Implementation of Functional Langauges

Implementation techniques for functional languages:

- Landin's SECD machine (1963) & successors
- Combinators (1979)
- Spineless Tagless G-machines (1991ish)
- Other developments: lambda lifting, supercombinators, special-purpose hardware for parallel graph reduction, etc.

The SECD Machine - a stack based machine

Consists of:

- S = stack
- E = environment: the current binding environment.
- C = code vector: the code to be evaluated.
- D = dump: other older contexts (which are restored after we're done evaluating a function).

The SECD machine uses applicative order evaluation.

To get normal order evaluation, pass an anonymous function (a thunk) rather than a value as an argument. Evaluate the function whenever the parameter value is needed.

This is the same as call-by-name in Algol-60. Improve efficiency by replacing the anonymous function call by its value after it is invoked -- this gives lazy evaluation.

Combinators

Turner 1979: An alternative implementation strategy, using combinator graphs.

A combinator is basically a function with no free variables or constants (See for example Hindly and Seldin, "Introduction to Combinators and Lambda-Calculus" for a formal treatment.)

Schonfinkel (1924) first described combinators. They provide a way of avoiding variables altogether in lambda calculus. (Variables cause a lot of complications in describing the rewrite rules, principally because of the need to avoid accidental collisions of variable names.)

Basic Idea

- Abstract away all variables, leaving code that can be executed on a simple machine. Use combinators to perform the abstraction.
- Result will be a graph.
- Execute using an abstract machine that does graph reduction.

Abstracting Variables Out

If we have a function definition:

```
def f x = ...
```

We first replace all functions in the definition of `f` with their curried versions:

```
def f x = E
```

Now we can abstract out the references to x:

```
def f = [x]E
```

Where the abstraction operation has the property:

`([x]E)x = E` (extensibility condition)

Notice that `[x]E` is similar to `(lambda (x) E)`, but `[x] E` is a textual, compile time operation.

Definition of Combinators

Turner now defines a basic set of combinators: S, K, and `I` (see Turner, pg. 34)

```
S f g x = f x (g x)
   K x y = x
      I x = x
```

(In fact we only need S and K, since SKK=I)

Rules for abstracting x

```
[x] (E1 E2) = S ([x] E1) ([x] E2)
       [x] x = I
       [x] y = K y,
```
where y is a constant or variable and x not equal to y

Proofs of Correctness

Take LHS of first rule, and apply it to x:
`[x] (E1 E2) x = E1 E2` (by extensionality)

RHS of first rule:
```
S ([x] E1) ([x] E2) x
      = (([x] E1) x) (([x] E2) x)
      = E1 E2
```
(extensionality, twice)

LHS of second rule, applied to x:
`([x] x) x = x` (extensionality)

RHS of second rule
`I x = x` (definition of I)

LHS of third rule:
`([x] y) x = y` (extensionality)

RHS of third rule
`K y x = y` (definition of K)

Example: successor function

```
suc x = plus 1 x

suc = [x] (plus 1 x)
   => S ([x] (plus 1)) ([x] x)
   => S (S (K plus) (K 1)) I
```

This is correct but long-winded. We add additional combinators B and C to get more compact graphs:

```
B f g x = f (g x)
C f g x = f x g
```

with these additions the successor function compiles to
```
suc = plus 1
```


Example: average function

avg x y = (x+y)/2

replace + and / by curried versions:

avg x y = divide (plus x y) 2

abstract y (treating x as a constant)

avg x = [y] (divide (plus x y) 2)
= S ([y] (divide (plus x y))) ([y] 2)
= S (S ([y] divide) ([y] (plus x y))) ([y] 2)
= S (S ([y] divide) ([y] (plus x y))) (K 2)
= C (S ([y] divide) ([y] (plus x y))) 2
= C (S (K divide) ([y] (plus x y))) 2
= C (S (K divide) (S ([y] (plus x)) ([y] y))) 2
= C (B divide (S ([y] (plus x)) ([y] y))) 2
= C (B divide (S (S ([y] plus) ([y] x)) ([y] y))) 2
= C (B divide (S (S (K plus) (K x)) I)) 2
= C (B divide (S (K (plus x)) I)) 2
= C (B divide (plus x)) 2

avg = [x] (C (B divide (plus x)) 2)
avg = [x] (C (B divide (plus x)) 2)
  = S ([x] (C (B divide (plus x)))) ([x] 2)
  = S ([x] (C (B divide (plus x)))) (K 2)
  = C ([x] (C (B divide (plus x)))) 2
  = C (S ([x] C) ([x] (B divide (plus x)))) 2
  = C (S (K C) ([x] (B divide (plus x)))) 2
  = C (B C ([x] (B divide (plus x)))) 2
  = C (B C (S ([x] B divide) ([x] (plus x)))) 2
  = C (B C (S (S ([x] B) ([x] divide)) ([x] (plus x)))) 2
  = C (B C (S (S (K B) (K divide)) ([x] (plus x)))) 2
  = C (B C (S (K B divide) ([x] (plus x)))) 2
  = C (B C (S (K B divide) ([x] (plus x)))) 2
  = C (B C (B (B divide) ([x] (plus x)))) 2
  = C (B C (B (B divide) (S ([x] plus) ([x] x)))) 2
  = C (B C (B (B divide) (S (K plus) I))) 2
  = C (B C (B (B divide) plus)) 2

ugh!

Turner also introduces combinators for pattern matching

Y combinator -- finds fixedpoints
 Y f = f (Y f)
used in local recursions
 E where x = ... x ...

example:
ham = 1: my_merge ham2 (my_merge ham3 ham5)
   where
   ham2 = map (*2) ham
   ham3 = map (*3) ham
   ham5 = map (*5) ham

S-K reduction machine

graph rewriting machine to interpret combinator code

Miranda uses normal order evaluation -- go down left branch of the tree
until a combinator is found. Apply it to the args, and replace that node
with the result.

S-K Reduction Example:

`suc 2 where suc x = 1+x`
(from Turner paper)
The compiler transforms this to:

```
([suc] (suc 2)) ([x] (plus 1 x))
```

We then convert to combinator form:

```
S ([suc] suc) ([suc] 2) ([x] (plus 1 x))

S I (K 2) ([x] (plus 1 x))

C I 2 ([x] (plus 1 x))

C I 2 (S ([x] (plus 1)) ([x] x))

C I 2 (S (S ([x] plus) ([x] 1)) ([x] x))

C I 2 (S (S (K plus) (K 1)) ([x] x))

C I 2 (S (K (plus 1)) ([x] x))

C I 2 (S (K (plus 1)) I)

C I 2 (plus 1)
```

We can now evaluate this using a series of graph transformations:

Remember rule for C:

```
C f g x = f x g
```

Initially:

```
C I 2 (plus 1)
```

Using the rule for C

```
I (plus 1) 2
```

Using the rule for I

```
plus 1 2
```

Using the rule for plus:

```
3
```


Self-Optimizing Code

Simple example:

code is a built-in function that maps characters to numbers (ascii codes)

e.g. code '0' = 48

```
makedigit n = code n - code '0'
```

After the first evaluation the expression (code '0') will be ***replaced*** by 48

another example:

```
foldr op r = f
  where
  f [] = r
  f (a:x) = op a(f x)

sum = foldr (+) 0
```

after the first evaluation of sum, it will be rewritten to the equivalent of

```
sum [] = 0
sum (a:x) = a+sum x
```


Later Developments

lambda lifting

supercombinators (combinators are abstracted from user's program)

G machine

strictness analysis

compilation to conventional single-processor architectures

compilation for conventional parallel hardware

special-purpose hardware for parallel graph reduction

lambda lifting:

if we have a local function definition with free variables, we can move

it to the top level by adding additional arguments that are then applied to the free variables

example:

`f x = e` where e contains a free variable y

define a new function

```
f' y x = e
```

at the top level

Replace calls to f by

```
f' y
```

supercombinators: combinators are abstracted from user's program

Johnnson et al, Chalmers University

this technique is used in e.g. one of the Haskell implementation