Combinators

An alternative approach to computation

Overview

- Why do we care?
- Functional languages
- Combinators
- More combinators
- It’s a bird--it’s a plane--it’s a SUPERCOMBINATOR!
- More on combinator evaluation
- Combinator architectures
- Other topics
- References

Why do we care?

1. Something about combinators WILL be on the final
2. There is a super awesome extra credit project on combinators

Why do we care?

1. We want the right tool for the job
2. Want to avoid ‘compute by side effect’
3. The big picture

Goals

1. Explain how a functional language differs from other programming paradigms
2. Explain what combinators are and how they work
3. Reduce a combinator expression (using normal order evaluation)

Functional Languages!

- Functions as first class objects
- “Single Assignment:” Trying to remove computing by side effects
  - No multiple assignments of values to variables
- You may have heard of or seen some already

Functional Languages!

Functional Languages: code examples

- Scheme:

```
(define fact
   (lambda (n)
     (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

Functional Languages: code examples

- Haskell:

```
factorial n = if n < 2 then 1 else n * factorial (n-1)
```

Functional arithmetic example

```
(+ (* 3 2) (/ 10 5))
```

Functional arithmetic example

(+ (* 3 2) (/ 10 5))

Functional arithmetic example

```
(+    (* 3 2)    (/ 10 5))
    (*    3    2)    (/ 10 5)
```

Functional arithmetic example

```
(+     (* 3 2)     (/ 10 5))
   (*    3    2)    (/ 10 5)
         6    (/    10    5)
```

Functional arithmetic example

```
(+    (* 3 2)    (/ 10 5))
    (*    3    2)    (/ 10 5)
          6    (/    10    5)
          6      2
```

Functional arithmetic example

(+ (* 3 2) (/ 10 5))

(* 3 2) (/ 10 5)

6 (/ 10 5)

6 2

8

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

((if True + -) (* 2 3) (- 3 7)))

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

((if True + -) (* 2 3) (- 3 7)))

(+ (* 2 3) (- 3 7))

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

((if True + -) (* 2 3) (- 3 7)))

(+ (* 2 3) (- 3 7))

(+ 6 (- 3 7))

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

((if True + -) (* 2 3) (- 3 7)))

(+ (* 2 3) (- 3 7))

(+ 6 (- 3 7))

(+ 6 -4)

Combinators: functions as 1st class objects

- This puts the ‘fun’ in functional programming

((if (= 5 5) + -) (* 2 3) (- 3 7)))

((if True + -) (* 2 3) (- 3 7)))

(+ (* 2 3) (- 3 7))

(+ 6 (- 3 7))

(+ 6 -4)

2

<pause for questions>

Combinators: high level

- Historical beginnings
- Simple functions (and only functions)
- No variables
- Shuffle things around

Combinators: this is it

- Look Ma - no data - only functions
- And only 3 functions needed for ANYTHING! (S, K, and I)
- Grammar of expressions is trivial! (caf = “constant application function”)

    <caf> -> <constant> | (<caf> <caf>*)

    <constant> -> S | K | I

Combinators: All we need is to “reorder”

Let a, b, c be arbitrary <caf>s:

S a b c -> ac(bc)

K a b -> a

I a -> a

Combinators: arithmetic revisited

- Numbers and operators are a lie!
  - The number k is the function +1 applied to 0 k times
- We *only* need S, K, and I
- but....

Combinators: built-ins

- 0 = K I

- Successor function:

Thus total solution is:

S(S(KS){S(S(KS)(S(KK)(KS)))(S{S(KS)(S(KK)(KK))}(KI))})(S{S(KS)S{S(KS)(S(KK)(KS))}
{S(S(KS)(S{S(KS)(S(KK)(KS))}{S(S(KS)(S(KK)(KK)))(S(KK)I)}))(S{S(KS)(S(KK)(KK))}(KI))}}
(S(KK)(KI)))

Combinators: arithmetic revisited

Even the I is unnecessary

S <u>K</u> <u>K</u> <u>a</u>

K <u>a</u> <u>(Ka)</u>

a

<pause for questions>

More combinators

Example 1:

Say we want a composition function B such that B x y z -> x(yz)

I strongly suspect B = S (KS) K

But we should probably check

More combinators

Example 1 reduction:

S (KS) K x y z

More combinators

Example 1 reduction:

S (KS) K x y z

K S x (Kx) y z

More combinators

Example 1 reduction:

S (KS) K x y z

K S x (Kx) y z

S (Kx) y z

More combinators

Example 1 reduction:

S (KS) K x y z

K S x (Kx) y z

S (Kx) y z

K x z (y z)

More combinators

Example 1 reduction:

S (KS) K x y z

K S x (Kx) y z

S (Kx) y z

K x z (y z)

x (y z)

More combinators

Example 2:

Maybe we would like to double the second argument as in W x y = x y y

Let’s check if W = S S (S K) works

More combinators

Example 2: reduction:

S S (SK) x y

More combinators

Example 2: reduction:

S S (SK) x y

S x ((SK) x) y

More combinators

Example 2: reduction:

S S (SK) x y

S x ((SK) x) y

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

More combinators

Example 2: reduction:

S S (SK) x y

S x ((SK) x) y

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

x y K y (x y)

More combinators

Example 2: reduction:

S S (SK) x y

S x ((SK) x) y

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

x y K y (x y)

x y y

More combinators

Example 3:

How about swapping the 2nd and 3rd arguments with C x y z -> x z y

C = S (S (K (S (K S) K)) S) (K K)

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

(K (Sx))y (((KK)x)y) z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

(K (Sx))y (((KK)x)y) z

Sx (((KK)x)y) z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

(K (Sx))y (((KK)x)y) z

Sx (((KK)x)y) z

xz KK x y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

(K (Sx))y (((KK)x)y) z

Sx (((KK)x)y) z

xz KK x y z

xz K y z

More combinators

Example 3: reduction:

S (S (K (S (K S) K)) S) (K K) x y z

S (K (S (K S) K)) S x ((KK)x) y z

K (S (K S) K) x (Sx) ((KK)x) y z

S (K S) K (Sx) ((KK)x) y z

(K S) (Sx) (K(Sx)) ((KK)x) y z

K S (Sx) (K(Sx)) ((KK)x) y z

S (K(Sx)) ((KK)x) y z

(K (Sx))y (((KK)x)y) z

Sx (((KK)x)y) z

xz KK x y z

xz K y z

xzy

POP QUIZ TIME

Pop Quiz: solutions

Solution 1 (note: this differs slightly from in class):

(S S I I K) x y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

I K ((II)K) x y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

I K ((II)K) x y z

K ((II) K) x y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

I K ((II)K) x y z

K ((II) K) x y z

II K y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

I K ((II)K) x y z

K ((II) K) x y z

II K y z

K y z

Pop Quiz: solutions

Solution 1:

(S S I I K) x y z

S I (II) K x y z

I K ((II)K) x y z

K ((II) K) x y z

II K y z

K y z

y

Pop Quiz: solutions

Solution 2 (note: this differs slightly from in class):

(S (SI) I K) w x y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

K(K (IK)) w x y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

K(K (IK)) w x y z

K (IK) x

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

K(K (IK)) w x y z

K (IK) x

I K y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

K(K (IK)) w x y z

K (IK) x

I K y z

K y z

Pop Quiz: solutions

Solution 2:

(S (SI) I K) w x y z

S I K (IK) w x y z

I (IK) (K(IK)) w x y z

I K(K(IK)) w x y z

K(K (IK)) w x y z

K (IK) x y z

I K y z

K y z

y

Supercombinators

In addition to the ones you showed

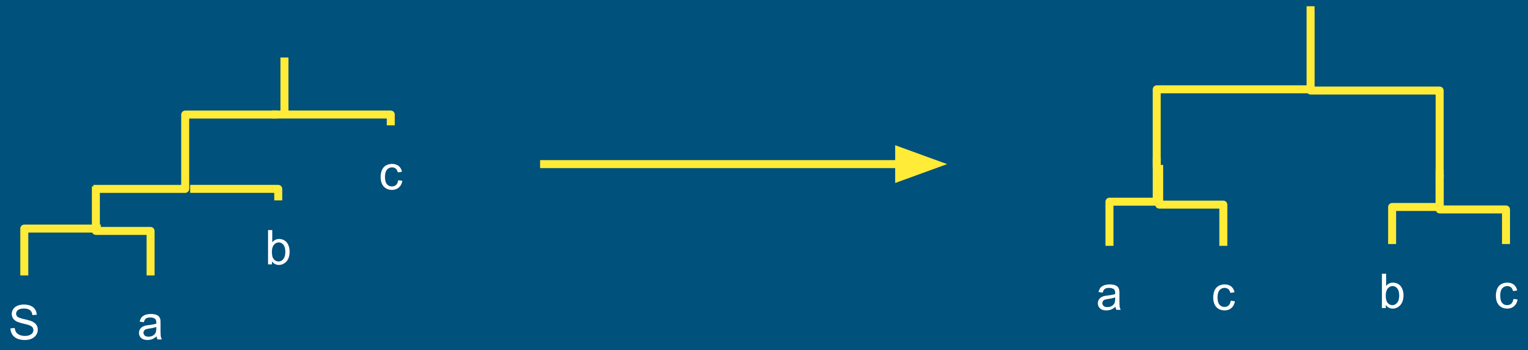
(B, C, W)

- P: P x y z -> z y x
- T: T x y -> y x
- J: J x -> I
- Y: Y x -> x (Y x)
- And infinitely more!

Think about how P, T, and J could be replaced by S, K, and I

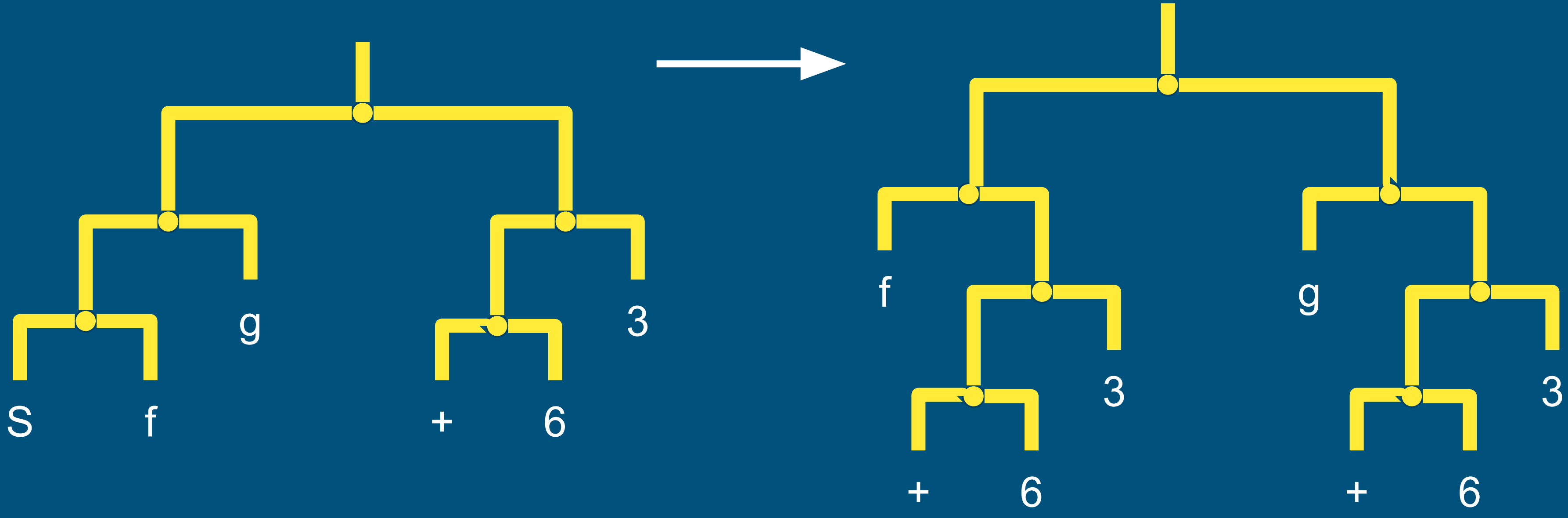
Combinators and computation

Graph reduction: (S a b c)



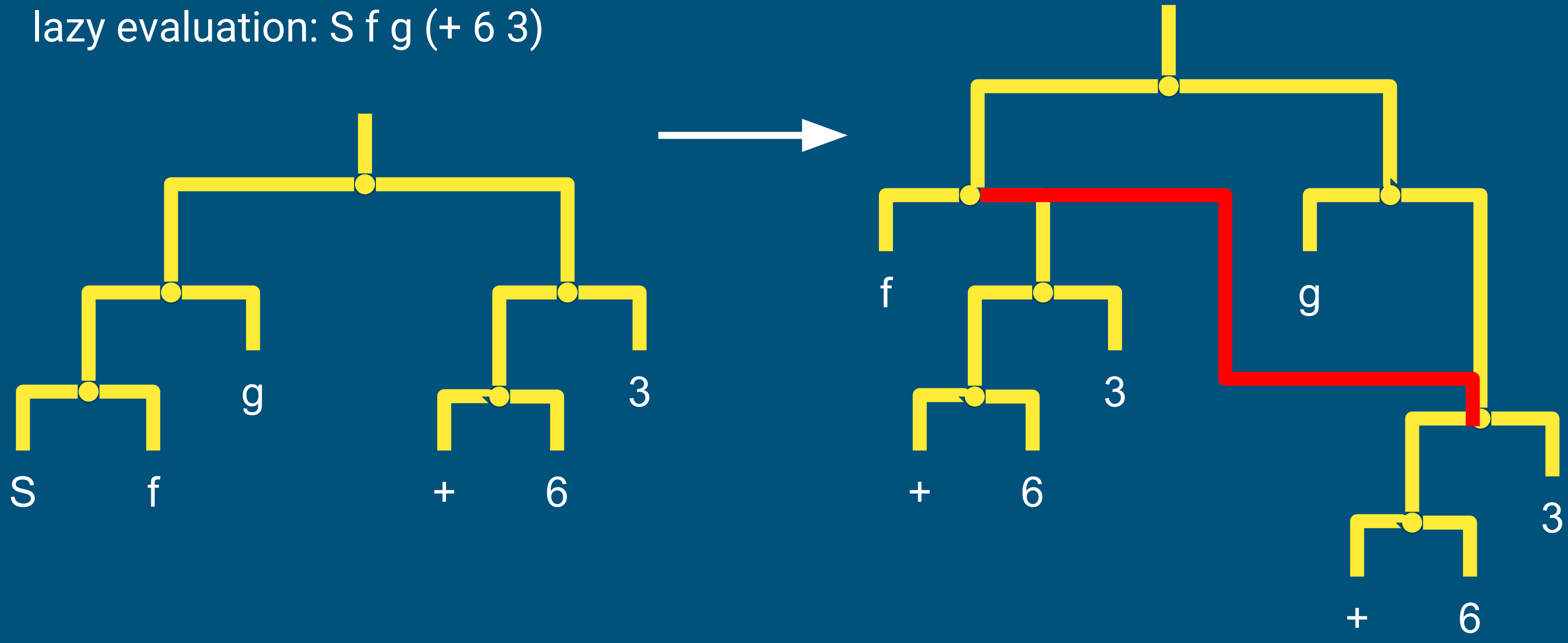
Combinators and computation

lazy evaluation: S f g (+ 6 3)



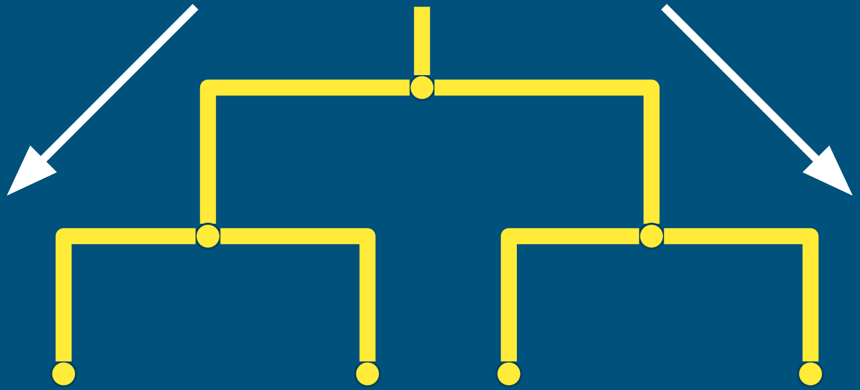
Combinators and computation

lazy evaluation: S f g (+ 6 3)



Combinators and computation

Parallel execution



Combinator architectures: SKIM machine

Background

- Programs run through interpreter which stood in as microcode
- Used memory more efficiently
- Any inefficiencies due to hardware bias

Combinator architectures: others

- SKIM (1980)
- Alice (1981)
- Curry (1986)
- Grip (1987)
- <v, G>-Machine (1989)
- PCM-1 (1989)
- ABC (1991)

Other topics

- Lambda functions
- Currying
- Futures
- Continuation

Conclusions

- You grew up with functions as second class citizens
  - Arguments are “data objects”, as are results
- But in reality we need ONLY 1st class functions
  - That take ONLY other functions as arguments
  - And yield ONLY other functions
- And only 3 functions are needed for expressing ANYTHING!
- Everything else is *computational sugar*

Conclusions

- Pure combinatory logic and pure functional paradigms are problematic in the real world
  - To do something useful we *need* side effects / states
- But operating in a functional way has extraordinary benefits

<pause for questions>

References/Further Reading

1. Cardone F, Hindley JR. History of lambda-calculus and combinatory logic. Handbook of the History of Logic. 2006;5:723-817.

2. Hartel PH, Muller HL. Functional C. Addison Wesley Longman; 1997.

3. Augustsson L, Johnsson T. Parallel graph reduction with the (v, G)-machine. InProceedings of the fourth international conference on Functional programming languages and computer architecture 1989 Nov 1 (pp. 202-213). ACM.

4. Hughes RJ. Super-combinators a new implementation method for applicative languages. InProceedings of the 1982 ACM symposium on LISP and functional programming 1982 Aug 15 (pp. 1-10). ACM.

5. Clarke TJ, Gladstone PJ, MacLean CD, Norman AC. Skim-the s, k, i reduction machine. InProceedings of the 1980 ACM conference on LISP and functional programming 1980 Aug 25 (pp. 128-135). ACM.

6. Kogge PM. The architecture of symbolic computers. McGraw-Hill, Inc.; 1990 Sep 1.

7. Suchocki R, Kalvala S. Microscheme: Functional programming for the Arduino. In2014 SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP 2015 Sep 11.

References/Further Reading

1. https://en.wikipedia.org/wiki/Combinatory_logic
2. http://wiki.c2.com/?WhenToUseWhatParadigm
3. http://lambda.jimpryor.net/topics/week3_combinatory_logic/
4. http://www.shido.info/lisp/idx_scm_e.html
5. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiT4YLnoMLXAhVCKCYKHc3YDiQQFggoMAA&url=http%3A%2F%2Fwww.macs.hw.ac.uk%2F~greg%2Flimits%2520to%2520computability%2FSKI%2520combinators%2520(really)%2520are%2520%2520Turing%2520complete.pptx&usg=AOvVaw21S1IDtt9QsRjJPVgAe_KM