Strong Normalization of the Simply-Typed Lambda Calculus in Lean by Decomposition Into the Call-By-Need SK Combinators

Dowland Aiello

5/26/25

Contents

1 Abstract

Proofs of strong normalization of the simply-typed lambda calculus have been exhaustively enumerated in the literature. A common strategy invented by W. W. Tait known as "Tait's method," (Robert Harper, 2022) interprets types as sets of "well-behaving" terms which are known to be strongly normalizing and composed of expressions in some such set. Strong normalization of the typed call-by-need SK combinator calculus has been

comparatively under-studied. Herein, I demonstrate that the typical proof of strong normalization using Tait's method holds for the typed SK combinator calculus. I also show that decomposition of the STLC into SK combinator expressions simplifies the typical proof of strong normalization.

2 A Type Discipline for the SK Combinators

I consider the usual SK combinator calculus defined as such:

$$Kxy = x \tag{1}$$

$$Sxyz = xz(yz) \tag{2}$$

A natural interpretation of the combinators as typed functions results in the dependent typing:

$$\frac{\Gamma \vdash A : K \;\; \Gamma, x : A \vdash B : L}{\Gamma \vdash (\forall x : A.B) : L}$$

$$\frac{}{\Gamma T_n : T_{n+1}}$$

$$\frac{\Gamma\alpha : T_n, \beta : T_m, x : \alpha, y : \beta}{\Gamma \vdash K : (\forall x, y.\alpha)}$$

$$\frac{\Gamma\alpha : T_n, \beta : T_m, x : \alpha, y : \beta}{\Gamma \vdash Kxy : \alpha}$$

$$\frac{\Gamma\alpha : T_n, \beta : T_m, \gamma : T_o, x : (\forall x : \alpha, y : \beta.\gamma), y : (\forall x : \alpha.\alpha), z : \alpha}{\Gamma \vdash S : (\forall x, y, z.\gamma)}$$

$$\frac{\Gamma\alpha : T_n, \beta : T_m, \gamma : T_o, x : (\forall x : \alpha, y : \beta.\gamma), y : (\forall x : \alpha.\alpha), z : \alpha}{\Gamma \vdash Sxz(yz) : \gamma}$$

3 Decomposition of the Simply-Typed Lambda Calculus into Dependently Typed SK Combinators

I utilize an SK compilation scheme outlined in "The Implementation of Functional Programming Languages" (Peyton Jones, Simon L., 1987):

$$(\lambda x.e_1\ e_2)\ arg = S(\lambda x.e_1)(\lambda x.e_2)\ arg \qquad (3)$$
$$(\lambda x.x) = SKK \qquad (4)$$
$$(\lambda x.c) = Kc \qquad (5)$$

I consider a generic simply-typed lambda calculus with base types B, a type constructor $\rightarrow$ and the type universe:

$$T = \{t \mid t \in B\}\ \cup\ \{t \mid \exists\ t_1 \in T, t_2 \in T, t = t_1 \rightarrow t_2\}$$

3.1   Type Expressivity & Equivalence

I define a mapping (M_t) from the $\rightarrow$ type constructor to $\forall$: $(\alpha \rightarrow \beta) \mapsto \forall x : \alpha.\beta$. I also assume the existence of a mapping (M_c) from the base types B to arbitrary objects in my dependently-typed SK combinator calculus. Type inference is trivially derived from the above inference rules: $\forall c \in B, \exists\ t, t', c : t \implies t' = M_t t \implies M_c c : t$.

It follows that every well-typed expression in our simply-typed lambda calculus has an equivalent well-typed SK expression:

Proof. Assume (1) that for all $c \in B, \exists!\ c' \in M_c, c' = M_c c$. Assume (2) that for all $\{t_1, t_2, t\} \subset T, t = (t_1 \rightarrow t_2), \exists!\ t' \in M_t, t' = M_t t$. Per above and induction on (1) there exists a mapping from every lambda expression to an SK combinator expression. It follows by induction on $e : t$, where e is well-typed per the inference rules that all $t \in$ the simply-typed T are in M_t. It suffices to conclude that all well-typed expressions have well-typed counterparts in the dependently-typed SK combinator calculus. $\square$

4   Proof

In order to prove strong normalization of the STLC, it suffices to demonstrate that a) no well-typed lambda calculus expression is inexpressible in the dependently-typed SK combinator calculus; and b) all well-typed SK combinator expressions are strongly normalizing.

4.1   Comprehensiveness of the SK Mapping

Proof. Suppose (1) there exists some well-typed expression e of type $t \in T$ in the STLC which is not representible in the dependently-typed SK combi-

nator calculus. By induction:

- If the expression is a constant, it must be contained in M_c, per the above lemma. **contradiction**

- If the expression is a well-typed expression contained in M_c which is a dependently-typed SK expression, its type is inferred per the inference rules. The expression is thus representible. **contradiction**

- If the expression is a well-typed lambda expression, its type is of the form: $\alpha \to \beta$, where $\{\alpha, \beta\} \subset T$. An image must exist in M_t per above of the form $\forall x : \alpha.\beta$.

    - Its body is also well-typed, and has a valid type. Its body is thus representible **by induction**.

    - The expression is thus representible, per the decomposition rules. **contradiction**

- If the expression is a well-typed application $e_1 e_2$, its left hand side is of type $\alpha \to \beta$, where $\{\alpha, \beta\} \subset T$. Its right hand side must be of type *beta*. The expression is thus of type t. By induction, the expression is representible. **contradiction**

Conclusion: no expression exists which has no image in the set of well-typed dependently-typed SK combinator expressions. $\qquad\square$

4.2 Strong Normalization of the Typed SK Combinators

I assume the existence of a one-step reduction function: `eval_once`

I define strong-normalization inductively (where e is an SK combinator expression) as:

```
inductive strongly_normalizing : Expr → Prop
  | trivial e : eval_once e = e → strongly_normalizing e
  | hard e : strongly_normalizing (eval_once e) →
            strongly_normalizing e
```

4.2.1 Reducibility Candidates

The K combinator is trivially strongly normalizing. It invokes no function applications, although it may produce an expression which contains an application. For example:

$$K(KK)y = KK$$

Borrowing from Tait's method, I define a mapping $R(t)$ where t is a type (expression) in our dependently-typed SK combinator calculus. The image of t is a set containing every well-typed expression of type t which is composed of expressions living in $R(t')$ for their respective types t'. I constrain the set such that all one-step reduxes of K are in R.

$$\forall \alpha : T_n, \ \beta : T_m, \ x : \alpha, \ y : \alpha, \ R(\forall x, y.\alpha) =$$
$$\{K \mid K : (\forall x, y.\alpha) \wedge \forall arg_1 : \alpha, \ arg_2 : \beta,$$
$$\texttt{eval\_once} \ K \ arg_1 \ arg_2 \in R(\alpha)\}$$

Or, more succinctly:

$$\forall \alpha : T_n, \ \beta : T_m, \ x : \alpha, \ y : \alpha, \ R(\forall x, y.\alpha) = \{K \mid K : (\forall x, \ y.\alpha) \wedge arg_1 \in R(\alpha)\}$$

$R(t)$ can be similarly extended to include the S combinator.

$$\forall \alpha : T_n, \ \beta : T_m, \ \gamma : T_o,$$
$$T_x = (\forall x : \alpha, y : \beta.\gamma), \ T_y = (\forall x : \alpha.\alpha), \ T_z = \alpha,$$
$$x : T_x, \ y : T_y, \ z : T_z,$$
$$R(\forall x, y, z.\gamma) = \{S \mid S : (\forall x, y, z.\gamma), \ \forall arg_1 : T_x, \ arg_2 : T_z, \ arg_3 : T_z,$$
$$arg_1 \in R(T_x) \wedge arg_2 \in R(T_y) \wedge arg_3 \in R(T_z)\}$$

Expressions which are obviously reducible and inert are as follows:

$$R(T_{n+1}) = \{T_n\}$$
$$\forall K : T_n, \ L : T_m, \ A : K, \ B : L, \ R(L) = \{\text{fall} \mid, \ \text{fall} = (\forall x : A.B) \wedge \text{fall} : L\}$$

4.2.2 Inductive Proof

It suffices in order to prove strong normalization of this sytem that a) all reducibility candidates in R are strongly-normalizing; and c) all well-typed expression $(e : t)$ can be expressed using expressions in $R(t)$.

1. Preservation

   In order to execute an inductive proof leveraging our definition of $R(t)$, it is useful to prove that evaluation maintains the typing of an expression.

   Lemma 4.1. *For all well-typed expressions, $e : t \implies (eval\_once\ e) : t$.*

   Proof. The proof is obvious for obviously reducible expressions of the form T_n and $(\forall x : A.B)$. The $K : t$ combinator is inert ($eval\_once\ k = k \implies t = t'$) except when it is provided two well-typed arguments: $K(x : t_1)(y : t_2)$. Per the inference rules, $(Kxy) : t$ is of the type $t = t_1$. Evaluation of Kxy is defined to be equivalent to x. Thus, preservation is trivially achieved. □

2. Proof Execution

   Lemma 4.2. *All expressions e which are well-typed with type t and occupy the set $R(t)$ are strongly normalizing.*

   Proof. Inductively:

   - All obviously reducing candidates are strongly normalizing:
     - All expressions of the form T_n are strongly normalizing, as they are inert.
     - All expressions of the form $(\forall x : A.B)$ are strongly normalizing, as they are inert.
   - All $K : t$ combinators in $R(t)$ are strongly normalizing. K is insert, and invokes no function applications. By the definition of $R(t)$, $K \in R(t)$ will produce only one-step reduxes which are in R, and which are strongly normalizing **by induction**. Thus, the expression is **strongly normalizing**.

- All $S : t$ combinators in $R(t)$ are strongly normalizing. S is not inert, and invokes $xz(yz)$. However, x, y, and z live in R, requiring that their one-step reduxes live in R and are strongly-normalizing. The expression is strongly-normalizing **by induction**.

$\square$

Lemma 4.3. *All well-typed expressions $(e : t)$ occupy the set $R(t)$.*

Proof. The proof is trivially proven for objects of the form T_n and $(\forall x : A.B)$, as above.

- All well-typed $K : t$ combinators are of the type $t = \forall x, y.\alpha$, where x is well-typed $(x : \alpha)$ and y is well-typed $(y : \beta)$. $x \in R(\alpha) \land y \in R(\beta)$ **by induction**.

$\square$

4.3 Strong Normalization of the STLC

4.4 Encoding in Lean

Peyton Jones, Simon L. (1987). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*, Prentice-Hall, Inc..
Robert Harper (2022). *How to (Re)Invent Tait's Method.*