizable that can be simplified by extracting and simplifying their well-typed frontier and then grafting it back*? Combinatorial search, using the `genSK` predicate finds them starting at size `8`.

Example 12: Simplifying some untypable terms for which normalization is non-terminating.

```
?- 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).
```

Note that, as expected, while simplification does not bring termination to the normalization predicate `eval/2`, it shows the existence of non-terminating computations for which a terminating simplification is possible.

V. Discussion

While the well-typed (and closed) frontier does not make sense for general lambda terms where closed terms may have open subterms, it makes sense for other combinator or supercombinator languages [11], some with practical uses in the compilation of functional languages.

Among the open problems we leave for future research, is to find out if concepts like the well-typed frontier of a richer combinator-language can be used for suggesting a fix to a program in a typed functional programming language, or to produce more precise error messages in case of type errors. For instance, it would be interesting to know if a minimal well-typed alternative can be be inferred and suggested to the programmer on a type error.

If one replaces the `unify_with_occurs_check` in predicate `skTypeOf` with the cyclic term unification (that most modern Prologs use by default), one can observe that every combinator expression passes the test! The predicate `uselessTypeOf` implements this variation.

```
uselessTypeOf(k,(A->(_B->A))).
uselessTypeOf(s,(((A->B->C)-> (A->B)->A->C))).
uselessTypeOf((A*B),Y):-
  uselessTypeOf(A,(X->Y)),
  uselessTypeOf(B,X).
```

After defining the predicates `notReallyTypable` and `sameAsAny`

```
notReallyTypable(X):-uselessTypeOf(X,_).

sameAsAny(L,M):-genSK(L,M),notReallyTypable(M).
```

one can notice the identical behavior of `sameAsAny` and `genSK`, meaning that failing the occurs-check is the exclusive reason of failure to infer a type. This happens in the presence of a unique basic type "o". However, in the case of a more realistic type system with multiple basic types like `Boolean`, `Int`, `String` etc., the failure of type inference could also be a consequence of mismatched basic types. Knowing more about these two reasons for failure might suggest weakened

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

Fig. 2. Comparison of sizes of the typeless trunk and the well-typed frontier of SK-terms, by size.

type systems where some limited form of circularity is acceptable, provided that no basic type mismatches occur. While strong normalization would be sacrificed if such circular types were accepted, one might note that this is already the case in practical languages, where fixpoint operators or recursive data type definitions are allowed.

VI. Related work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [4]. Combinators originate in Moses Schönfinkel's 1924 paper, and independently, in Haskell Curry's work in 1927. A modern introduction to combinators and their relation to lambda calculus is [3] and a first application of an extended set of combinators in the implementation of functional programming languages is [12]. Originally introduced in [2], the de Bruijn notation makes terms equivalent up to α-conversion and facilitates their normalization.

Combinatorics of lambda terms and combinators, including enumeration, random generation and asymptotic behavior has seen an increased interest recently (see [13], [6], [5], [7], [8]), partly motivated by applications to software testing. In [10], types are used to generate random terms for software testing.

Of particular interest are the results of [5] where recurrence relations and asymptotic behavior are studied for several families of lambda terms. Empirical evaluation of the density of closed simply-typed general lambda terms, described in [5], indicates extreme sparsity for large sizes. However, the problem of their exact asymptotic behavior is still open. This has, in part, motivated our interest in the empirical evaluation of the density of simply-typed SK-combinator trees, where we observed significantly higher initial densities and where there's a chance that the open problem of their asymptotic behavior might be easier to tackle.

VII. Conclusions and future work

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. While a draconian simplification of real-life programming languages, this well-known and well-researched subset of lambda calculus has revealed some interesting new facts about the density and distribution of their types. The new concepts of *well-typed frontier* and *typeless trunk* of an untypable term can be generalized to realistic combinator and supercombinator-based intermediate languages used by compilers for functional languages and proof assistants. As they give precise hints about the points where type inference failed, they are likely to be useful for debugging programs and give more meaningful compile-time error messages. The ability to extend (sure) termination beyond simply-typed terms, by evaluating and then grafting back their well-typed frontier might be useful, if ported to a more realistic context, to improve combinator-based compilers.

Our logic programming-based investigation has relied on synergies between unification of logic variables (including support for cyclic terms and occurs-check) with Prolog's backtracking mechanism. These features, present in today's Prolog systems, recommend logic programming as a convenient meta-language for exploring, at a small scale, some interesting concepts at the foundations of other programming paradigms.

Future work is likely to focus on studying how these results extend to other families of combinators and supercombinators that occur in practical languages as well as on random SK-tree generation eg., by extending Rémy's algorithm [14] from binary trees to SK-combinator trees. This would allow fast generation of very large SK-combinator expressions that could give better empirical estimates on the asymptotic behavior of the concepts introduced in this paper and their properties. Also, as a step toward more practical uses, lifting the concept of 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.

Acknowledgement

This research has been supported by NSF grant `1423324`.

References

[1] H. P. Barendregt, *The Lambda Calculus Its Syntax and Semantics*, revised ed. North Holland, 1984, vol. 103.

[2] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem," *Indagationes Mathematicae*, vol. 34, pp. 381–392, 1972.

[3] J. R. Hindley and J. P. Seldin, *Lambda-calculus and combinators: an introduction*. Cambridge University Press Cambridge, 2008, vol. 13.

[4] H. P. Barendregt, "Lambda calculi with types," in *Handbook of Logic in Computer Science*. Oxford University Press, 1991, vol. 2.

[5] K. Grygiel and P. Lescanne, "Counting and generating lambda terms," *J. Funct. Program.*, vol. 23, no. 5, pp. 594–628, 2013.

[6] O. Bodini, D. Gardy, and B. Gittenberger, "Lambda-terms of bounded unary height." in *ANALCO*. SIAM, 2011, pp. 23–32.

[7] R. David, K. Grygiel, J. Kozik, C. Raffalli, G. Theyssier, and M. Zaionc, "Asymptotically almost all λ-terms are strongly normalizing," *Preprint: arXiv: math. LO/0903.5505 v3*, 2010.

[8] K. Grygiel, P. M. Idziak, and M. Zaionc, "How big is BCI fragment of BCK logic," *J. Log. Comput.*, vol. 23, no. 3, pp. 673–691, 2013.

[9] R. P. Stanley, *Enumerative Combinatorics*. Belmont, CA, USA: Wadsworth Publ. Co., 1986.

[10] M. H. Palka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST'11. New York, NY, USA: ACM, 2011, pp. 91–97.

[11] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. NJ, USA: Prentice-Hall, Inc., 1987.

[12] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979.

[13] R. David, C. Raffalli, G. Theyssier, K. Grygiel, J. Kozik, and M. Zaionc, "Some properties of random lambda terms," *Logical Methods in Computer Science*, vol. 9, no. 1, 2009.

[14] J.-L. Rémy, "Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire," *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, vol. 19, no. 2, pp. 179–195, 1985.