Implementing functional operators using SKI combinator calculus

The Greek Empress

2020

1 Notation

This document will explain the inner workings of various operations using APL as a tool of thought where feasible. Assumption is being made that numbers are being represented using church encoding in form of $\lambda f.\lambda z.f^n(z)$, where $f^0(x) = x$ and $f^n(x) = f(f^{n-1}(x))$ for example: $x = 0$, $f(x) = 1$, $f(f(x)) = 2$.

Application is assumed to follow the terms of β-reduction - for example, `2 {α×*2ω} 2` would become `(λαω . α×*2ω) 2 2`. Sometimes the SKI calculus expression will be written freehand. In this case, the rule on adding parenthesis to build a binary tree is simple: $\alpha\beta\gamma$ becomes $((\alpha\beta)\gamma)$ (bind to the left), $\alpha\beta\gamma\delta$ becomes $(((\alpha\beta)\gamma)\delta)$, and so on - assuming greek letters represent distinct terms.

2 Constants

Basic boolean constants are relatively straightforward to implement: `true`: `((S(KK))I)`, `false` and `0`: `(KI)`. To demonstrate the use of Church numerals, let's look at this example for `2, 3` and `4`:

```
1: ((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI))
2: ((S((S(K((S(KS))(S(KI)))))((S(KK))I)))1)
3: ((S((S(K((S(KS))(S(KI)))))((S(KK))I)))2)
4: ((S((S(K((S(KS))(S(KI)))))((S(KK))I)))3)
```

The application of the same `successor` formula yields next church numerals.

3 Operators

Most of operators presented are (certainly) overengineered and their operation can be represented using smaller bits of code. The successor formula (`succ←1+⊢`) follows:

```
((S(K(S((S(K((S(KS))(S(KI)))))((S(KK))I)))))((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(
S(KI)))))((S(K(S(K((S(KS))(S(KI)))))))((S((S(KI))((S(K((S(KS))(S(K((S(KS))(S(KI))))))))(
(S(K(S(KK))))((S(KK))I)))))(K((S(KK))I))))))(KI)))
```

Let’s try applying `succ` to `1`:

```
(succ 1)
(((S(K(S((S(K((S(KS))(S(KI)))))((S(KK))I)))))((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))
(S(KI)))))((S(K(S(K((S(KS))(S(KI)))))))((S((S(KI))((S(K((S(KS))(S(K((S(KS))(S(KI))))))))
((S(K(S(KK))))((S(KK))I)))))(K((S(KK))I))))))(KI)))((S((S(KI))((S(K((S(KS))(S(KI)))))((S
(KK))I))))(KI)))((S((S(K((S(KS))(S(KI)))))((S(KK))I)))((S((S(KI))((S(K((S(KS))(S(KI)))))
((S((S(K((S(KS))(S(KI)))))((S(KK))(K((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI))
))))((S(KK))I)))))(KI)))
```

Although the output tree is different than the expected one for `2`, both expressions evaluate to the same result:

```
((S((S(K((S(KS))(S(KI)))))((S(KK))I)))((S((S(KI))((S(K((S(KS))(S(KI)))))((S((S(K((S(KS))
(S(KI)))))((S(KK))(K((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI))))))((S(KK))I)))
))(KI)))
(x0->(x1->x0(x0(x1))))

((S((S(K((S(KS))(S(KI)))))((S(KK))I)))((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI
)))
(x0->(x1->x0(x0(x1))))
```

The predecessor formula is vastly different (`pred←⊢-1`):

```
((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(KI)
)))))))((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(KI)))))((S(K(S(K((S(KS))(S(KI))))))
)((S((S(KI))((S(K((S(KS))(S(K((S(KS))(S(KI))))))))((S(K(S(KK))))((S(KK))I)))))(K((S(K(S(
K(S(K(S((S(KI))I))))))))((S(K(S(K(S((S(K((S(KS))(S(KI)))))((S(KK))I)))))))((S(K(S(K(S(KK
))))))((S(K(S(KK))))((S(KK))I))))))))))(K((S(KK))I)))))(KI))
```

Let’s try applying `prec` to `1`:

```
(prec 1)
```

```
(((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(KI
))))))))((S((S(K((S(K((S((S(K((S(KS))K)))S))(KK))))(S(KI)))))((S(K(S(K((S(KS))(S(KI)))))
))((S((S(KI))((S(K((S(KS))(S(K((S(KS))(S(KI))))))))((S(K(S(KK))))((S(KK))I)))))(K((S(K(S
(K(S(K(S((S(KI))I))))))))((S(K(S(K(S((S(K((S(KS))(S(KI)))))((S(KK))I)))))))((S(K(S(K(S(K
K))))))((S(K(S(KK))))((S(KK))I))))))))))(K((S(KK))I)))))(KI))((S((S(KI))((S(K((S(KS))(S(
KI)))))((S(KK))I))))(KI)))
(x0->(x1->x1))
```

As expected, the result is zero ($\lambda x.x$). Let's try implementing arihmetic - starting with multiplication and exponentation (`mul`←×, `exp`←∗).

```
mul=((S((S(KI))((S(K((S(KS))(S(K((S(KS))(S(KI))))))))((S(K(S(KK))))((S(KK))I)))))(K((S((
S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI))))

exp=((S(K(S((S(KI))I))))((S(KK))I))
```

Applying `exp` to `3` and `3`:

```
(exp 3 3)

(((S(K(S((S(KI))I))))((S(KK))I))((S((S(K((S(KS))(S(KI)))))((S(KK))I)))((S((S(K((S(KS))(S
(KI)))))((S(KK))I)))((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(KI))))((S((S(K((S(KS
))(S(KI)))))((S(KK))I)))((S((S(K((S(KS))(S(KI)))))((S(KK))I)))((S((S(KI))((S(K((S(KS))(S
(KI)))))((S(KK))I))))(KI)))))

(x0->(x1->x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(x0(
x0(x1)))))))))))))))))))))))))))))
```

The result is `27` encoded as a Church numeral. The code could be shortened using the duplication snippet:

```
dup=((SI)I)

((SI)I)α =>
(Iα)(Iα) =>
αα
```

Another two operations follow - fixed point combinator and a zero check:

```
is_zero=((S((S(KI))((S((S(KI))I))(K(K(KI))))))(K((S(KK))I)))
Y=((S(K((S((S(KI))I))I)))((S((S(KI))((S(K((S(KS))(S(KI)))))((S(KK))I))))(K((S((S(KI))I))
I))))
```

Everything presented is sufficient to write a factorial program using SKI calculus. This problem is left as an excercise for the reader. The fixed point combinator can be devised more efficiently (look: John Tromp's fixed point combinator).

4 Finishing words

The factorial function can be defined as follows: $\lambda f n.(\mathrm{i}_0\, n)1(\mathrm{mul}\, n(f(\mathrm{pred}\, n)))$. While implementing SKI calculus simplification tool, there are many things to consider. Taken for example term `(((SI)I)((SI)I))`, it's impossible to reduce it further. The term `((KI)(((SI)I)((SI)I)))` (derived from the irreducible term) will diverge, because:

```
((KI)(((SI)I)((SI)I)))
one K step: I
one S step: ((KI)(((SI)I)((SI)I)))
```

This means, if the `S` combinator gets evaluated first, it's impossible to reduce the sequence further. On the other hand, if the `K` combinator gets evaluated first, the entire expression evaluates to `I`. These precautions need to be taken while implementing a SKI calculus simplification tool / SKI calculus-based calculator.