CENG444: Compiling Functional Languages

Cem Bozşahin

Cognitive Science Department, Informatics Institute,
Middle East Technical Univ. (ODTÜ), Ankara

This is a short summary of Simon Peyton Jones's 1987 book *The Implementation of Functional Programming Languages*, chapters 2, 12, 13 and 16.

- *λ-calculus*: Uniform representation and evaluation of functions.
- *Graph reduction*: A simple Virtual Machine for FPLs.
- *Supercombinators:* Life without global variables.
- **SK**-*machine:* Life without variables and intermediate abstractions.

  It is a real machine, not abstract or toy! (See Clarke et al. 1980)

λ-calculus is a uniform way to represent functions, without a need for names.

$f(x,y) = x + y$ $\qquad$ $\lambda x.\lambda y. + x\, y$

Application of functions to arguments:

$f(3,4) = 7$

$\lambda x.\lambda y.(+x\, y)3\ 4 = \lambda y.(+3\ y)4 = (+3\ 4) = 7$

Functions can be arguments:

$f(g,x) = g(x)$ $\qquad$ $\lambda g.\lambda x.gx$

Lambda calculus

Function notation (λ notation)

Function abstraction and application (β conversion)

Function equivalence (α and η conversions)

β-conversion: $(\lambda x.E)M \leftrightarrow_{\beta} E[M/x]$

$E[M/x]$: expression E where M is substituted for **free occurences** of x.

Some common data structures in their lambda calculus interpretation:

$\textbf{CONS} = \lambda a.\lambda b.\lambda f.f\ a\ b$

$\textbf{HEAD} = \lambda c.c(\lambda a.\lambda b.a)$

$\textbf{TAIL} = \lambda c.c(\lambda a.\lambda b.b)$

So that we have

CONS$x\ y$ = list with head x and tail y

$\textbf{HEAD}(\textbf{CONS}x\ y) = x$

$\textbf{TAIL}(\textbf{CONS}x\ y) = y$

Substitute definitions to obtain expected behaviour:

$$\mathbf{HEAD}(\mathbf{CONS}x\ y) = (\lambda c.c(\lambda a.\lambda b.a))(\mathbf{CONS}x\ y)$$

$$= \mathbf{CONS}xy(\lambda a.\lambda b.a) = (\lambda a.\lambda b.\lambda f.f\ a\ b)\ x\ y\ (\lambda a.\lambda b.a)$$

$$= (\lambda b.\lambda f.f\ x\ b)y(\lambda a.\lambda b.a) = (\lambda f.f\ x\ y)(\lambda a.\lambda b.a)$$

$$= (\lambda a.\lambda b.a)\ x\ y = (\lambda b.x)\ y = x$$

All common PL expressions can be given a λ-calculus interpretation.

Expected behaviour:

$\mathbf{IF}\,\mathbf{TRUE}\,E_1\ E_2 = E_1$

$\mathbf{IF}\,\mathbf{FALSE}\,E_1\ E_2 = E_2$

The following definitions give that behaviour:

$\mathbf{IF} = \lambda f.\lambda d.\lambda e.fde$

$\mathbf{TRUE} = \lambda a.\lambda b.a$

$\mathbf{FALSE} = \lambda a.\lambda b.b$

Ex: $\textbf{IF}\,\textbf{TRUE}x\ y = x$

$\textbf{IF}\,\textbf{TRUE}x\ y = (\lambda f.\lambda d.\lambda e.f)\textbf{TRUE}x\ y$

$= (\lambda d.\lambda e.\textbf{TRUE}d\ e)x\ y = (\lambda e.\textbf{TRUE}x\ e)y$

$= (\lambda e.\textbf{TRUE}x\ e)y = \textbf{TRUE}x\ y$

$= (\lambda a.\lambda b.a)xy = (\lambda b.x)y = x$

What if we defined $\textbf{IF} = \lambda f.f$?

A combinator is a lambda term without free variables.

We can eliminate renaming of variables, because it's a theorem of λ-calculus.

The following are equivalent functions (modulo variable names). They *behave* the same.

$\lambda x. + \ 1 \ x \qquad \lambda y. + \ 1 \ y$

α-conversion: $(\lambda x.M) \leftrightarrow_\alpha (\lambda y.M[y/x])$

These are equivalent too, because they behave the same as well:

$\lambda x. + \ 1 \ x \qquad + \ 1$

η-conversion: $\lambda x.F\ x \leftrightarrow_\eta F$ (if x does not occur free in F)

note: $\lambda x. *\ x\ x$ is not eta reducible to $(*\ x)$

How do we do recursion without names? The fixpoint combinator **Y** takes care of that:

$\mathbf{FAC} = \lambda n.\mathbf{IF}(=\ n\ 0)1(*\ n(\mathbf{FAC}(-\ n\ 1)))$

β conversion in the direction of abstraction gives:

$\mathbf{H} =_{\beta} \lambda fac.(\lambda n.\mathbf{IF}(=\ n\ 0)\ 1\ (*\ n\ (fac(-\ n\ 1))))\ \ \mathbf{FAC}$

$\mathbf{FAC} = \mathbf{H}\ \mathbf{FAC}$, where **H** is $\lambda fac.(\cdots)$

FAC is called the fixpoint of **H.**

Note that **H** does not refer to **H**, so our recurring name problem is solved.

Big question: Can we do this for *any* function **H**? YES!!

All we need is a definition **Y** that takes a function and returns its fixpoint:

$\mathbf{YH} = \mathbf{H}(\mathbf{YH})$

Amazingly, **Y** can be defined as a lambda abstraction, i.e., WITHOUT RECURSION.

It's called the fixpoint combinator:

$\mathbf{Y} = \lambda h.(\lambda x.h\,(x\ x))\ (\lambda x.h(x\ x))$

Now, def. of **FAC** is non-recursive because

$\mathbf{FAC} = \mathbf{H}\ \mathbf{FAC}$ and

$\mathbf{YH} = \mathbf{H}(\mathbf{YH})$

Therefore,

$\mathbf{FAC} = \mathbf{Y}\ \mathbf{H}$

Does **FAC** behave correctly?

$\mathbf{FAC} p = \mathbf{YH} p =$

$\mathbf{H}(\mathbf{YH})\ p =$

$\lambda fac.(\lambda n.\mathbf{IF}(=\ n\, 0)\ 1\ (*\ n\ (fac(-\ n\, 1)))))(\mathbf{YH})\ p =$

$\lambda n.\mathbf{IF}(=\ n\, 0)\ 1\ (*\ n\ ((\mathbf{YH})\ (-\ n\, 1)))\ p =$

$\mathbf{IF}(=\ p\, 0)\ 1\ (*\ p\ ((\mathbf{YH})\ (-\ p\, 1))) =$

In case $p \neq 0$, rewrite **Y H** as **H** (**Y H**)

So that **H** can get its argument $(-\ p\, 1)$

Here's evaluation of 'FAC 1' to show the effect of recursion:

$\mathrm{FAC} = \mathbf{Y}\,\mathrm{H}$
$\mathrm{FAC}\ 1 = \mathbf{Y}\ \mathrm{H}\ 1 =$
$\mathrm{H}\ (\mathbf{Y}\ \mathrm{H})\ 1 =$
$\lambda f \lambda n.\ \mathrm{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (f\ (-\ n\ 1)))\ (\mathbf{Y}\ \mathrm{H})\ 1 =$
$\lambda n.\ \mathrm{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (\mathbf{Y}\ \mathrm{H}\ (-\ n\ 1)))\ 1 =$
$\mathrm{IF}\ (=\ 1\ 0)\ 1\ (\times\ 1\ (\mathbf{Y}\ \mathrm{H}\ (-\ 1\ 1))) =$
$\times\ 1\ (\mathbf{Y}\ \mathrm{H}\ 0) =$
$\times\ 1\ (\mathrm{H}\ (\mathbf{Y}\ \mathrm{H})\ 0) =$
$\times\ 1\ \ ((\lambda f \lambda n.\ \mathrm{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (f\ (-\ n\ 1))))\ (\mathbf{Y}\ \mathrm{H})\ 0) =$
$\times\ 1\ \ ((\lambda n.\ \mathrm{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (\mathbf{Y}\ \mathrm{H}\ (-\ n\ 1))))\ 0) =$
$\times\ 1\ \ (\mathrm{IF}\ (=\ 0\ 0)\ 1\ (\times\ 0\ (\mathbf{Y}\ \mathrm{H}\ (-\ 0\ 1)))) =$
$\times\ 1\ 1 =$
1

Does **Y** behave correctly?

$\mathbf{YH} = (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h(x\ x)))\ \mathbf{H} =$

$(\lambda x.\mathbf{H}\ (x\ x))\ (\lambda x.\mathbf{H}\ (x\ x)) =$ (apply 1st to 1nd):

$\mathbf{H}((\lambda x.\mathbf{H}\ (x\ x))\ (\lambda x.\mathbf{H}\ (x\ x))) =$

$\mathbf{H}(\mathbf{Y}\ \mathbf{H})$

Summary:

- all data can be represented as lambda abstractions.

- all functions can be represented as lambda abstractions.

How do we evaluate them?

Methods for β-reduction:

With variables : Graph reduction

With piece-meal internal abstractions: Supercombinators

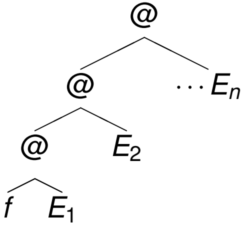
Without variables, with only compile-time abstractions: **SK** machine

Graph reduction as program execution

The strategy of choice in LISP.

General template of a program in FPL: $f\ E_1\ E_2\ \cdots E_n$

In Graph notation:



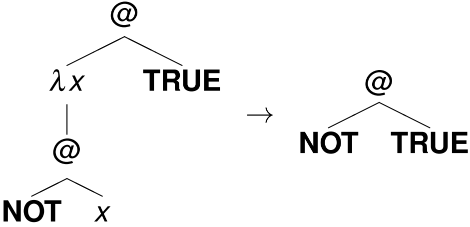
f may be 1) data; 2) built-in function with $k \leq n$ arguments; 3) lambda abstraction; 4) variable

case 1: Program is done.

case 2: Redex is $(f\ E_1 \cdots E_k)$

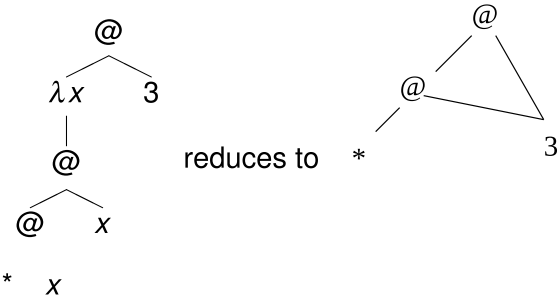
case 3: Redex is $(f\ E_1)$

case 4: error (because the variable is free—it's leftmost) ex: a graph with lambdas: $(\lambda x.\mathbf{NOT}x)$ **TRUE** $\rightarrow_\beta$ **NOT TRUE**



Why is it called *graph* reduction?

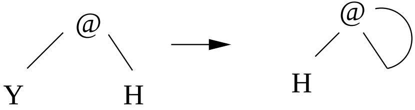
$(\lambda x. * \ x \ x)3 = 9$: Substitute pointers to the argument for formal parameter x



Imagine a function $(\lambda x.E\ x\ x\ x\ x)M$ where M is a huge function.

Without pointer substitution, we would evaluate M four times for no reason, because it will always yield the same value.

Y reduction is best described as a graph as well:



$\mathbf{Y}\ f = f\ (\mathbf{Y}\ f) = f\ (f\ (\mathbf{Y}\ f)) =$

$f\ (f\ (f\ (\mathbf{Y}\ f)))$

Supercombinators:

In Graph Reduction, finding a lambda body to reduce during execution is costly; it requires a tree walk at every step.

$**S** is called a supercombinator if

$\$\mathbf{S} = \lambda x_1.\lambda x_2 \ldots \lambda x_n.E$

and

1) $**S** has no free variables

2) E is not a lambda abstraction

3) any lambda expression in E is a supercombinator

4) $n \geq 0$

Because of (2), supercombinator's arguments can be supplied all at once.

ex: $\lambda x. * \ x \ x$

$\lambda x.\lambda y. - \ x \ y$

5

The following are not supercombinators:

$\lambda x. - \ y \ x$

$\lambda f.f(\lambda x.f \ x \ x)$

Combinators: A lambda expression with no occcurences of a free variable.

Combinators have by convention names, such as **B**, **S**, **K**, **I**, **C**, **T** etc.

Less than a handful is enough to write ANY lambda expression (assuming no free variables) WITHOUT VARIABLES, as Curry and Feys showed in 1958.

Amazingly, two are enough: **S** and **K**.

Supercombinators have names made-up during compilation; they are not primitives (hence the $**X** notation).

Combinatory Logic of Curry & Feys is equivalent to λ-calculus and Turing Machines.

The (boxing) strategy in Supercombinator Compilation:

1) Derive a set of supercombinator definitions (upper box)

2) Single Expression to be evaluated (lower box)

3) Compile until bottom expression has no lambdas, just evaluations.

Supercombinator definitions
Expresion to be evaluated

Ex: Program $(\lambda x.\lambda y.* \ x \ y)3 \ 4$ compiles to

\$**XY** $x \ y \ = * \ x \ y$
\$**XY**3 4

All arguments must be supplied; expression below is not a supercomb.

$\$\mathbf{XY}\ x\ y\ = *\ x\ y$
$\$\mathbf{XY}3$

Ex: Compiling the program: **FAC** 5

$\mathbf{FAC} = \lambda n.\mathbf{IF}(=\ n\ 0)\ 1\ (*\ n\ (\mathbf{FAC}(-\ n\ 1)))$

β conversion in the direction of abstraction gives:

$=_\beta \lambda fac.(\lambda n.\mathbf{IF}(=\ n\ 0)\ 1\ (*\ n\ (fac(-\ n\ 1))))\ \ \mathbf{FAC}$

Let $F = \lambda fac.(\lambda n.\mathbf{IF}(=\ n\,0)\ 1\ (*\ n\,(fac(-\ n\,1))))$

Not a supercombinator: inner lambda expr has a free variable (*fac*)

By β-abstraction, inner lambda body is equivalent to

$\$\mathbf{N}\ fac = \lambda w.\lambda n.\mathbf{IF}(=\ n\,0)\ 1\ (*\ n\,(w\ (-\ n\,1))))\ fac$

and $\mathbf{N} is a supercombinator

The compiled boxes look like:

$\mathbf{FAC} = \cdots$ $\$\mathbf{N} w\ n\ = \mathbf{IF} (=\ n\ 0)\ 1\ (*\ n\ (w\ (-\ n\ 1)))$
$\lambda fac.(\lambda n.\$\mathbf{N}\ fac\ n)\ \mathbf{FAC}\ 5$

Compile until bottom expression has no lambdas, just evaluations.

This is possible because the lambda body $\lambda fac.(\cdots)$ is a supercombinator.

$\mathbf{FAC} = \cdots$ $\$\mathbf{N} w\ n\ = \mathbf{IF} (=\ n\ 0)\ 1\ (*\ n\ (w\ (-\ n\ 1)))$ $\$\mathbf{NF}\ fac\ n\ = \$\mathbf{N}\ fac\ n$
$\$\mathbf{NF}\ \mathbf{FAC}\ 5$

By η-conversion, we can conclude $\$\mathbf{NF} = \$\mathbf{N}$, and eliminate one of them.

Having **FAC** at the bottom might look circular, but We know that **FAC** $=$ **Y H** where **Y** is the fixpoint combinator, and

$$\mathbf{H} = \lambda fac.(\lambda n.\mathbf{IF}(=\ n\,0)\ 1\ (*\ n\,(fac(-\ n\,1)))) = \$\mathbf{N}\ \ fac\ n$$

We can revise the definitions to reflect that.

$$\boxed{\begin{array}{l} \mathbf{Y} = \cdots \\ \$\mathbf{H}\ fac\ n\ =\ \$\mathbf{N}\ fac\ n \\ \$\mathbf{N} w\ n\ = \mathbf{IF}(=\ n\,0)\ 1\ (*\ n\,(w\ (-\ n\,1))) \\ \hline \$\mathbf{N}\ (\mathbf{Y}\ \mathbf{H})\ 5 \end{array}}$$

Therefore $\$\mathbf{H} = \$\mathbf{N}$.

Final compilation is:

$$\boxed{\begin{array}{l} \mathbf{Y} = \cdots \\ \$\mathbf{N} w\ n\ = \mathbf{IF}(=\ n\,0)\ 1\ (*\ n\,(w\ (-\ n\,1))) \\ \hline \$\mathbf{N}\ (\mathbf{Y}\ \$\mathbf{N})\ 5 \end{array}}$$

SK Machine

Miranda and SASL use **SK** virtual machines.

If we don't want ANY variables, including those in the abstractions, we can use the universal combinators **S** and **K**.

Note that, Supercombinator Method's overhead for keeping environment is minimal but NOT ZERO: there are only local variables, values of which are all supplied at the same time.

SK systems allow any function to be written without variables, in terms of **S**, **K** and built-in functions (like **Y**) and constants.

It is easier to see them at work, starting with **I** as well.

$$\mathbf{S} f\ g\ x \ \rightarrow\ f\ x\ (\ g\ x\ )$$

$$\mathbf{K}\ x\ y \ \rightarrow\ x$$

$$\mathbf{I}\ x \ \rightarrow\ x$$

S allows us to see that the following functions are equivalent:

$F_1 = \lambda x.\,(e_1\ e_2\ )$

$F_2 = \mathbf{S}\,(\lambda x.e_1)\,(\lambda x.e_2)$

$F_1\ a = F_2\ a$

If e_1 and e_2 are lambda expressions, **S** allows us to push down x one level more, until it is no longer possible:

$\lambda x.\,(\ ((\lambda y.e_3\ e_4\ )\ ((\lambda y.e_5\ e_6\ ))\ )$

$\mathbf{S}\,(\mathbf{S}\,(\lambda x.e_3)\,(\lambda x.e_4))(\mathbf{S}\,(\lambda x.e_5)\,(\lambda x.e_6))$

If expresssions *e* are simplest, they can be

$\lambda x.x = \mathbf{I}$

$\lambda x.c = \mathbf{K}\ c$

We can use these equivalences as *program transformations*.

ex: $(\lambda x. *\ x\ x)\ 5$

$= \mathbf{S}\ (\lambda x. *\ x)\ (\lambda x.x)\ 5$

$= \mathbf{S}\ (\mathbf{S}\ (\lambda x.*)\ (\lambda x.x))\ (\lambda x.x)\ 5$

$= \mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ *)\ \mathbf{I})\ \mathbf{I}\ 5$

I is actually not needed, all we need is two!

$\mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$

Try **I** *a* and **S K K***a*

In case of multiple lambdas, you must push in the innermost one first: take $\lambda x \lambda y. + xy$

$\lambda x \lambda y. + xy = \lambda x(\lambda y((+x)y))$ by associativity

$\lambda x(\mathbf{S}(\lambda y. + x)(\lambda y.y)) =$

$\lambda x(\mathbf{S}(\mathbf{S}(\lambda y.+)(\lambda y.x))(\mathbf{SKK})) =$ now push x inward

$\mathbf{S}(\lambda x.\mathbf{S}(\mathbf{S}(\mathbf{K}+)(\mathbf{K}x)))(\lambda x.\mathbf{SKK}) =$

$\mathbf{S}(\lambda x.\mathbf{S}(\mathbf{S}(\mathbf{K}+)(\mathbf{K}x)))(\mathbf{S}(\lambda x.\mathbf{SK})(\lambda x.\mathbf{K})) =$

$\mathbf{S}(\lambda x.\mathbf{S}(\mathbf{S}(\mathbf{K}+)(\mathbf{K}x)))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

$\mathbf{S}(\mathbf{S}(\lambda x.\mathbf{S})(\lambda x.\mathbf{S}(\mathbf{K}+)(\mathbf{K}x)))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\lambda x.\mathbf{S}(\mathbf{K}+))(\lambda x.\mathbf{K}x)))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

you can use eta-reduction too; try on $\lambda x.\mathbf{K}x$

$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\lambda x.\mathbf{S}(\mathbf{K}+))\mathbf{K}))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{S}(\lambda x.\mathbf{S})(\lambda x.\mathbf{K}+))\mathbf{K}))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\lambda x.\mathbf{K})(\lambda x.+)))\mathbf{K}))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{K}+)))\mathbf{K}))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KK}))(\mathbf{KK})) =$

Phew!!

Now, try $(\lambda x.\lambda y. + xy)3\,4 = +3\,4$ and above to see their equivalence.

Can we write **Y** with **S** and **K**? Certainly!

$$\mathbf{Y} = \mathbf{SSK}(\mathbf{S}(\mathbf{K}(\mathbf{SS}(\mathbf{S}(\mathbf{SSK}))))\mathbf{K})$$

It is not pretty, but it does the job.

Note that although the programmer can use variables in the source language for convenience,

the compiler completely eliminates them via program transformations.

There is no environment to keep during run-time.

For an entertaining story of **S**tarling and **K**estrel, try Ray Smullyan's 1985 book.

The Turkish equivalents are, imo, **S**aka and **K**erkenez.

Clarke, T. J., P. J. Gladstone, C. D. MacLean, and A. C. Norman (1980). SKIM - the S, K, I reduction machine. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP '80, New York, NY, USA, pp. 128–135. ACM.

Smullyan, R. (1985). *To Mock a Mockingbird*. New York: Knopf.