Implementation of Functional Languages

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.)

Schönfinkel (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:

```edef f x = ...```

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

```edef f x = E```

Now we can abstract out the references to `x`:

```edef 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
\[ \text{suc} \ x = \text{plus} \ 1 \ x \]
\[ \text{suc} = [x] \ (\text{plus} \ 1 \ x) \]
\[ \rightarrow S \ ([x] \ (\text{plus} \ 1)) \ ([x] \ x) \]
\[ \rightarrow S \ (S \ (K \ \text{plus}) \ (K \ 1)) \ I \]

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
\[ \text{suc} = \text{plus} \ 1 \]

Example: average function
\[ \text{avg} \ x \ y = (x+y)/2 \]

replace + and / by curried versions:
\[ \text{avg} \ x \ y = \text{divide} \ (\text{plus} \ x \ y) \ 2 \]

abstract y (treating x as a constant)
\[ \text{avg} \ x = [y] \ (\text{divide} \ (\text{plus} \ x \ y)) \ 2 \]
\[ = S \ ([y] \ (\text{divide} \ (\text{plus} \ x \ y))) \ ([y] \ 2) \]
\[ = S \ (S \ ([y] \ \text{divide}) \ ([y] \ (\text{plus} \ x \ y)))) \ ([y] \ 2) \]
\[ = S \ (S \ ([y] \ \text{divide}) \ ([y] \ (\text{plus} \ x \ y)))) \ 2 \]
\[ = C \ (S \ (K \ \text{divide}) \ ([y] \ (\text{plus} \ x \ y)))) \ 2 \]
\[ = C \ (S \ (K \ \text{divide}) \ (S \ ([y] \ (\text{plus} \ x)))) \ ([y] \ 2) \]
\[ = C \ (B \ \text{divide} \ (S \ ([y] \ (\text{plus} \ x)))) \ ([y] \ 2) \]
\[ = C \ (B \ \text{divide} \ (S \ (S \ (K \ \text{plus}) \ (K \ l))))) \ 2 \]
\[ = C \ (B \ \text{divide} \ (S \ (K \ (\text{plus} \ x)))) \ 2 \]
\[ = C \ (B \ \text{divide} \ (S \ (K \ (\text{plus} \ x)))) \ 2 \]
avg = [x] (C (B divide (plus x)) 2)
= S ([x] (C (B divide (plus x)))) ( [x] 2 )
= C ([x] (C (B divide (plus x)))) ( [x] 2 )
= C (S ([x] C) ([x] (B divide (plus x)))) 2
= C (S (K C) ([x] (B divide (plus x)))) 2
= C (B (S ([x] (B divide (plus x))))) 2
= C (B (S ([x] B divide) ([x] (plus x)))) 2
= C (B (S (S ([x] B) ([x] divide)) ([x] (plus x)))) 2
= C (B (S (K B) (K divide)) ([x] (plus x)))) 2
= C (B (B (B divide) ([x] (plus x)))) 2
= C (B (B (B divide) (S ([x] divide) ([x] (plus x))))) 2
= C (B (B (B divide) (S (S ([x] divide) ([x] (plus x))))) 2
= C (B (B (B divide) (S (K divide) ([x] (plus x))))) 2
= C (B (B (B divide) (S (K (plus 1)) ([x] x)))) 2
= C (B (B (B divide) (S (K (plus 1)))) 2
= C (B (B (B divide) (plus 1)) 2

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:
\[ C \ I \ 2 \ (\text{plus}\ 1) \]

Using the rule for C
\[ I \ (\text{plus}\ 1) \ 2 \]

Using the rule for I
\[ \text{plus}\ 1 \ 2 \]

Using the rule for plus:
\[ 3 \]

Self-Optimizing Code

Simple example:
\[ \text{code is a built-in function that maps characters to numbers (ascii codes)} \]
\[ \text{e.g. code '0' = 48} \]
\[ \text{makedigit n = code n - code '0'} \]

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

another example:
\[ \text{foldr op r = f} \]
\[ \text{where} \]
\[ f \ [\] = r \]
\[ f \ (a:x) = \text{op a(f x)} \]
\[ \text{sum} = \text{foldr} \ (\text{+}) \ 0 \]

after the first evaluation of sum, it will be rewritten to the equivalent of
\[ \text{sum} \ [\] = 0 \]
\[ \text{sum} \ (a:x) = a + \text{sum} \ x \]

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

e.g. 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