Church encoding,
the CC machine,
and the CK machine

CS245 — Fall 2019

Authors: Kris Micinski + Thomas Gilray
Writing a functional compiler

Functional compilers translate input expressions in one intermediate representation (intermediate language) into equivalent expressions in another, simpler IR.

\[ \text{translate} : \text{IR}_0 \rightarrow \text{IR}_1 \]
Church encoding

*Church encoding* is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.
Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, ..., as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

“If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/#if/etc...)?”
Church encoding

*Church encoding* is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, ..., as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.
Project 4: Church compiler

Your goal is to compile a significant subset of Scheme (with a few simplifications) down to the lambda calculus. This requires you to desugar (simplify) most forms, to curry all functions, and to church-encode all non-function values.
\[ e ::= (\text{letrec } ([x (\lambda (x \ldots) e)]) \]
| \[(\text{let } ([x \ e] \ldots) \ e)\]
| \[(\lambda (x \ldots) \ e)\]
| \[(\ e\ e\ \ldots)\]
| \[x\]
| \[(\text{if } e\ e\ e\ e)\]
| \[(\text{prim } e\ e) | (\text{prim } e)\]
| \[d\]

\[ d ::= \mathbb{N} | \#t | \#f | '()\]
\[ x ::= <\text{vars}>\]
\[ \text{prim} ::= + | - | * | \text{not} | \text{cons} | \ldots\]
e ::= (letrec ([x (lambda (x …) e)])
| (let ([x e] …) e)
| (lambda (x …) e)
| (e e …)
| x
| (if e e e)
| (+ e e) | (* e e)
| (cons e e) | (car e) | (cdr e)

x ::= <vars>
e ::= (lambda (x) e)
| (e e)
| x
Scheme IR \[\rightarrow\] \[2\]

\[\lambda\text{-calculus}\]

\[\lambda\text{-calculus}\] \[\rightarrow\] \[\text{interp}\]

\[\text{interp}\] \[\rightarrow\] \[
(\lambda (f) \\
(\lambda (x) \\
(f (f x))))
\]
Today, we’ll start with:

\[
e ::= (\text{letrec } ([x (\lambda (x \ldots) e)]) ) \\
| (\text{let } ([x e] \ldots) e) \\
| (\lambda (x \ldots) e) \\
| (e e \ldots) \\
| x \\
| (\text{if } e e e e) \\
| (+ e e) | (*) e e e \\
| (\text{cons } e e) | (\text{car } e) | (\text{cdr } e) \\
| d
\]

\[
d ::= \mathbb{N} | \#t | \#f | \text{'}() \\
x ::= <\text{vars}>
\]
Desugaring Let
(let ([x e] ...) ebody)
(let ([x e] ... ebody))

((\ (x ... ) ebody) e ... )
Currying
\[(\lambda (x\ y\ z)\ e) \rightarrow (\lambda (x)\ (\lambda (y)\ (\lambda (z)\ e))))\]

\[(\lambda (x)\ e) \rightarrow (\lambda (x)\ e)\]

\[(\lambda ()\ e) \rightarrow (\lambda (\_ )\ e)\]
\[(f\ a\ b\ c\ d) \quad \longrightarrow \quad (((((f\ a)\ b)\ c)\ d)\)

\[(f\ a) \quad \longrightarrow \quad (f\ a)\]

\[(f) \quad \longrightarrow \quad (f\ (\lambda\ (x)\ x))\]
\[ e ::= (\text{letrec } ([x (\lambda (x) e)]) ) \\
| (\lambda (x) e) \\
| (e \ e) \\
| x \\
| (\text{if } e \ e \ e \ e) \\
| ((+ e) e) \ | ((\ast e) e) \\
| ((\text{cons } e) e) \ | (\text{car } e) \ | (\text{cdr } e) \\
| d \\
\]

\[ d ::= \mathbb{N} \ | \ #t \ | \ #f \ | \ '() \]

\[ x ::= <\text{vars}> \]
Conditionals & Booleans
(if #t e_T e_F)

\[
\downarrow
\]

\begin{align*}
\text{e}_T
\end{align*}

\hfill

(\text{if} \ #f \ e_T \ e_F)

\[
\downarrow
\]

\begin{align*}
\text{e}_F
\end{align*}
$$\((\lambda (t f) t) \; e_T \; e_F\) \quad \((\lambda (t f) f) \; e_T \; e_F\)$$

$$\quad \downarrow \quad \downarrow$$

$$\((\lambda (t f) t) \; v_T \; v_F\) \quad ((\lambda (t f) f) \; v_T \; v_F)$$

$$\quad \downarrow \quad \downarrow$$

$$v_T \quad v_F$$
What issues arise with this encoding?
$$((\lambda (t \ f) \ t) \ e_T \ \Omega)$$
\(((\lambda \ (t \ f) \ (t)) \ (\lambda \ () \ e_T) \ (\lambda \ () \ \Omega))\)
\[ e ::= (\text{letrec} ([x (\lambda (x) e)]) )
| (\lambda (x) e)
| (e e)
| x
| ((+ e) e) | ((* e) e)
| ((\text{cons} e) e) | (\text{car} e) | (\text{cdr} e)
| d \]

\[ d ::= \mathbb{N} | '() \]

\[ x ::= <\text{vars}> \]
Natural Numbers
Hint: turn all nouns into verbs!

(Focus on the behaviors that are implicit in values.)
\((\lambda (f) \ (\lambda (x) \ (f^N \ x)))\)

0: \((\lambda (f) \ (\lambda (x) \ x))\)

1: \((\lambda (f) \ (\lambda (x) \ (f \ x)))\)

2: \((\lambda (f) \ (\lambda (x) \ (f \ (f \ x))))\)

3: \((\lambda (f) \ (\lambda (x) \ (f \ (f \ (f \ x))))))\)
church+ = (λ (n) (λ (m) 
(λ (f) (λ (x) 
...))))

church+ = (\( \lambda \) (n) (\( \lambda \) (m) 
(\( \lambda \) (f) (\( \lambda \) (x) 
(((n f) (((m f) x))))))))
\textit{church}^* = (\lambda (n) (\lambda (m) \\
(\lambda (f) (\lambda (x) \\
...))))
church* = (λ (n) (λ (m) 
  (λ (f) (λ (x) 
    ((n (m f)) x))))))
f_{N^M} = f_{N^*M}
\[ e ::= (\text{letrec} ([x (\lambda (x) e)]) ) \]
\| (\lambda (x) e) \\
\| (e \ e) \\
\| x \\
\| ((\text{cons} \ e) \ e) \ | (\text{car} \ e) \ | (\text{cdr} \ e) \\
\| d \\
\]

\[ d ::= '() \]

\[ x ::= <\text{vars}> \]
Lists
The fundamental problem:

We need to be able to case-split.

The solution:

We take two callbacks as with #t, #f!
′() = (λ (when-cons) (λ (when-null) (when-null)))

(cons a b) = (λ (when-cons) (λ (when-null) (when-cons a b)))
Try an Example. How can we define null?
Try an Example. How can we define null?

\[
\text{church:null?} = (\lambda (p) \\
(p (\lambda (a \ b) \ #f) \\
(\lambda () \ #t)))
\]
\[ e ::= (\text{letrec } ([x \ (\lambda (x) \ e)]) ) \]
| (\lambda (x) \ e)
| (e \ e)
| x

\[ x ::= \langle \text{vars} \rangle \]
Infinity
\[ ((\lambda (x) (x \ x)) \ (\lambda (x) (x \ x))) \]
Key: U takes a function and calls it on itself

\[ (((\lambda \ (x) \ (x \ x)) \ (\lambda \ (x) \ (x \ x))) \ U) \]
(define U (λ (f) (f f)))

(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))

(let ([fib (U (lambda (f)
                  (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))]
       (fib 3))

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)
(define U (λ (f) (f f)))

(letrec ([(fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1))))))])
  (fib 3))

(let ([(fib (U (lambda (f)
                   (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))))
  (fib 3))

What can I type right here to make fib work?

(f f)
Y combinator
(letrec ([fact (λ (n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))])

(fact 5))
Key idea: instead of

(let ([mk (\ (mk) (\ (n) 
  (if (= n 0) 
    1 
    (* n ((mk mk) (- n 1))))))))

((mk mk) 5))
\[(Y \ f) = f \ (Y \ f)\]

(It’s a fixed-point combinator!)
Three step process for deriving Y

\[(Y \ f) = f \ (Y \ f)\]

\[Y = (\lambda \ (f) \ (f \ (Y \ f)))\]  
1. Treat as definition

\[mY = (\lambda \ (mY) \ \\
    (\lambda \ (f) \ \\
    (f \ ((mY \ mY) \ f))))\]
2. Lift to mk-Y, use self-application

\[mY = (\lambda \ (mY) \ \\
    (\lambda \ (f) \ \\
    (f \ (\lambda \ (x) (((mY \ mY) \ f) \ x))))))\]  
3. Eta-expand
U-combinator: $(U \ U) \text{ is Omega}$

\[
Y = (U \ (\lambda \ (y) \ (\lambda \ (f) \ (f \ (\lambda \ (x) (((y \ y) \ f) \ x))))))
\]
(let ([fact (Y (λ (fact) (λ (n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))))]
  (fact 5)))
Try an example!!!

(define Y (((λ (x) (x x))) (λ (y) (λ (f) (f (λ (x) (((y y) f) x)))))))

(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2))))))

Rewrite this to use the Y combinator instead
\[ e ::= (\text{lambda} (x) \ e) \]
\[
| \ (e \ e) \\
| \ x
\]
De-churching

(define (church->nat cv) )

(define (church->list cv) )

(define (church->bool cv) )

(define (church->bool cv) )
(define (church->nat cv)
   ((cv add1) 0))

(define (church->list cv)
)

(define (church->bool cv)
)

(define (church->list cv)
)

(define (church->bool cv)
)
(define (church->nat cv) 
  ((cv add1) 0))

(define (church->list cv) 
  ((cv (λ (car) (λ (cdr) (cons car (church->list cdr))))))
  (λ (na) '()))

(define (church->bool cv) 
  )
(define (church->nat cv)
  ((cv add1) 0))

(define (church->list cv)
  ((cv (
        λ (car)
          (λ (cdr)
            (cons car
              (church->list cdr))))
       (λ (na) '())))

(define (church->bool cv)
  ((cv (λ () #t))
   (λ () #f)))
(letrec ([map (λ (f lst)
  (if (null? lst)
    '()
    (cons (f (car lst))
      (map f (cdr lst)))]))

(map (λ (x) (+ 1 x))
  '(0 5 3)))
(define lst
  ((((((((((λ (Y-comb)
    (λ (church:null?)
      (λ (church:cons)
        (λ (church:car)
          (λ (church:cdr)
            (λ (church:+)
              (λ (church:*)
                (λ (church:not)
                  ((λ (map)
                    ((map
                      (λ (x)
                        (((church:+ (λ (f) (λ (x) (f x)))
                          x))))
                      ((church:cons (λ (f) (λ (x) x)))
                        ((church:cons
                          (λ (f)
                            (λ (x) (f (f (f (f (f x)))))))
                          ((church:cons
                            (λ (f) (λ (x) (f (f x)))))
                            ((church:cons
                              (λ (when-null)
                                (when-null (λ (x) x))))))))))))
                      (λ (when-null)
                        (when-null (λ (x) x))))))))))))
        (λ (when-null)
          (when-null (λ (x) x)))))
      (Y-comb)
      (λ (map))
      (map
        (λ (x)
          ((church:+ (λ (f) (λ (x) (f x)))
            x))))
      ((church:cons (λ (f) (λ (x) x)))
        ((church:cons
          (λ (f)
            (λ (x) (f (f (f (f (f x)))))))
          ((church:cons
            (λ (f) (λ (x) (f (f x)))))
            ((church:cons
              (λ (when-null)
                (when-null (λ (x) x)))))))))))))
  (map church->nat (church->list lst))
'(1 6 4)
Abstract Machine Zoo

C Term-rewriting Machine
Evaluation contexts

Restrict the order in which we may simplify a program’s redexes

(left-to-right) CBV evaluation

\[ \mathcal{E} ::= (\mathcal{E} \ e) \]
\[ | \ (\nu \ \mathcal{E}) \]
\[ | \ \square \]

(left-to-right) CBN evaluation

\[ \mathcal{E} ::= (\mathcal{E} \ e) \]
\[ | \ \square \]

\[ \nu ::= (\lambda (x) \ e) \]
\[ e ::= (\lambda (x) \ e) \]
\[ | \ (e \ e) \]
\[ | \ x \]
Context and redex

For CBV a redex must be $(\mathcal{E} \quad \mathcal{E})$

For CVN, a redex must be $(\mathcal{E} \quad \mathcal{E})$

\[
\begin{align*}
\mathcal{E} &= (\Box (\lambda (w) w)) \\
\mathcal{E}[((v \quad v))] &= (((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) (\lambda (w) w)) \\
r &= (((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))
\end{align*}
\]
Context and redex

\[ \mathcal{E}[r] = \]
\[ (((\lambda \; x) \; ((\lambda \; (y) \; y) \; x)) \; (\lambda \; (z) \; z)) \; (\lambda \; (w) \; w)) \]

\[ \mathcal{E} = (\square \; (\lambda \; (w) \; w)) \]

\[ r = (((\lambda \; (x) \; ((\lambda \; (y) \; y) \; x)) \; (\lambda \; (z) \; z)) \]
\[ \rightarrow_\beta (((\lambda \; (y) \; y) \; (\lambda \; (z) \; z)) \]
Put the reduced redex back in its evaluation context...

\[ \mathcal{E} = (□ (λ (w) w)) \]

\[ r = (((λ (x)) (((λ (y) y) x)) (λ (z) z))) \rightarrow_β (((λ (y) y)) (λ (z) z)) \]

\[ \mathcal{E}[r] \]

\[ (((λ (y) y)) (λ (z) z)) (λ (w) w)) \]
Exercises—can you evaluate…

1) \(((\lambda (y) y) (\lambda (z) z)) (\lambda (w) w))\)

2) \(((\lambda (u) (u u)) (\lambda (x) (\lambda (x) x)))\)

3) \(((\lambda (x) x) (\lambda (y) y))
\(((\lambda (u) (u u)) (\lambda (z) (z z))))\)
Abstract Machine Zoo

C  Term-rewriting Machine

CC  Context and Redex Machine

CK  Control / Continuation Machine

Next time…