

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	1
2	A Type Discipline for the SK Combinators	2
3	Decomposition of the Simply-Typed Lambda Calculus into Dependently Typed SK Combinators	2
3.1	Type Expressivity & Equivalence	3
4	Proof	3
4.1	Comprehensiveness of the SK Mapping	3
4.2	Strong Normalization of the Typed SK Combinators	4
4.2.1	Reducibility Candidates	4
4.2.2	Inductive Proof	5
4.3	Strong Normalization of the STLC	6
4.4	Encoding in Lean	6

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:

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

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

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

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

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):

$$(\lambda x. e_1 \ e_2) \ arg = S(\lambda x. e_1)(\lambda x. e_2) \ arg \tag{3}$$

$$(\lambda x. x) = SKK \tag{4}$$

$$(\lambda x. c) = Kc \tag{5}$$

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 combinator 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: $\alpha \rightarrow \beta$, 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 $\alpha \rightarrow \beta$, where $\{\alpha, \beta\} \subset T$. Its right hand side must be of type β . The expression is thus of type α . 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. \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 reduses of K are in R .

$$\{K \mid \forall \alpha : T_n, \beta : T_m, x : \alpha, y : \alpha, K : (\forall x, y. \alpha), \forall arg_1 : \alpha, arg_2 : \beta, \text{eval_once } K \ arg_1 \ arg_2 \in R(\alpha)\} \quad (6)$$

Or, more succinctly:

$$\{K \mid \dots, arg_1 \in R(\alpha)\}$$

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

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

Expressions which are obviously reducible and inert are as follows:

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

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 b) all well-typed expression $(e : t)$ can be expressed using only expressions in $R(t)$.

1. Strong Normalization of Reducibility Candidates

Lemma 4.1. *All expressions e which are well-typed with type t and occupy the set $R(t)$ are strongly normalizing.*

Proof. Inductively:

•

□

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*.