Lambda Calculus - Combinators (8A)

Young Won Lim
12/3/24

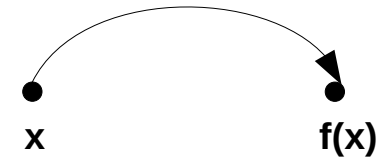Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Fix point (1)

In mathematics, a **fixed point** (**fixpoint**),

also known as an **invariant point**,

is a value that does <u>not</u> <u>change</u> under a given transformation.

Specifically, for functions,

a **fixed point** is an element

that is mapped to itself by the function.

Formally, **c** is a fixed point of a function **f**

if **c** belongs to both the domain and the codomain of **f**, and

  f(c) = c.

x   **f(x)**

c   fixed point   **f(c) = c**

Fix point (2)

For example, if **f** is defined on the real numbers by

$\qquad$ **f**(**x**) = **x**^{2} − 3**x** + 4 ,

then 2 is a fixed point of **f**, because **f**(2) = 2.

Not all functions have fixed points: for example,

f(**x**) = **x** + 1, has no fixed points,

since **x** is never equal to **x** + 1 for any real number.

In graphical terms, a fixed-point **x** means

the point (**x**, **f**(**x**)) is on the line **y** = **x**, or in other words

the graph of **f** has a point in common with that line.

https://en.wikipedia.org/wiki/Fixed_point_(mathematics)

Extensionality (1)

In logic, extensionality, or extensional equality,

refers to principles that judge objects to be equal

if they have the same external properties.

It stands in contrast to the concept of intensionality,

which is concerned with whether

the internal definitions of objects are the same.

https://en.wikipedia.org/wiki/Extensionality

Extensionality (2)

Consider the two functions **f** and **g**

mapping from and to natural numbers,

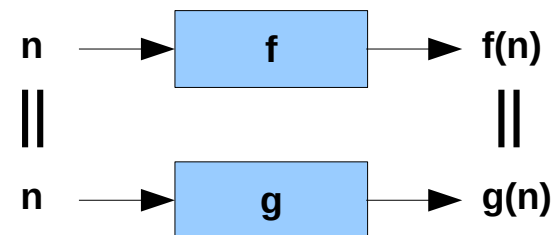defined as follows:

   To find **f(n)**, first <u>add</u> **5** to **n**, then <u>multiply</u> by **2**.   (n + 5)*2

   To find **g(n)**, first <u>multiply</u> **n** by **2**, then <u>add</u> **10**.   2*n + 10

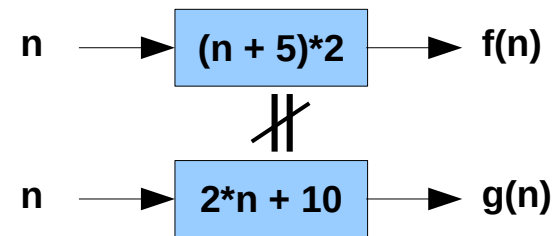These functions are extensionally <u>equal</u>;

given <u>the same input</u>, both functions always <u>produce</u> <u>the same value</u>.

But the <u>definitions</u> of the functions are <u>not</u> <u>equal</u>,

and in that intensional sense the functions are <u>not</u> <u>the same</u>.

extensionally <u>equal</u>

n →⊳ [**f**] ⊳→ **f(n)**

‖   ‖

n →⊳ [**g**] ⊳→ **g(n)**

intensionally <u>inequal</u>

n →⊳ [**(n + 5)*2**] ⊳→ **f(n)**

≠

n →⊳ [**2*n + 10**] ⊳→ **g(n)**

Extensionality (3)

Similarly, in natural language

there are many predicates (relations)

that are intensionally different

but are extensionally identical.

For example, suppose that a town has one person named Joe,

who is also the oldest person in the town.

Then, the two predicates "being called Joe",

and "being the oldest person in this town"

are intensionally distinct,

but extensionally equal

for the (current) population of this town.

https://en.wikipedia.org/wiki/Extensionality

Combinatory Logic

Combinatory logic is a <u>notation</u>

to <u>eliminate</u> the need for quantified variables in mathematical logic.

<u>without</u> using quantified variables

It was introduced by Moses Schönfinke and Haskell Curry,

and has more recently been used in computer science

as a <u>theoretical model</u> of <u>computation</u>

theoretical model of computation

and also as a <u>basis</u> for the design of

functional programming

<u>functional programming</u> languages.

It is based on combinators

combinators

Combinator

combinators were introduced by Schönfinkel in 1920

with the idea of providing an <u>analogous way</u>

– to <u>build up</u> functions

– to <u>remove</u> any mention of variables

– particularly in predicate logic.

A combinator is a higher-order function

that uses <u>only</u> function application

earlier defined combinators

to <u>define</u> a result from its arguments.

Combinators:
define a result by its argument
<u>without</u> free variables

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator Definitions (1)

Combinator : A lambda expression containing <u>no</u> <u>free</u> variables.

the word is usually understood more specifically

to refer to certain combinators of special importance,

in particular the following <u>four</u>:

I = λx . x   **Identity**

K = λx . λy . x   **Constant function**

S = λx . λy . λz . x(z)(y(z))   **Substitution operator**

Y = λf . (λu . f(u(u))) (λu . f(u(u)))

https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/combinator

Combinator informal description (1-1)

Informally, a tree (**xy**) can be thought of

as a function **x** applied to an argument **y**.

When evaluated (i.e., when the function is "applied" to the argument),

the tree "returns a value", i.e., transforms into another tree.

The "function", "argument" and the "value" are

either combinators or binary trees.

If they are binary trees,

they may be thought of as functions too, if needed.
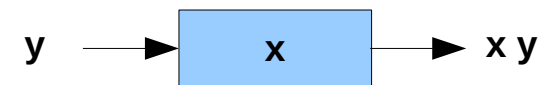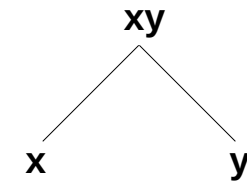
https://en.wikipedia.org/wiki/SKI_combinator_calculus

Combinator informal description (1-2)

Although the most <u>formal representation</u> of the objects in this system

requires binary trees,

for <u>simpler typesetting</u>

they are often represented as parenthesized expressions,

as a <u>shorthand</u> for the tree they represent.

Any subtrees may be parenthesized,

but often only the <u>right-side</u> subtrees are parenthesized,

with left associativity implied for any <u>unparenthesized applications</u>.
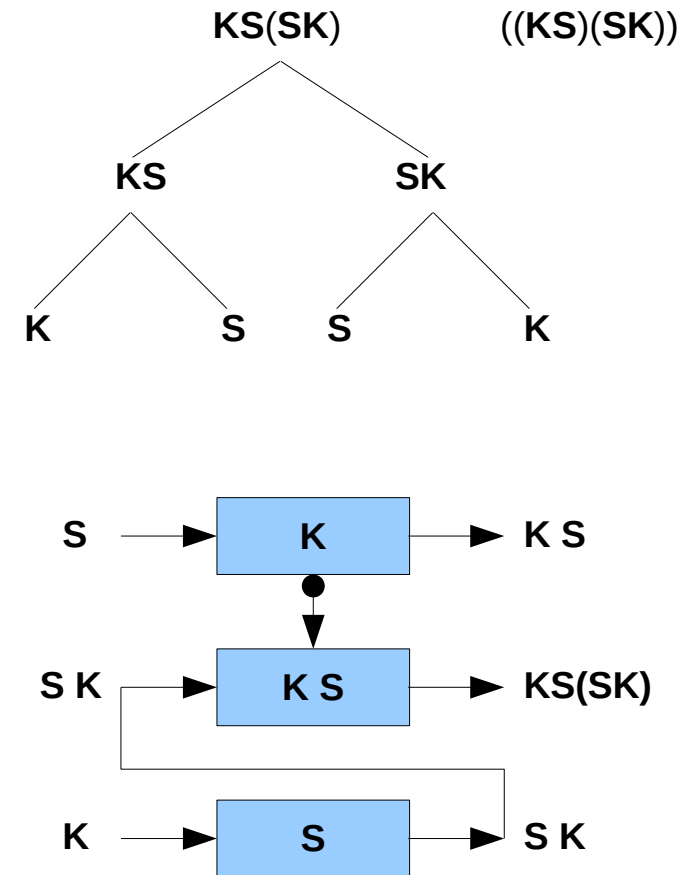
For example, **ISK** means ((**IS**)**K**).
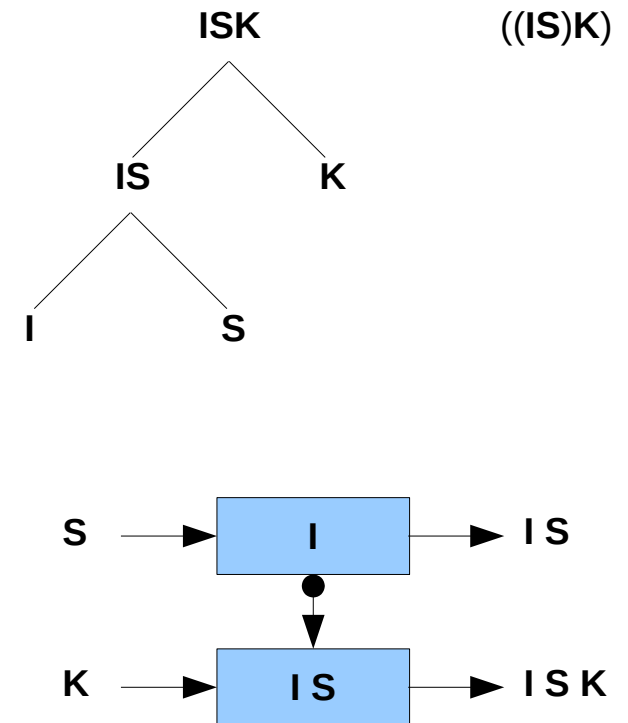
https://en.wikipedia.org/wiki/SKI_combinator_calculus

Combinator informal description (1-3)

a tree whose left subtree is the tree **KS**

and whose right subtree is the tree **SK**

can be written as **KS**(**SK**).

If more explicitness is desired,

the implied parentheses can be included as well: ((**KS**)(**SK**)).

KS(**SK**)   ((**KS**)(**SK**))

I combinator

The evaluation operation is defined as follows:

x, **y**, and **z** represent expressions

made from the functions **S**, **K**, and **I**, and set values:

I returns its argument:

   I x = x

x → [**I**] → x

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (1-1)

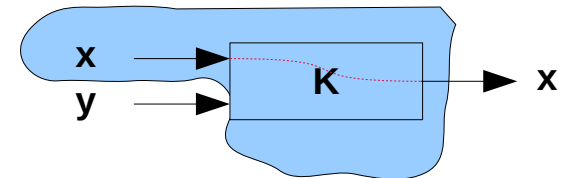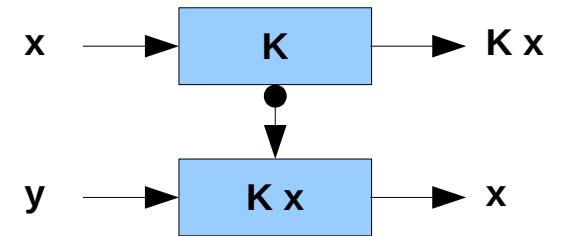The simplest example of a combinator is **I**, the identity combinator, defined by

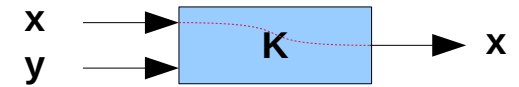$$(I\ x) = x \qquad \text{for all terms } x.$$

K combinator

K, when applied to any argument **x**,

yields a one-argument constant function **K x**,

which, when applied to any argument **y**, returns **x**:

   K x y = x

Another simple combinator is **K**,

which manufactures constant functions:

(K x) is the function which, for any argument, returns **x**, so we say

((K x) y) = x   for all terms **x** and **y**.

Or, following the convention for multiple application,

(K x y) = x

https://en.wikipedia.org/wiki/Combinatory_logic

S combinator

S is a substitution operator.

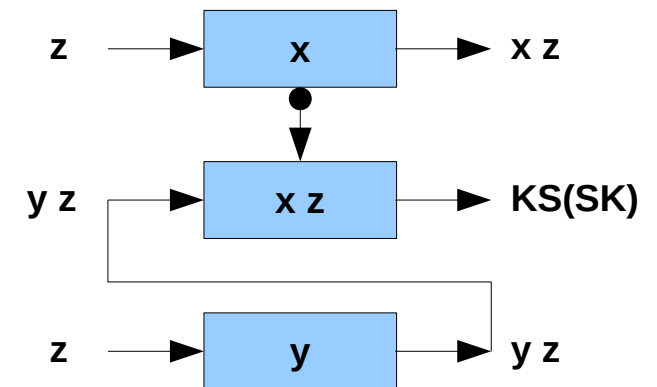takes three arguments (**x y z**)

returns the result of **x z** applied to the result of **y z**

the first argument (**x**) applied to the third (**z**),

which is then applied to the result

of the second argument (**y**) applied to the third (**z**).

S x y z = x z (y z)
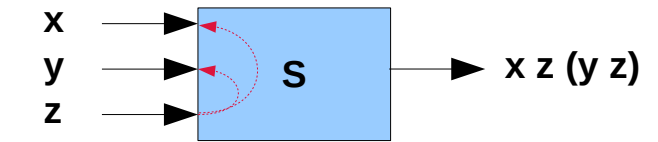
a function of **x z** with the argument **y z**
a function of **x** with the argument **z**
a function of **y** with the argument **z**

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Examples of Combinators (2-1)

A third combinator is **S**, which is a <u>generalized version</u> of application:

(S x y z) = (x z (y z))

S <u>applies</u> **x** to **y**

after first <u>substituting</u> **z** into each of them (**x** and **y**)

x is <u>applied</u> to **y**

inside the environment **z**.

Examples of Combinators (2-2)

Given **S** and **K**, **I** itself is *unnecessary*,

since it can be built from the other two:

   ((S K K) x)

      = (S K K x)

      = (K x (K x))

      = x

for any term **x**.

Combinator informal description (3-1)

SKSK evaluates to **KK**(**SK**) by the S-rule.

Then if we evaluate **KK**(**SK**), we get **K** by the K-rule.

As no further rule can be applied, the computation halts here.

For all trees **x** and all trees **y**,

SKxy will always evaluate to **y** in two steps, **Ky**(**xy**) = **y**,

so the ultimate result of evaluating **SKxy**

will always equal the result of evaluating **y**.

We say that **SKx** and **I** are "functionally equivalent" for any **x**

because they always yield the same result when applied to any **y**.

$$S\ x\ y\ z = x\ z\ (y\ z) \quad \text{S-rule}$$

$$S\ K\ S\ K = K\ K\ (S\ K)$$

$$K\ x\ y = x \quad \text{K-rule}$$

$$K\ K\ (S\ K) = K$$

$$S\ K\ x\ y = K\ y\ (x\ y) = y$$

$$I\ y = y$$

Combinator informal description (3-2)

it can be shown that SKI calculus is <u>not</u> the <u>minimum system</u>

that can fully perform the computations of lambda calculus,

as all occurrences of I in any expression can be replaced

by (**SKK**) or (**SKS**) or (**SK x**) for any **x**,

and the resulting expression will yield the same result.

So the "**I**" is merely <u>syntactic sugar</u>.

Since **I** is <u>optional</u>, the system is also referred

as SK calculus or

SK combinator calculus.

$$S\ K\ K\ y\ = K\ y\ (K\ y)\ = y$$

$$I\ y = y$$

$$S\ K\ S\ y\ = K\ y\ (S\ y)\ = y$$

$$I\ y = y$$

$$S\ K\ x\ y\ = K\ y\ (x\ y)\ = y$$

$$I\ y = y$$

https://en.wikipedia.org/wiki/SKI_combinator_calculus

Note that although **((S K K) x) = (I x)** for any **x**,

(S K K) itself is <u>not</u> equal to **I**.

We say the terms are extensionally equal.

Extensional equality captures the <u>mathematical notion</u>

of the equality of functions:

that two functions are <u>equal</u>

if they always <u>produce</u> the <u>same results</u> for the <u>same arguments</u>.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (3-2)

In contrast, the terms themselves,

together with the reduction of primitive combinators,

capture the notion of intensional equality of functions:

that two functions are equal

only if they have identical implementations

up to the expansion of primitive combinators.

https://en.wikipedia.org/wiki/Combinatory_logic

Examples of Combinators (3-3)

There are <u>many ways</u> to <u>implement</u> an identity function;

(S K K) and **I** are among these ways.

(S K S) is yet another.

We will use the word <u>equivalent</u> to indicate <u>extensional</u> equality,

<u>reserving</u> <u>equal</u> for <u>identical</u> combinatorial terms.

https://en.wikipedia.org/wiki/Combinatory_logic

A more interesting combinator is

the fixed point combinator or **Y combinator**,

which can be used to implement recursion.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinator Definitions (2)

The combinators **I**, **K**, and **S** were introduced by Schönfinkel and Curry,

who showed that <u>any</u> λ-expression can <u>essentially</u> be <u>formed</u>

by <u>combining</u> them.

More recently combinators have been applied

to the design of implementations for functional languages.

In particular **Y** (also called the paradoxical combinator)

can be seen as <u>producing</u> fixed points, since **Y(f)** reduces to **f(Y(f))**.

I = λx . x

K = λx . λy . x

S = λx . λy . λz . x(z)(y(z))

Y = λf . (λu . f(u(u))) (λu . f(u(u)))

Combinatory Logic and Lambda Calculus (1)

Lambda calculus is concerned with objects called lambda-terms,

which can be represented by the following three forms of strings:

v

λv. E₁

(E₁ E₂)

where **v** is a variable name drawn

from a predefined infinite set of variable names,

and **E₁** and **E₂** are lambda-terms.

https://en.wikipedia.org/wiki/Combinatory_logic

Terms of the form **λv. E$_1$** are called abstractions.

The variable **v** is called the formal parameter of the abstraction,

and **E$_1$** is the body of the abstraction.

The term **λv. E$_1$** represents the function

      applied to an argument,

      binds the formal parameter **v** to the argument

      computes the resulting value of **E$_1$**

      returns **E$_1$**, with every occurrence of **v** replaced by the argument.

v

λv. E$_1$

(E$_1$ E$_2$)

https://en.wikipedia.org/wiki/Combinatory_logic

Terms of the form (E_1 E_2) are called **applications**.

applications <u>model</u> function invocation or execution:

the function represented by E_1 is to be <u>invoked</u>,

with E_2 as its argument, and the <u>result</u> is computed.

Combinatory Logic and Lambda Calculus (3-2)

If E_1 (the applicand) is an abstraction, the term may be reduced:

E_2, the argument, may be substituted into the body of E_1

in place of the formal parameter v of E_1,

and the result is a new lambda term which is equivalent to the old one.

If a lambda term contains no subterms of the form $((\lambda v.\ E_1)\ E_2)$

then it cannot be reduced, and is said to be in normal form.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Logic and Lambda Calculus (4)

The motivation for this <u>definition</u> of reduction is

that it <u>captures</u> the <u>essential behavior</u> of all mathematical functions.

For example, consider the function

that computes the square of a number. We might write

   The **square** of **x** is **x * x**   (using ***** to indicate multiplication.)

x here is the formal parameter of the function.

To <u>evaluate</u> the **square** for a particular argument, say 3,

we insert it into the definition in place of the formal parameter:

   The **square** of **3** is **3 * 3**

Combinatory Logic and Lambda Calculus (5)

To <u>evaluate</u> the resulting expression **3 * 3**, we would have to resort

to our knowledge of <u>multiplication</u> and the <u>number</u> **3**.

Since any <u>computation</u> is simply a <u>composition</u> of

     the <u>evaluation</u> of suitable functions

     <u>on</u> suitable primitive arguments,

this simple substitution principle suffices

to capture the <u>essential mechanism</u> of <u>computation</u>.

Combinatory Logic and Lambda Calculus (6)

Moreover, in lambda calculus, notions such as '**3**' and '*****'

can be represented <u>without</u> any need for <u>externally</u> <u>defined</u>

primitive operators or constants.

It is possible to identify terms in lambda calculus,

which, when suitably <u>interpreted</u>, behave like the <u>number</u> **3**

and like the <u>multiplication operator</u> *****, q.v. Church encoding.

Combinatory Logic and Lambda Calculus (7)

Lambda calculus is known to be computationally equivalent

in power to many other plausible models for computation

(including Turing machines);

that is, any calculation that can be accomplished

in any of these other models can be expressed in lambda calculus,

and vice versa.

According to the Church-Turing thesis,

both models can express any possible computation.

Combinatory Logic and Lambda Calculus (8-1)

lambda-calculus can represent any conceivable computation

using only the simple notions

of function abstraction and application

based on simple textual substitution of terms for variables.

abstraction is not even *required*.

Combinatory logic is

a model of computation equivalent to lambda calculus,

but without abstraction.

Combinatory Logic and Lambda Calculus (8-2)

Combinatory logic is

a model of computation equivalent to lambda calculus,

but without abstraction.

The advantage of this is that

evaluating expressions in lambda calculus is quite complicated

because the semantics of substitution must be specified

with great care to avoid variable capture problems.

evaluating expressions in combinatory logic is much simpler,

because there is no notion of substitution.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Calculus

abstraction is the only way to <u>manufacture</u> functions

in the lambda calculus

Instead of abstraction,

combinatory calculus provides a <u>limited</u> <u>set</u> of primitive functions

out of which other functions may be built.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Terms (1)

A combinatory term has one of the following forms:

Syntax	Name	Description
x	Variable	A character or string representing a combinatory term.
P	Primitive function	One of the combinator symbols **I**, **K**, **S**.
(M N)	Application	Applying a function to an argument. **M** and **N** are combinatory terms.

https://en.wikipedia.org/wiki/Combinatory_logic

Combinatory Terms (2)

The primitive functions are combinators, or functions that,

when seen as lambda terms, contain <u>no</u> free variables.

To shorten the notations, a general convention is that ($E_1\ E_2\ E_3\ \ldots\ E_n$),

or even $E_1\ E_2\ E_3\ \ldots\ E_n$, denotes the term ($\ldots$ (($E_1\ E_2$) E_3) $\ldots\ E_n$) .

This is the same general convention (left-associativity)

as for multiple application in lambda calculus.

https://en.wikipedia.org/wiki/Combinatory_logic

Reductions in Combinatory Logic

In combinatory logic, each primitive combinator comes

with a reduction rule of the form

$$(P \; x_1 \; ... \; x_n) = E$$

where **E** is a term mentioning only variables from the set {x_1 ... x_n}.

It is in this way that primitive combinators behave as functions.

Iota combinator (1)

It is possible to define a complete system

using only one (improper) combinator.

An example is Chris Barker's iota combinator,

which can be expressed in terms of **S** and **K** as follows:

ιx = xSK

Iota combinator (2)

It is possible to reconstruct **S**, **K**, and **I** from the iota combinator.

Applying ɩ to itself gives ɩɩ = ɩ**SK** = **SSKK** = **SK(KK)**

which is functionally equivalent to **I**.

ɩx   = x**SK**

ɩɩ   = ɩ**SK**

ɩ**S**   = **SSK**

ɩɩ = (ɩ**S**)**K** = (**SSK**)**K** = **SK(KK)**

ɩɩ y = ɩ**SKy** = **SK(KK)y** = **Ky (KK)y = y**   ➡️   ɩɩ = **I**

ɩx = x**SK**

ɩɩ = ɩ**SK**   *and*   ɩ**S** = **SSK**
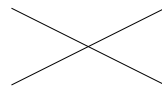
(ɩ**S**)**K** = (**SSK**)**K**

SSKK=**SK KK**

SKKK = **KK KK** = **K**

K K (**KK**)   = **K**

ɩɩ y = ɩ**SKy** = **SK(KK)y** = **Ky (KK)y = y**

S K x y = K y (x y) = y

I y = y

Iota combinator (3)

K can be constructed by applying ι twice to **I** (= ιι)

(which is equivalent to application of ι to itself):

ι(ι(ιι)) = ι(ιιSK) = ι(ISK) = ι(SK) = SKSK = **K.**

ι(ι(ιι))   = ι(ιιSK)

       = ι(ISK)   ⬅   ιι = I

       = ι(SK)

       = (SK)SK

       = K K SK

       = K

ιx   = xSK

ιι   = ιSK

ιS   = SSK

ιι = (ιS)K = (SSK)K = SK(KK)

ιι = ι(SK) = (SK)SK = K K SK = K

K can be constructed by applying ι twice to **I** (= ιι)

(which is equivalent to application of ι to itself):

ι(ι(ιι)) = ι(ιιSK) = ι(**I**SK) = ι(SK) = SKSK = **K.**

Applying ι one more time to **ι(ι(ιι))** gives

ι(ι(ι(ιι))) = ιK = KSK = S

ιx   = xSK

ιι   = ιSK

ιS   = SSK

ιι = (ιS)K = (SSK)K = SK(KK)

ι(ι(ιι))   = ι(ιιSK)

              = ι(ISK)

              = ι(SK)

              = (SK)SK

ιx   = xSK

ιι   = I

ιιι   = K

ιιιι = S

Improper Combinator

Improper combinators, meaning that they are expressed
in terms of other combinators rather than pure abstractions.

To be precise: in lambda calculus a proper combinator is
an expression of the form $(\lambda.x_1x_2\ldots P(x_1,x_2,\ldots))$,
where $P(x_1,x_2,\ldots)$ only has x_1, x_2 etc. as free variables,
and does not contain any abstractions.

So for example, $(\lambda xyz.x(zz))$ is a proper combinator,
but $(\lambda x.x(\lambda y.y))$ is not, because it contains x applied to a lambda term.

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf