SKI Calculus

Minimal Formal Computation in AlexForth on 6502

Alexandre Dumont
@adumont

#FORTH2020
Feb 10th, 2024

312 M. Schönfinkel.

Da y willkürlich ist, können wir dafür ein beliebiges Ding oder eine beliebige Funktion einsetzen, also z. B. Cx. Dies gibt:

$$Ix = (Cx)(Cx).$$

Nach der Erklärung von S bedeutet dies aber:

$$SCCx,$$

so daß wir erhalten:

$$I = SCC.$$ [3])

Übrigens kommt es in dem Ausdruck SCC auf das letzte Zeichen C gar nicht einmal an. Setzen wir nämlich oben für y nicht Cx, sondern die willkürliche Funktion φx, so ergibt sich entsprechend:

$$I = SC\varphi,$$

wo also für φ jede beliebige Funktion eingesetzt werden kann[4]).

2. Nach der Erklärung von Z ist

CURRY: *An Analysis of Logical Substitution.* 371

III. POSTULATES.

None.

IV. RULES.

0. If x and y are entities, then (xy) shall be an entity.

1. $(=)$ shall have the properties of identity. These properties may be specified by a few simple rules; but in this treatment we shall not go into that detail. We shall treat $(=)$ as if it were precisely the intuitive relation of equality.

2. If x and y are any entities, then

$$Kxy = x$$

3. If x, y, z are entities, then

$$Sxyz = xz(yz)$$

4. If X and Y are combinations of S and K, and if there exists an integer n such that by application of the preceding rules we can formally reduce the expressions $Xx_1\,x_2 \cdots x_n$ and $Yx_1\,x_2 \cdots x_n$ to combinations of $x_1\,x_2 \cdots x_n$ which have the same structure, then $X = Y$.

If the above primitive frame were a part of a general theory of logic, the term entity would include not only the various combinations of S and K, but all the notions of logic as well. In the sequel we shall accordingly speak of the application of combinations S and K to various logical notions, and of the resulting notions to each other, just as if these notions had been adjoined to the above frame.

The *raison d'être* of the theory based on this frame is the following fact:

Content

- Introduction
- SKI Combinator calculus
- Challenge in Forth
- SKI Implementation in AlexForth 6502
- Booleans
- References

Da y willkürlich ist, können wir dafür ein beliebiges Ding oder eine beliebige Funktion einsetzen, also z. B. Cx. Dies gibt:

$$Ix = (Cx)(Cx).$$

Nach der Erklärung von S bedeutet dies aber:

$$SCCx,$$

so daß wir erhalten:

$$I = SCC.$$ [3])

Übrigens kommt es in dem Ausdruck SCC auf das letzte Zeichen C gar nicht einmal an. Setzen wir nämlich oben für y nicht Cx, sondern die willkürliche Funktion φx, so ergibt sich entsprechend:

$$I = SC\varphi,$$

wo also für φ jede beliebige Funktion eingesetzt werden kann[4]).

2. Nach der Erklärung von Z ist

III. POSTULATES.

None.

IV. RULES.

0. If x and y are entities, then (xy) shall be an entity.

1. $(=)$ shall have the properties of identity. These properties may be specified by a few simple rules; but in this treatment we shall not go into that detail. We shall treat $(=)$ as if it were precisely the intuitive relation of equality.

2. If x and y are any entities, then

$$Kxy = x$$

3. If x, y, z are entities, then

$$Sxyz = xz(yz)$$

4. If X and Y are combinations of S and K, and if there exists an integer n such that by application of the preceding rules we can formally reduce the expressions $Xx_1 x_2 \cdots x_n$ and $Yx_1 x_2 \cdots x_n$ to combinations of $x_1 x_2 \cdots x_n$ which have the same structure, then $X = Y$.

If the above primitive frame were a part of a general theory of logic, the term entity would include not only the various combinations of S and K, but all the notions of logic as well. In the sequel we shall accordingly speak of the application of combinations S and K to various logical notions, and of the resulting notions to each other, just as if these notions had been adjoined to the above frame.

The *raison d'être* of the theory based on this frame is the following fact:

How it started?

Alexandre Dumont @adumont · 21 jul. 2021

Now that I have implemented FORTH, I'm looking for a new challenge. Any recommendation?

Tony "Abolish ICE" Arcieri 🦀🌹 @bascule · 22 jul. 2021

SKI calculus

2 2

Alexandre Dumont @adumont · 22 jul. 2021

What is this? I have never heard of it!

1 1

Tony "Abolish ICE" Arcieri 🦀🌹 @bascule

It's one of the most minimal forms of the untyped lambda calculus, composed of three functions, but actually two because one can be composed from the other two

SKI combinator calculus

What is SKI combinator calculus ?

- It was introduced by **Moses Schönfinkel** (in 1920) and further developed later by **Haskell Curry** (in 1927)

- It is a **combinatory logic system** and a **formal computational system**
- The first, and a **minimal** formalism for **universal computation**
- Relevant in the mathematical theory of algorithms because it is an **extremely simple Turing-complete language**
- It can be seen as a *reduced version* of **lambda calculus** (Alonso Church, 1936)

Combinators

- Combinators are **higher-order functions** with **no free variables**, that
  - **take one function as an argument** and
  - **return a function**

- In mathematics and computer science, a higher-order function (HOF) is a function that does at least one of the following:
  - takes one or more functions as arguments
  - returns a function as its result

Application

- Combinators are **...functions**. They take a **function** as argument, and they return a **function**!

So far everything is a function

- Applying a combinator ***F*** to an argument x is called **Application** and is writen *F*x
- Application is **left associative**: $Fxy = (Fx)\,y$ ($\neq$ $F(x(y))$ or $F(xy)$)

S, ***K*** and ***I*** combinators **definitions**

Identity Identitätsfunktion	***Constant*** Konstanzfunktion	***Substitution*** Verschmelzungfunktion
Ix = x	***K***xy = x	***S***fgx = fx(gx)

I can also be defined in terms of ***S*** and ***K***: ***I=SKK***
So ***S*** and ***K*** are the only building blocks needed to have a **turing complete language**!

Challenge in FORTH

First approach: using *normal* Forth words

The *Application* of a combinator consumes 1 argument.

Seems easy: we have **words**, and a **stack**… Let's try postfix notation:

*I*x = x :	x ***I*** $\rightarrow$ x	ok… ***I*** looks like **NOP**
Kxy=x :	y x ***K*** $\rightarrow$ x	***K*** looks like **NIP**…

…Except it's not! 🤦

What is wrong?

I x = x
K xy = x

Let's have a look at ***K I*** : ***KI***xy = (***KI***)xy = (***KI***x)y = ***I***y = y

Now with *normal* Forth words:

y x ***I*** → y x
y x ***K*** → **x** which would be wrong!

Several issues arise if we define combinators as *normal Forth words*:

- Combinators take only 1 argument! ***K*** can only see x (it can't drop the y)
- Left-associativity when we're in Forth (post-fixed): we need a way to **delay execution**!

How to address *delayed application*?

(1) We introduce a Forth word "**)**" that means "**apply**": : **)** (xt --) **EXEC** ;

Example: *K*xy=x y x ***K*** **)** → y ***K*X**) → x

(2) We need ***K*** to be a word that leaves an XT on the stack: K_{XT}

We'll then **apply K** (it executes the XT) with **)** :

$$y\ x\ K_{XT}\ ) \rightarrow y\ KX_{XT}\ ) \rightarrow x$$

Application behaviour of ***K***:

In this case the *application behaviour* of ***K*** is to leave on the stack the XT of a new word, whose application behavior will drop **y** and push **x** back to the stack

Implementation in AlexForth 6502

ENTER, and :FUNC

```
\ ENTER, starts a new word in dictionary without header
\ $4C is 6502's JMP
: ENTER, ( -- )  $4C C,   COMPILE COLON ;

\ Create a Function (or combinator)
\ Combinators are Higher Order Functions, meaning
\ they take a function as an argument and return a function
\ which we will eventually apply later using ")"
: :FUNC ( "name" -- ) CREATE ENTER, ] ;
```

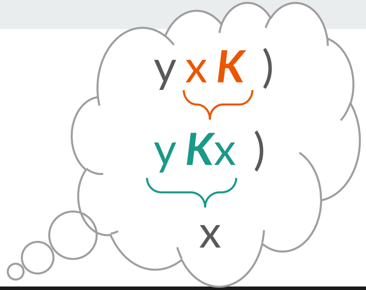
Application operator)

```
\ Application operator
: )   ( xt -- xt ) EXEC  ; \ Apply <=> "Application"

\ Syntactic sugar definitions
: ))  ( xt -- xt ) ) )   ;
: ))) ( xt -- xt ) ) ) ) ;
```

I Combinator

```
: ENTER, ( -- )   4C C,   COMPILE COLON ;
: :FUNC ( "name" -- ) CREATE ENTER, ] ;

\ Identity Combinator
\ Ix=x    λx.x
:FUNC I ( do nothing ) ;
```

When *applied*, the word ***I*** will do nothing



K Combinator

```
:FUNC K  \ Constant Combinator   Kxy=x   λxy.x
  HERE   \ leaves the XT of the Kx word on the stack
  ENTER, \ now we compile the Kx word
  COMPILE DROP      \ Drop Y
  COMPILE LIT       \ Push x onto the stack
  SWAP   \ put X back on TOS
  ,      \ store x into the definition of Kx
  COMPILE EXIT
;
```

Application Behaviour of K

The anonymous **Kx** word is a **closure**, *enclosing* the value of **x** present on the stack when *applying* ***K*** *to* ***x***

S Combinator

Sxyz = xz(yz)

The ***S*** word is a "function".

- when *applied*, S will create a new word SX enclosing **x**
- when *applied*, *SX* will create a new word SXY enclosing **y** (and **x**),
- when *applied*, SXY will execute the expected S behaviour on **x**, **y** and **z**

```
:FUNC S \  Sxyz = xz(yz)   λxyz.xz(yz)
  HERE
  ENTER,
  COMPILE HERE
  COMPILE ENTER,
  COMPILE SWAP
  COMPILE COMPILE COMPILE DUP
  COMPILE COMPILE COMPILE LIT \ y
  COMPILE , \ y
  COMPILE COMPILE COMPILE )
  COMPILE COMPILE COMPILE SWAP
  COMPILE COMPILE COMPILE LIT
  COMPILE LIT \ x
  SWAP
  , \ store x
  COMPILE ,
  COMPILE COMPILE COMPILE ))
  COMPILE COMPILE COMPILE EXIT
  COMPILE EXIT
;
```

Defining new combinators

As a example, here we define the ***KI*** combinator, in terms of ***K*** and ***I***:

```
\ Kite Combinator
\ KIxy=y    λxy.y

I K )   CONSTANT  KI
```

Booleans in **SKI** calculus

Boolean helper *functions*: **.T .F** and **BOOL**

```
\ We define those two functions so we
\ can check results of boolean operations
:FUNC .T .( TRUE )  ;          \ ."  " is .(  ) in AlexFORTH
:FUNC .F .( FALSE ) ;

ok  .T ) → TRUE
ok  .F ) → FALSE

: BOOL .F .T ;
```

Booleans: **True** & **False**

```
\ BOOLEANS
K  CONSTANT T     \ TRUE  λxy.x
KI CONSTANT F     \ FALSE λxy.y



ok   .F .T    T    ))) → TRUE
ok   BOOL     F    ))) → FALSE
```

NOT combinator: λfxy.fyx

```
\ NOT = λfxy.fyx
\ S(S(KS)(S(KK)(S(KS)I)))(KK)
K K ) I S K ) S )) K K ) S )) S K ) S )) S ))
CONSTANT NOT

ok  BOOL  T NOT )  )))   → FALSE
ok  BOOL  F NOT )  )))   → TRUE
```

AND combinator: λpq.pqp

```
\ AND = λpq.pqp
\ S(S(KS)I)K
K I S K ) S )) S ))
CONSTANT AND

ok  BOOL  T T AND ))  )))  → TRUE
ok  BOOL  F T AND ))  )))  → FALSE
ok  BOOL  T F AND ))  )))  → FALSE
ok  BOOL  F F AND ))  )))  → FALSE
```

OR combinator: λpq.ppq

```
\ OR = λpq.pKq
\ S(S(KS)(S(KK)(SII)))(KI)
I K ) I I S )) K K ) S )) S K ) S )) S ))
CONSTANT OR

ok  BOOL  T T OR  ))  )))  → TRUE
ok  BOOL  F T OR  ))  )))  → TRUE
ok  BOOL  T F OR  ))  )))  → TRUE
ok  BOOL  F F OR  ))  )))  → FALSE
```

NAND combinator: λpq.p(q(KI)(K))K

```
\ NAND = λpq.p(q(KI)(K))K
\ S(S(KS)(S(S(KS)K)(K(S(SI(K(KI)))(KK)))))(K(KK))
K K ) K ) K K ) I K ) K ) I S )) S )) K ) K S K ) S )) S )) S K ) S )) S ))
CONSTANT NAND

ok  BOOL  T T NAND ))  )))  → FALSE
ok  BOOL  F T NAND ))  )))  → TRUE
ok  BOOL  T F NAND ))  )))  → TRUE
ok  BOOL  F F NAND ))  )))  → TRUE
```

XOR or **Equality** combinator: λpq.p(q(T)(F))(q(F)(T))

```
\ XOR = λpq.p(q(K)(KI))(q(KI)(K))
\ S(S(KS)(S(S(KS)K)(K(S(SI(KK))(K(KI))))))(K(S(SI(K(KI)))(KK)))
K K ) I K ) K ) I S )) S )) K ) I K ) K ) K K ) I S )) S )) K ) K S K )
S )) S )) S K ) S )) S ))
CONSTANT XOR


ok   BOOL   T T XOR ))   )))   ⟶ TRUE
ok   BOOL   F T XOR ))   )))   ⟶ FALSE
ok   BOOL   T F XOR ))   )))   ⟶ FALSE
ok   BOOL   F F XOR ))   )))   ⟶ TRUE
```

Demo

References

- Implementation approach inspired in “S/K/ID: Combinators in Forth.” by Johan G. F. Belinfante, in Journal of FORTH Application and Research archive 4 (1987)
    - Download: https://vfxforth.com/flag/jfar/vol4/no4/article6.pdf
- Lambda Calculus:
    - Fundamentals of Lambda Calculus & Functional Programming in JavaScript, by Gabriel Lebec
    - A Flock of Functions: Combinators, Lambda Calculus, & Church Encodings in JS - Part II, by Gabriel Lebec
- Combinators:
    - Standford, CS242: Programming Languages(https://web.stanford.edu/class/cs242/materials.html) Combinator Calculus
    - https://en.wikipedia.org/wiki/SKI_combinator_calculus
    - Combinators: A Centennial View—Stephen Wolfram Writings
    - Where Did Combinators Come From? Hunting the Story of Moses Schönfinkel, by Stephen Wolfram

Links

AlexForth repository: https://github.com/adumont/hb6502/tree/main/forth

Emu6502 repository: https://github.com/adumont/emu6502

My web page with links to all my projects (and these slides): https://adumont.github.io/

Interact with me on Twitter: @adumont https://twitter.com/adumont

Forth2020 meetings archive, recordings and how to join us:
https://github.com/forth2020/zoom-presentations

Alexandre Dumont
@adumont

Thank you!