TDK-thesis

Zhenyun Yin

There is no finite model of the SK combinator calculus

Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Author:

Zhenyun Yin

Computer Science BSc

II. grade

Supervisor:

Ambrus Kaposi

Associate professor

Budapest, 2024

Contents

Chapter 1

Introduction

Combinatory logic [1] was proposed by Moses Schönfinkel around 1920, and rediscovered by Haskell Curry in 1927 [2]. It is the earliest universal model of computation, possessing the same expressive power as the λ-calculus, which was invented by Alonso Church in the 1930s [3]. The emergence of combinatory logic [1] allows us to simplify function representations by eliminating the need for variables, so that in computer science this logic serves as a simplified computational model, finding applications in computability theory and proof theory. Specifically, the SKI combinator calculus, as a combinatory logic system and computational system [4], is also the simplest known Turing-complete language.

This thesis explores the properties of combinator algebras [5] and seeks to answer several key questions: Is there a two-element combinator algebra? If not, can we generalize that no finite model exists from the absence of a two-element combinator algebra? Finally, we prove the non-existence of a finite non-trivial model of the SK combinator calculus. The answers to these questions are well-known; the novelty of our work is that it comes with a complete Agda proof, see [6].

1.1 Related work

Combinatory logic is a discipline that has been developed for about a hundred years, and many conclusions have long been proven by many researchers. In these books [7, 5, 8], many properties of combinatory logic and combinatory algebras are explained in detail. The methodology of these books is different from ours. We understand languages as algebraic theories and several well-known results are clarified by this new perspective, e.g. the correspondence of lambda calculus and combinatory logic [9].

Lemma 6.2.6 in the book by Bimbo [8] proves that there is no nontrivial finite combinatory algebra. The method of proof is different from ours, we rely on Church encoding of products, while she uses iterated right-associated sequences of K combinators.

1.2 Structure of this thesis

Chapter 2 introduces the SK combinator calculus, defines our notion of model, and presents examples illustrating the translation from lambda terms and the Church encoding of natural numbers. We give some examples of programming in SK combinators.

Chapter 3 provides proof that no finite model exists, employing methods such as translation from lambda terms, Church encoding of finite types, and the pigeonhole principle.

We conclude this thesis by outlining future work, which primarily involves exploring infinite models for the SK combinator calculus.

Chapter 2

Programming in a model

Combinator calculus avoids the use of contexts and variables to express higher-order functions. However, writing and understanding programs in SK combinator calculus is more challenging compared to lambda calculus. For this reason, we derive lambda expressions first. This method improves the derivation of SK combinator terms, making it a more accessible approach than direct programming with SK combinators. In this chapter, we will explore the SK combinator calculus, define our notion of the model, and give several examples of programming in combinatory logic. Because combinatory terms are quite unreadable, we use the conversion from lambda terms [10].

2.1 SK combinator calculus

The SK combinator calculus forms a foundational part of combinatory logic, utilizing a minimal set of operators to encode complex computational expressions. Within this calculus, the operators S and K with parentheses can be combined in various ways to form expressions. To understand how to use the combinator calculus to express any Turing-computable functions without giving variables, we need to focus on their abstract structure.

And all the notions like f, g, u in the beta rule are meta variables standing for arbitrary combinator terms.

These terms are manipulated through specific rewriting rules to achieve computation: $(\mathrm{K}u)f \implies u$, where the K combinator ignores its second argument, and $((\mathrm{S}f)g)u \implies (fu)(gu)$, where the S combinator applies the result of applying its first argument to an argument to the result of applying its second argument to the same argument.

Let us consider these rules with equivalent lambda expressions:

1. Kβ: $(\mathrm{K}u)f = u$ which represents $\lambda uf.u$
2. Sβ: $((\mathrm{S}f)g)u = (fu)(gu)$ which represents $\lambda fgu.fu(gu)$

or with their algebraic structure:

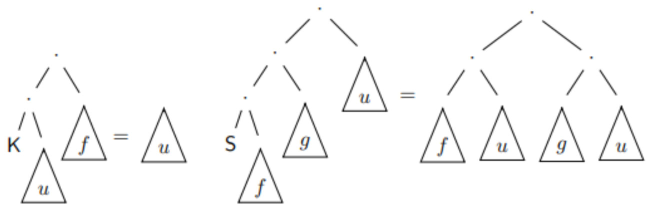

Figure 2.1: Combinator calculus syntax [11]

In fact, the clearest and most formal representation of expressions within the SK combinator calculus uses binary trees. It is easier to see that (KK)K is not equal to K(KK), indicating that this structure does not have the associativity.

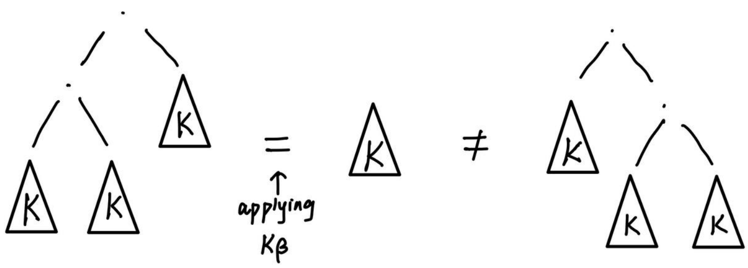

Figure 2.2: (KK)K $\neq$ K(KK)

2.2 Combinator algebras

Combinator calculus as an algebraic theory [9] can be formalized as the following record type in Agda [12]. A model consists of one sort (terms), one binary operation (parenthesized to the left), two nullary operations, and two term equations:

```
record Model : Setι where
  infixl 5 _·_
  field
    Tm  : Set
    _·_ : Tm → Tm → Tm
    K S : Tm
```

```
  Kβ : ∀{u f} → K · u · f ≡ u
  Sβ : ∀{f g u} → S · f · g · u ≡ f · u · (g · u)
```

Code 2.1: A model of combinator calculus in Agda

- _·_ : Binary Operation, a function that represents the application of one term to another.
- K: A term representing the K combinator.
- S: A term representing the S combinator.
- Kβ and Sβ: The term equations representing the K-Reduction and S-Reduction

2.3 Combinator birds

In this section, we assume an arbitrary model of combinator calculus.

For our examples, we use the terminology and some examples from the book by Smullyan [13], where combinators are represented by different birds.

2.3.1 Identity bird

Firstly, we have the identity bird - I combinator, which allows us to express the identity function ($f(u) = u$). We define I as a term using our built-in combinators S and K, and then we prove its computation rule using the equations Sβ and Kβ. The proof consists of three steps: the first step is by definition (witnessed by refl); the second step uses Sβ; the third step applies Kβ.

```
I : Tm
I = S · K · K
Iβ : ∀{u} →I · u ≡ u
Iβ {u} =
  (I · u)
            ≡⟨ refl ⟩
  (((S · K) · K) · u)
            ≡⟨ Sβ ⟩
  ((K · u) · (K · u))
            ≡⟨ Kβ ⟩
  refl
```

Code 2.2: I combinator

2.3.2 Bluebird

Now, we can introduce the concept of function composition. Given two functions f and g, we aim to express their composition such that $h(x) = f(g(x))$. The B combinator can effectively achieve this goal.

When we want to apply an equation to only part of a term rather than the entire expression, we consider using *cong*(congruence). Congruence refers to the property of an equivalence relation that is preserved under certain operations.

```
cong : ∀ {A B} {m n : A} →(f : A → B) → m ≡ n → f m ≡ f n
```

Code 2.3: Congruence

```
B : Tm
B = S · (K · S) · K
Bβ : ∀{f g u} →B · f · g · u ≡ f · (g · u)
Bβ {f}{g}{u} =
  (B · f · g · u)
            ≡⟨ refl ⟩
  (S · (K · S) · K · f · g · u)
            ≡⟨ cong (λ z →z · g · u) Sβ ⟩
  (K · S · f · (K · f) · g · u)
            ≡⟨ cong (λ z →z · (K · f) · g · u) Kβ ⟩
  (S · (K · f) · g · u)
             ≡⟨ Sβ ⟩
  (K · f · u · (g · u))
             ≡⟨ cong (λ z →z · (g · u)) Kβ ⟩
  refl
```

Code 2.4: B combinaton

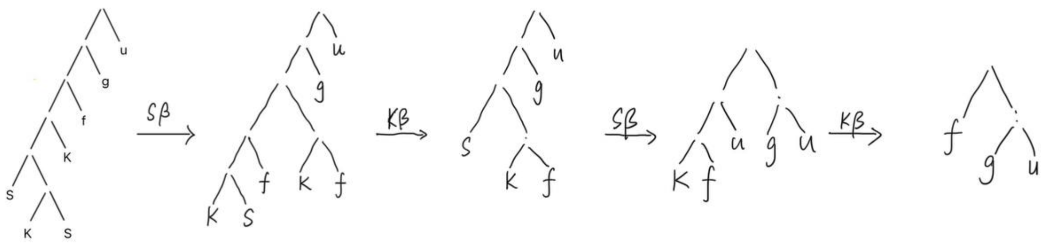

Figure 2.3: The changes in the structure when proving Bβ

2.3.3 Cardinal

Let's explore the concept of argument swapping. Given two functions f and g, and an argument u, we sometimes need to swap the order of g and u in their application to f. The C combinator adeptly facilitates this manipulation, allowing us to express $h(x,y) = f(y,x)$ by rearranging the inputs such that $C \cdot f \cdot u \cdot g \equiv f \cdot g \cdot u$.

```
C : Tm
C = S · (B · B · S) · (K · K)
Cβ : ∀{f g u} →C · f · u · g ≡ f · g · u
Cβ {f}{g}{u} =
  (C · f · u · g)
            ≡⟨ refl ⟩
  (S · (B · B · S) · (K · K) · f ·  u · g)
            ≡⟨ cong (λ z →z · u · g) Sβ ⟩
  (B · B · S · f · (K · K · f) ·  u · g)
            ≡⟨ cong (λ z →z · (K · K · f) ·  u · g) Bβ ⟩
  (B · (S · f) · (K · K · f) ·  u · g)
            ≡⟨ cong (λ z →z  · g) Bβ ⟩
  (S · f · ((K · K · f) ·  u)  · g)
            ≡⟨ cong (λ z →(S · f) · (z · u) · g) Kβ ⟩
  (S · f · (K ·  u) · g)
            ≡⟨ Sβ ⟩
  (f · g ·  (K ·  u · g))
            ≡⟨ cong (λ z →f · g · z) Kβ ⟩
  refl
```

Code 2.5: C combinator

2.3.4 Why bird

Next, we examine recursive functions and their computation. Recursive functions are those that call themselves from within their own code[14]. To express recursion algebraically, we need a mechanism that allows a function to reference itself. This is where the Y combinator [15], also known as the fixed-point combinator, comes into play.

The Y combinator enables the definition of recursive functions in a language that does not natively support recursion.

Before we find the why bird (Y combinator), we'd better find the mockingbird (M combinator) and the lark (L combinator). Otherwise, we need to catch a lot of SK combinators as in this expression $Yf = f(Yf)$. The normal form of Y combinator is (((SS)K)((S(K((SS)(S((SS)K)))))K).

```
M : Tm
M = S · I · I
Mβ : ∀{f} →M · f ≡ f · f
Mβ {f} =
   (M · f)
            ≡⟨ refl ⟩
   (S · I · I · f)
            ≡⟨ Sβ ⟩
   ((I · f) · (I · f))
            ≡⟨ cong (λ z →z · z) Iβ ⟩
   refl
```

Code 2.6: M combinator

```
L : Tm
L = C · B · M
Lβ : ∀{f g} →L · f · g ≡ f · ( g · g )
Lβ {f}{g}=
    (L · f · g)
            ≡⟨ refl ⟩
    (C · B · M · f · g)
            ≡⟨ cong (λ z →z · g) Cβ ⟩
    (B ·  f · M · g)
            ≡⟨ Bβ ⟩
    (f · (M · g))
            ≡⟨ cong (λ z →f · z) Mβ ⟩
```

```
    refl
```

Code 2.7: L combinator

```
Y : Tm
Y = S · L · L
Yβ : ∀{f} →Y · f ≡ f · (Y · f)
Yβ {f} =
    (Y · f)
            ≡⟨ refl ⟩
    (S · L · L · f)
            ≡⟨ Sβ ⟩
    (L · f · (L · f))
            ≡⟨ Lβ ⟩
    (f · ((L · f) · (L · f)))
            ≡⟨ cong ( λ z →f · z) (sym Sβ) ⟩
    refl
```

Code 2.8: Y combinator

Once we have the Y combinator, we can define more recursive functions. e.g. The factorial function.

For a quick overview of additional combinator birds and their corresponding lambda expressions, see [16].

2.3.5 Church encoding

Church encoding is a technique used in lambda calculus to represent computations. Specifically, it introduces Church numerals, which are representations of natural numbers in lambda notation. By turning numbers and basic arithmetic operations into functions, this encoding allows numbers to be manipulated within lambda calculus entirely through function applications.

Here are some examples of Church encoding, it's a bit different from the form of Church number in Wikipedia [17] (the arguments are in the other order), but they express the same meaning. For instance, natural number n is given by applying n times s (successor) to z (zero). Here, n,s and z are meta variables.

```
zero' : Tm
zero' = K
zeroβ : ∀{z s} →zero' · z · s ≡ z
```

```
zeroβ = Kβ
one : Tm
one = C · I
oneβ : ∀{z s} →one · z · s ≡ s · z
oneβ {z}{s} =
    (one · z · s)
            ≡⟨ refl ⟩
    (C · I · z · s)
            ≡⟨ Cβ ⟩
    (I · s · z)
            ≡⟨ cong (λ x →x · z) Iβ ⟩
    refl
```

Code 2.9: Zero and one

The lambda expression of suc is $\lambda nzs.s(nzs)$. We can also find this pattern by analyzing the expression Church number. It applies one more s to the previous number.

```
succ : Tm
succ =  B · (S · I)
sucβ : ∀{n z s} →succ · n · z · s ≡ s · (n · z · s)
sucβ {n}{z}{s}=
    (succ · n · z · s)
            ≡⟨ refl ⟩
    (B · (S · I) · n · z · s)
            ≡⟨ cong (λ x →x · s ) Bβ ⟩
    (S · I · (n · z) · s)
            ≡⟨ Sβ ⟩
    (I · s · (n · z · s))
            ≡⟨ cong (λ x →x · (n · z · s) ) Iβ ⟩
    refl
```

Code 2.10: Suc

The lambda expression of isZero is λ n. n true (λ x. false) and it could be R(K false)(T true) in combinator calculus. The true is the same as K and it computes as true $\cdot\ x\ \cdot\ y = x$; false is the same as S $\cdot$ K, computing as false $\cdot\ x\ \cdot\ y = y$. Then we can translate the lambda expression to combinators iszero = R $\cdot$ (K $\cdot$ (S $\cdot$ K)) $\cdot$ (C $\cdot$ I $\cdot$ K) and prove the two properties of isZero. If the number is zero, return true; If the number is (suc n), return false.

```
R : Tm
R = B · B · one
Rβ : ∀{n z s} →R · n · z · s ≡ z · s · n
Rβ {n}{z}{s}=
  cong (λ x → x · z · s) Bβ ■
  cong (λ x → x) Bβ ■
  cong (λ x → x) oneβ

iszero : Tm
iszero = R · (K  · (S · K)) · (C · I · K)
iszeroβ' : iszero · zero' ≡ K
iszeroβ' =
    (iszero · zero')
            ≡⟨ refl ⟩
    (R · (K  · (S · K)) · (C · I · K) · K)
            ≡⟨ Rβ ⟩
    (C · I · K · K · (K  · (S · K)))
            ≡⟨ cong (λ z →z · (K · (S · K)) ) Cβ ⟩
    (I · K · K · (K  · (S · K)))
            ≡⟨ cong (λ z →z · K · (K · (S · K)) ) Iβ ⟩
    (K · K · (K · (S · K)))
            ≡⟨ zeroβ ⟩
    refl
iszeroβ'' : ∀ {n} →iszero · (succ · n) ≡ S · K
iszeroβ'' {n} =
    (iszero · (succ · n))
            ≡⟨ refl ⟩
    (R · (K  · (S · K)) · (C · I · K) · (succ · n))
            ≡⟨ Rβ ⟩
    (C · I · K · (succ · n) · (K  · (S · K)))
            ≡⟨ cong (λ z →z · (K · (S · K)) ) Cβ ⟩
    (I · (succ · n) · K · (K  · (S · K)))
            ≡⟨ cong (λ z →z · K · (K · (S · K)) ) Iβ ⟩
    (succ · n · K · (K  · (S · K)))
            ≡⟨ sucβ ⟩
    (K  · (S · K) ·  (n · K · (K  · (S · K))))
            ≡⟨ Kβ ⟩
    refl
```

Code 2.11: IsZero

Chapter 3

Models

3.1 Syntax

The syntax of combinator calculus forms a model with the universal property saying that there is a unique homomorphism (function preserving application, K and S) into any model. It is called the initial model or the free model over the empty set. It can be constructed as binary trees (figure 2.1) with S or K at the leaves, quotiented as follows.

This viewpoint is explained in the course notes by Kaposi [18].

The unique morphism from the syntax to a model M is denoted by the double bracket operation computing as follows.

```
⟦_⟧ : Syn.Tm → M.Tm
⟦ Syn.K ⟧ = M.K
⟦ Syn.S ⟧ = M.S
⟦ u Syn.· v ⟧ = ⟦ u ⟧ M· ⟦ v ⟧
```

Figure 3.1: Initial model

3.2 Trivial model

Is there any finite model in combinator calculus? If the number of element is one, then we find our trivial model (just as every algebraic theory has a trivial model).

```
trivial : Model
trivial = record { Tm = ⊤ ; _·_ = λ _ _ →tt ; K = tt ; S = tt ; Kβ =
    refl ; Sβ = refl }
```

Code 3.1: Trivial model of the SK combinator calculus

The terms of the trivial model are represented by a singleton set containing only the element tt, and all operations within this model yield tt as their result. Every equation between terms holds because the only element of Tm is tt.

3.3 No two-element model

What happens when the type of Tm is restricted to a two-element set? For instance, consider Tm defined as Bool, which has elements t(true) and f(false), or as Fin 2, which includes elements (fzero and fsuc fzero). Since these sets are isomorphic, either can be used to formalize this scenario.

Using the combinators K and S, we can define the terms proj1, proj2, and proj3, for which $\mathrm{proj}_i \cdot u_1 \cdot u_2 \cdot u_3 = u_i$ (the i-th argument) holds.

```
proj1 = (B · K) · K --∀ {a b c} →((proj1 · a) · b) · c ≡ a
proj2 = K · K        --∀ {a b c} →((proj2 · a) · b) · c ≡ b
proj3 = K · (K · I) --∀ {a b c} →((proj3 · a) · b) · c ≡ c
```

Code 3.2: Proj1-3

Suppose that the set {t, f} forms a combinator algebra. Based on the pigeonhole principle, two of these projections must coincide; Specifically, permutations and combinations without duplication give us all possible cases of equivalence: proj1 ≡ proj2, proj1 ≡ proj3, and proj2 ≡ proj3. Our goal is to prove for every case we can find t ≡ f which produces the contradiction. Therefore, we will select appropriate terms to illustrate this contradiction for each case.

The three proj functions are the projections coming from the Church encoding of ternary products.

- Case1: proj1 ≡ proj2, then proj1 · t · f · f ≡ proj2 · t · f · f → t ≡ f.
- Case2: proj1 ≡ proj3, then proj1 · t · f · f ≡ proj3 · t · f · f → t ≡ f.
- Case3: proj2 ≡ proj3, then proj2 · f · t · f ≡ proj3 · f · t · f → t ≡ f.

Once we prove all these cases lead to true ≡ false, we can produce the bottom. Because the constructors of any same type are disjoint. After that, we conclude that there is no such model whose term is two-element type.

```
  -- combining fromPigeon and case1', case2', case3'
```

```
  contra : true ≡ false
  contra with fromPigeon
  contra | inl (inl x) = case1 x
  contra | inl (inr x) = case2 x
  contra | inr x = case3 x

  bot : ⊥
  bot with contra
  bot | ()

notbool : (m : Model) → let module m = Model m in m.Tm ≡ Bool →
notbool m refl = notBoolModel.bot _·_ K S Kβ Sβ
  where
    open Model m
```

Code 3.3: No two-element model

The entire formalization see: Code A.1.

3.4 No finite model

First, we need to express that the term is any finite set. In Agda, we have a dependent type Fin, representing a finite set of natural numbers smaller than a given number n.

This is the definition of Fin in Agda.

```
data Fin : ℕ →Set where
  fzero : {n : ℕ} →Fin (suc n)
  fsuc  : {n : ℕ} →Fin n → Fin (suc n)
```

Code 3.4: Fin

The constructor fzero is used to indicate the smallest element, zero, in any non-empty finite set of size suc n (which is n+1). The constructor fsuc allows incrementing an existing element from Fin n to represent the next natural number in Fin (suc n), effectively building the set recursively.

Now it's time to prove there is no nontrivial finite model. We can set Tm = Fin n to express the term could be any number from 0 to (n-1). We set n to be suc (suc m) for some arbitrary m (we assume that there are at least two terms in our model). Even if there are infinitely many natural numbers, inside the model, our set of terms is always finite. It shows the expressiveness of dependent type and also lays the foundation for formalizing various mathematical theorems.

Following the step in proving no two-element model, we have to define our proj to get the i-th term.

The formula of proj is :

$$(\text{Proj } n\ i) \cdot n_0 \cdot \ldots \cdot n_n = n_i$$

The lambda expression of proj is:

$$\lambda n_0 \ldots n_n . n_i$$

Before we summarize the rules of the i-th item of n elements, we hope to observe more and obtain their combinatorial algebra. Let's continue to analyze the 3-element cases. Here are the the formulas of projs and their corresponding lambda expressions.

$$\begin{array}{ll} \forall a\,b\,c \to (\text{Proj } 3\ \ 0) \cdot a \cdot b \cdot c \equiv a & \lambda abc.a \\ \forall a\,b\,c \to (\text{Proj } 3\ \ 1) \cdot a \cdot b \cdot c \equiv b & \lambda abc.b \\ \forall a\,b\,c \to (\text{Proj } 3\ \ 2) \cdot a \cdot b \cdot c \equiv c & \lambda abc.c \end{array}$$

Using Kiselyov Combinator Translation[10] with Eta-optimization, we can translate the lambda term into the combinator term accurately. We also implemented a model to convert lambda calculus into the normal form of SK combinator calculus, see Calculator.

The situation for 4-element set is similar. We summarize the combinators obtained so far in the table. Then we can observe the table and make guesses about the patterns of table generation.

Table 3.1: Combinators in proj

n	0	1	2	3
1	I			
2	BKI	KI		
3	BK(BKI)	K(BKI)	K(KI)	
4	BK(BK(BKI))	K(BK(BKI))	K(K(BKI))	K(K(KI))

In this table, we can see the following changes:

$$I \to BKI \to BK(BKI) \to BK(BK(BKI))$$

The first row starts with I and keeps adding BK to the result of the previous row.

$$I \to KI \to K(KI) \to K(K(KI))$$

The pattern in the other direction could be found in the diagonals. It's simpler than the pattern in the first line, only add a K combinator to the previous result. It's also correct for other diagonals, not only in the main diagonal.

We can write our proj function based on these two patterns. The first argument of proj is the number of elements and the second argument represents the index. And we do induction on n and i.

```
proj : ∀(n : ℕ) →Fin (suc n) → Tm
proj zero fzero = I
proj (suc n) fzero = B · K · proj n fzero
proj (suc n) (fsuc x) = K · proj n x
```

Code 3.5: Proj

proj (suc n) fzero = B $\cdot$ K $\cdot$ proj n fzero is the pattern in the first row, and we call proj n fzero recursively; proj (suc n) (fsuc x) = K $\cdot$ proj n x is the pattern on diagonal, after we construct the outside K, we call proj n x.

How can we promise our proj is correct? Given a vector to represent the n elements, we want to prove the property of proj. this proj-beta rule is saying proj n x $\cdot$s us will give same result as lookup the x-th element from the vector us.

```
projβ : (n : ℕ)(x : Fin (suc n))(us : Vec Tm (suc n)) →
        proj n x ·s us ≡ lookup us x
projβ zero fzero (u :: []) = Iβ
projβ (suc n) fzero (u :: u' :: us) =
        (B · K · (proj n fzero) · u · u' ·s us)
            ≡ cong (λ x →(x · u') ·s us) Bβ
        (((K · (proj n fzero · u)) · u') ·s us)
            ≡ cong (_·s us) Kβ
        ((proj n fzero · u) ·s us)
            ≡ projβ n fzero (u :: us)
        refl

```

```
projβ (suc n) (fsuc x) (u :: u' :: us) =
    (K · (proj n x)  · u · u' ·s us)
        ≡ cong (λ x →x · u' ·s us) Kβ
    (proj n x · u' ·s us)
        ≡ projβ n x (u' :: us)
    refl
```

Code 3.6: The beta rule of proj

The next step is to generate the (us : Vec Tm (suc n)) which fulfills the elements with given indexes(i,j) are different (Like 0 and 1). Other elements are not important, so we can just set the j-th element to 1, others return 0.

```
almost0 : {j : Fin (suc n)} → Fin (suc n) → Tm
almost0 {j} x with x eq? j
... | inr b = fzero  --no
... | inl a = fsuc fzero --yes

flatten : ∀{ι}{A : Set ι}{n : ℕ} →(Fin n →A) →Vec A n
flatten {n = zero}  f = []
flatten {n = suc n} f = f fzero :: flatten {n = n} (f ∘ fsuc)

us : {j : Fin (suc n)} → Vec Tm (suc n)
us {j} = flatten (almost0 {j})
```

Code 3.7: Vector us

Now we have our vector us, and these are the properties of us.

$$\text{almost0i} : \forall i\, j : \text{Fin}\ (\text{suc}\ n) \rightarrow i \neq j \rightarrow \text{almost0}\{j\}i \equiv \text{fzero}$$
$$\text{almost0j} : \forall j : \text{Fin}\ (\text{suc}\ n) \rightarrow \text{almost0}\{j\}j \equiv (\text{fsuc fzero})$$

```
flattenval : ∀{ι}{A : Set ι}{n : ℕ}(f : Fin n →A)(x : Fin n) →
             f x ≡ lookup (flatten f) x
flattenval {n = suc n} f fzero    = refl
flattenval {n = suc n} f (fsuc x) = flattenval {n = n} (f ∘ fsuc) x
```

Code 3.8: flattenval

In mathematics, the pigeonhole principle posits that if we place n items into m containers and n is greater than m, at least one container will necessarily contain more than one item.

Here is the expression from Agda standard library[19]:

$$\text{pigeonhole} : \forall\{m,n\} \to m < n \to (f : \text{Fin } n \to \text{Fin } m) \to \exists_2 \lambda i\ j \to i \neq j \times f(i) \equiv f(j)$$

Our recurrence of the pigeonhole principle proof, see: Code C

Given a proposition where m is less than n, for any function f that maps from Fin n to Fin m, there exists at least one pair of distinct elements (i,j) from finite set n such that i is not equal to j, yet the function f maps both i and j to the same value in finite set m.

In our model, the smaller set is Fin n representing the kinds of elements, the larger set is Fin (suc n)) representing how many positions for these elements. So the function f for our model mapping the index to the term is:

```
f : Fin (suc n) → Fin n
f x = proj n x
```

Code 3.9: f

Here is the entire procession to get the proposition $0 \equiv 1$ using all the tools we have.

```
contra : fzero ≡ fsuc fzero
contra with fromPigeon
... | i , j , i≠j , fi=fj =
  fzero
          ≡ sym (almost0i i≠j )
  almost0 i
          ≡ flattenval almost0 i
  lookup us i
          ≡ sym (projβ (suc (suc m)) i (us {j}))
  ((proj (suc (suc m)) i) ·s us)
          ≡ cong (λ x →x ·s us ) fi=fj
  ((proj (suc (suc m)) j) ·s us)
         ≡ (projβ (suc (suc m)) j (us {j}))
  lookup us j
        ≡ sym (flattenval almost0 j)
  almost0 j
        ≡ (almost0j {j} )
  refl
```

Code 3.10: Proof

With the contradict $1 \equiv 0$, we can reach the bottom. Then we can conclude that there is no finite model in SK combinator calculus.

```
notfinite : (m : Model){n : ℕ} →let module m = Model m in m.Tm ≡ (Fin (suc (suc n))) →
notfinite record { Tm = .(Fin (suc (suc n))) ; _·_ = _·_ ; K = K ; S = S ; Kβ = Kβ ; Sβ = Sβ } {n} refl =
  notFiniteModel2.bot n _·_ K S Kβ Sβ
```

Code 3.11: Not finite

Chapter 4

Summary and results

This thesis has explored some foundational algebraic properties of the SK combinator calculus, specifically formalizing the non-existence of non-trivial finite models.

In the future, we would like to formalize infinite models and study their properties. We would also like to study the relationship of combinator calculus and lambda calculus, in particular, what is the notion of lambda calculus, which is equivalent to non-extensional combinator calculus: What is the combinator calculus equivalent to lambda calculus without Eta-rule? These questions were not answered in the algebraic setting before.

Acknowledgements

I want to express my gratitude to ELTE Informatikai Kar for their unwavering support throughout my study. Special thanks are due for the funding provided under the HALL_23 Scholarship and the consistently conducive academic environment that greatly facilitated my study and research activities.

I would like to express my deepest appreciation to my supervisor Ambrus Kaposi, for his invaluable insights into this field and for his patience in helping me grasp concepts that were initially beyond my understanding. I am also grateful to Zsók Viktória for sparking my interest in functional programming.

Special thanks to Szumi Xie, Viktor Bense, and Márton Petes of the Type Theory group, who have each been pivotal in my academic journey. I'm especially grateful to Marci for teaching me Agda this semester, which greatly enhanced my programming skills. I am equally grateful to Szumi and Viktor for their consistent guidance and encouragement, which have been instrumental in steering my thinking in the right direction whenever I felt lost and in helping me tackle my challenges together.

During the process of finishing this paper, I received guidance and valuable suggestions from many. I wish to extend my sincere thanks to Reinhard Wilhelm for carefully reading my thesis again and providing enlightening comments about writing.

It has been both my pleasure and my fortune to collaborate with each of you.

Appendix A

No bool model

```
module notBoolModel

  (_·_ : Bool → Bool → Bool)
  (K S : Bool)
  (Kβ : ∀{u f} →(K · u) · f ≡ u)
  (Sβ : ∀{f g u} →((S · f) · g) · u ≡ (f · u) · (g · u))

  where

  Tm = Bool
  I : Tm
  I = S · K · K
  Iβ : ∀{u} →I · u ≡ u
  Iβ {u} =
    (I · u)
               ≡⟨ refl ⟩
    (((S · K) · K) · u)
               ≡⟨ Sβ ⟩
    ((K · u) · (K · u))
               ≡⟨ Kβ ⟩
    refl

  B : Tm
  B = S · (K · S) · K
  Bβ : ∀{f g u} →B · f · g · u ≡ f · (g · u)
  Bβ {f}{g}{u} =
    (B · f · g · u)
               ≡⟨ refl ⟩
```

```
  (S · (K · S) · K · f · g · u)
            ≡⟨ cong (λ z →z · g · u) Sβ ⟩
  (K · S · f · (K · f) · g · u)
            ≡⟨ cong (λ z →z · (K · f) · g · u) Kβ ⟩
  (S · (K · f) · g · u)
             ≡⟨ Sβ ⟩
  (K · f · u · (g · u))
             ≡⟨ cong (λ z →z · (g · u)) Kβ ⟩
  refl

fst : Tm  -- proj 0
fst = (B · K) · K
fstβ : ∀ {a b c} →((fst · a) · b) · c ≡ a
fstβ {a}{b}{c} =
  cong  (λ z →(z · b) · c) Bβ' ■
  cong  (λ z →z · c) Kβ ■
  Kβ
snd : Tm -- proj 1
snd = K · K
thd : Tm -- proj 2
thd = K · (K · I)
sndβ : ∀ {a b c} →((snd · a) · b) · c ≡ b
sndβ {a}{b}{c} =
  cong  (λ z →(z · b) · c) Kβ ■
  Kβ
thdβ : ∀ {a b c} →((thd · a) · b) · c ≡ c
thdβ {a}{b}{c} =
  cong  (λ z →(z · b) · c) Kβ ■
  cong  (λ z →z · c) Kβ ■
  Iβ

pigeonHoleBool : (a b c : Tm) → a ≡ b ⊔ a ≡ c ⊔ b ≡ c
pigeonHoleBool true true true = inl (inl refl)   -- All true
pigeonHoleBool false false false = inl (inl refl) -- All false
pigeonHoleBool true true false = inl (inl refl)  -- a ≡ b
pigeonHoleBool true false true = inl (inr refl)  -- a ≡ c
pigeonHoleBool false false true = inl (inl refl) -- a ≡ b
pigeonHoleBool false true false = inl (inr refl) -- a ≡ c
pigeonHoleBool true false false = inr refl -- b ≡ c
pigeonHoleBool false true true = inr refl

```

```
case1 : fst ≡ snd →((fst · true) · false) · false ≡ ((snd · true) ·
    false) · false
case1 = λ x →cong (λ f →((f · true) · false) · false) x
case1' : fst ≡ snd →true ≡ false
case1' x = trans (sym fstβ)  (trans (case1 x) sndβ)

case2 : fst ≡ thd →((fst · true) · false) · false ≡ ((thd · true) ·
    false) · false
case2 = λ x →cong (λ f →((f · true) · false) · false) x
case2' : fst ≡ thd →true ≡ false
case2'  x = trans (sym fstβ) (trans (case2 x) thdβ )

case3 : snd ≡ thd →((snd · false) · true) · false ≡ ((thd · false) ·
     true) · false
case3 =   λ x →cong (λ f →((f · false) · true) · false) x
case3' : snd ≡ thd →true ≡ false
case3' x =  trans (sym sndβ) (trans (case3 x) thdβ)

fromPigeon : fst ≡ snd ⊔ fst ≡ thd ⊔ snd ≡ thd
fromPigeon  = pigeonHoleBool fst snd thd
contra : true ≡ false
contra with fromPigeon
contra | inl (inl x) = case1' x
contra | inl (inr x) = case2' x
contra | inr x = case3' x
-- comnbining fromPigeon and case1', case2', case3'
bot :
bot with contra
bot | ()
```

Code A.1: notboolmodel

```
notbool : (m : Model) → let module m = Model m in m.Tm ≡ Bool →
notbool m refl = notBoolModel.bot _·_ K S Kβ Sβ
  where
    open Model m
```

Code A.2: Conclusion : no bool model

Appendix B

No finite model

```
module notFiniteModel
  (m : ℕ)
  (_·_ :  Fin (suc (suc m)) → Fin (suc (suc m)) → Fin (suc (suc m)))
  (let infixl 5 _·_; _·_ = _·_)
  (K S :  Fin (suc (suc m)))
  (Kβ : {u f : Fin (suc (suc m))} →(K · u) · f ≡ u)
  (Sβ : {f g u : Fin (suc (suc m))} → ((S · f) · g) · u ≡ (f · u) · (g
     · u))

  where
    n = suc (suc m)
    Tm = Fin n
    infixl 5 _·s_
    I : Tm
    I = S · K · K
    Iβ : ∀{u} →I · u ≡ u
    Iβ {u} =
      (I · u)
                 ≡ refl
      (((S · K) · K) · u)
                 ≡ Sβ
      ((K · u) · (K · u))
                 ≡ Kβ
      refl


    B : Tm
    B = S · (K · S) · K
```

```
Bβ : ∀{f g u} →B · f · g · u ≡ f · (g · u)
Bβ {f}{g}{u} =
  (B · f · g · u)
            ≡ refl
  (S · (K · S) · K · f · g · u)
            ≡ cong (λ z →z · g · u) Sβ
  (K · S · f · (K · f) · g · u)
            ≡ cong (λ z →z · (K · f) · g · u) Kβ
  (S · (K · f) · g · u)
            ≡ Sβ
  (K · f · u · (g · u))
             ≡ cong (λ z →z · (g · u)) Kβ
   refl

_·s_ : Tm → {n : ℕ} →Vec Tm n → Tm
_·s_ t {zero}  [] = t
_·s_ t {suc n} (u :: us)  = t · u ·s us

lookup : ∀{ι}{A : Set ι}{n : ℕ} →Vec A n →Fin n →A
lookup (x :: xs) fzero    = x
lookup (x :: xs) (fsuc i) = lookup xs i

proj : ∀(n : ℕ) →(Fin (suc n)) →Tm
proj zero fzero = I
proj (suc n) fzero = B · K · proj n fzero
proj (suc n) (fsuc x) = K · proj n x

m≤sucm : {m : ℕ} →m < (suc m)
m≤sucm {zero} = s≤s z≤n
m≤sucm {suc m} = s≤s m≤sucm

almost0 : {j : Fin (suc n)} → Fin (suc n) → Tm
almost0 {j} x with x eq? j
... | inr b = fzero
... | inl a = fsuc fzero

flatten : ∀{ι}{A : Set ι}{n : ℕ} →(Fin n →A) →Vec A n
flatten {n = zero}  f = []
flatten {n = suc n} f = f fzero :: flatten {n = n} (f ∘ fsuc)

us : {j : Fin (suc n)} → Vec Tm (suc n)
```

```
us {j} = flatten (almost0 {j})

projβ : (n : ℕ)(x : Fin (suc n))(us : Vec Tm (suc n)) →proj n x ·s
    us ≡ lookup us x
projβ zero fzero (u :: []) = Iβ
projβ (suc n) fzero (u :: u' :: us) =
                (B · K · (proj n fzero) · u · u' ·s us)
                            ≡ cong (λ x →(x · u') ·s us) Bβ
                (((K · (proj n fzero · u)) · u') ·s us)
                            ≡ cong (_·s us) Kβ
                ((proj n fzero · u) ·s us)
                            ≡ projβ n fzero (u :: us)
                refl

projβ (suc n) (fsuc x) (u :: u' :: us) =
                (K · (proj n x)  · u · u' ·s us)
                            ≡ cong (λ x →x · u' ·s us) Kβ
                (proj n x · u' ·s us)
                            ≡ projβ n x (u' :: us)
                refl

flattenval : ∀{ι}{A : Set ι}{n : ℕ}(f : Fin n →A)(x : Fin n) →
            f x ≡ lookup (flatten f) x
flattenval {n = suc n} f fzero     = refl
flattenval {n = suc n} f (fsuc x) = flattenval {n = n} (f ∘ fsuc) x


almost0i : ∀{i j : Fin (suc n)} →i ≠j →almost0 {j} i ≡ fzero
almost0i {i}{j} i≠j with i eq? j
... | inr b = refl
... | inl a with i≠j a
... | ()

almost0j :  ∀{j : Fin (suc n)} →almost0 {j} j ≡ (fsuc fzero)
almost0j {j} with j eq? j
... | inl a = refl
... | inr ¬j with ¬j refl
... | ()

f : Fin (suc n) → Fin n
f x = proj n x -- proj (suc n) x
```

```
  fromPigeon : ∃ λ i j →i ≢ j × f i ≡ f j
  fromPigeon  = pigeonhole m≤sucm f

  contra : fzero ≡ fsuc fzero
  contra with fromPigeon
  ... | i , j , i≢j , fi=fj =
    fzero
             ≡ sym (almost0i i≢j )
    almost0 i
             ≡ flattenval almost0 i
    lookup us i
             ≡ sym (projβ (suc (suc m)) i (us {j}))
    ((proj (suc (suc m)) i) ·s us)
             ≡ cong (λ x →x ·s us ) fi=fj
    ((proj (suc (suc m)) j) ·s us)
            ≡ (projβ (suc (suc m)) j (us {j}))
    lookup us j
           ≡ sym (flattenval almost0 j)
    almost0 j
           ≡ (almost0j {j} )
    refl

  bot :
  bot with contra
  bot | ()
```

Code B.1: notfinitemodel

```
notfinite : (m : Model){n : ℕ} →let module m = Model m in m.Tm ≡ (Fin
    (suc (suc n))) →
notfinite record { Tm = .(Fin (suc (suc n))) ; _·_ = _·_ ; K = K ; S =
     S ; Kβ = Kβ  ; Sβ = Sβ  } {n} refl =
  notFiniteModel2.bot n _·_  K  S Kβ  Sβ
```

Code B.2: Conclusion : no finite model

Appendix C

Pigeonhole principle proof

This is a recurrence of the pigeonhole principle proof.

For the complete one see: php.agda.

For the original proof, see: pigeonhole.

```
pigeonhole :   {m n} → m < n → (f : Fin n → Fin m) →
                ∃ λ i j →i ≠j × f i ≡ f j
```

Code C.1: Pigeonhole principle in Agda

```
pigeonhole : ∀ {m n} → m < n → (f : Fin n → Fin m) →
             ∃₂ λ i j → i ≢ j × f i ≡ f j
pigeonhole (s≤s z≤n) f = contradiction (f fzero) λ()
pigeonhole (s≤s (s≤s m≤n)) f with any?' (DecProof (λ x →  (f fzero) ≟ (f (fsuc x))))
... | inl (j , f₀≡fⱼ) = fzero , ((fsuc j) , ((λ {()}) , f₀≡fⱼ))
... | inr f₀≢fₖ with pigeonhole (s≤s m≤n) (λ j → punchOut (f₀≢fₖ  ∘ (j ,_ )) )
... | i , j , i≢j  , fᵢ≡fⱼ = fsuc i , (fsuc j , (i≢j ∘ fsuc-injective , punchOut-injective (f₀≢fₖ  ∘  ((i ,_))) _ fᵢ≡fⱼ))
```

Figure C.1: Pigeonhole principle proof

Bibliography

[1] Moses Schönfinkel. "Über die Bausteine der mathematischen Logik". In: Mathematische Annalen 92 (1924), pp. 305–316. url: https://api.semanticscholar.org/CorpusID:118507515.

[2] Jonathan P. Seldin. "The Logic of Church and Curry". In: Logic from Russell to Church. 2009. url: https://api.semanticscholar.org/CorpusID:28020641.

[3] Alonzo Church. "A formulation of the simple theory of types". In: Journal of Symbolic Logic 5.2 (1940), pp. 56–68. doi: 10.2307/2266170.

[4] Contributors to Wikimedia projects. SKI combinator calculus - Wikipedia. [Online; accessed 13. May 2024]. Mar. 2024. url: https://en.wikipedia.org/w/index.php?title=SKI_combinator_calculus&oldid=1214595603.

[5] J. Roger Hindley and Jonathan P. Seldin. Lambda-Calculus and Combinators: An Introduction. 2nd ed. USA: Cambridge University Press, 2008. isbn: 0521898854.

[6] JanetYin. ttt. [Online; accessed 13. May 2024]. May 2024. url: https://github.com/JanetYin/ttt/blob/master/src/SK/Church_encoding.agda.

[7] Hendrik Pieter Barendregt. The Lambda Calculus: Its Syntax and Semantics. New York, N.Y.: Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., 1981.

[8] Katalin Bimbó. Combinatory Logic: Pure, Applied and Typed. Taylor & Francis, 2011, pp. 138–139.

[9] Thorsten Altenkirch et al. "Combinatory Logic and Lambda Calculus Are Equal, Algebraically". In: 8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023). Ed. by Marco Gaboardi and Femke van Raamsdonk. Vol. 260. Leibniz International Proceedings in

Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 24:1–24:19. isbn: 978-3-95977-277-8. doi: 10.4230/LIPIcs.FSCD.2023.24. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2023.24.

[10] Ben Lynn. Kiselyov Combinator translation, Lambda calculus - Kiselyov Combinator Translation. 13 February 2024. 2024. url: https://crypto.stanford.edu/~blynn/lambda/kiselyov.html.

[11] Ambrus Kaposi. [Online; accessed 13. May 2024]. Apr. 2024. url: https://akaposi.github.io/nyelv.pdf.

[12] The Agda Wiki. [Online; accessed 13. May 2024]. May 2024. url: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[13] Raymond M. Smullyan. To Mock a Mockingbird: And Other Logic Puzzles. New York: Oxford University Press, 1985.

[14] Contributors to Wikimedia projects. Recursion (computer science) - Wikipedia. [Online; accessed 25. May 2024]. Apr. 2024. url: https://en.wikipedia.org/w/index.php?title=Recursion_(computer_science)&oldid=1220097348.

[15] Contributors to Wikimedia projects. Fixed-point combinator - Wikipedia. [Online; accessed 13. May 2024]. Apr. 2024. url: https://en.wikipedia.org/w/index.php?title=Fixed-point_combinator&oldid=1220438428.

[16] Combinator Birds. [Online; accessed 13. May 2024]. May 2024. url: https://www.angelfire.com/tx4/cus/combinator/birds.html.

[17] Contributors to Wikimedia projects. Church encoding - Wikipedia. [Online; accessed 13. May 2024]. Jan. 2024. url: https://en.wikipedia.org/w/index.php?title=Church_encoding&oldid=1195268149.

[18] akaposi / typesystems / src / main.pdf — Bitbucket. [Online; accessed 13. May 2024]. May 2024. url: https://bitbucket.org/akaposi/typesystems/src/master/src/main.pdf.

[19] Data.Fin.Properties. [Online; accessed 13. May 2024]. May 2024. url: https://agda.github.io/agda-stdlib/master/Data.Fin.Properties.html.

List of Figures

List of Tables

List of Codes