The 90 minute Scheme to C compiler

Marc Feeley

Université de Montréal

Goals

- Goals
  - explain how Scheme can be compiled to C
  - give enough detail to “do it at home”
  - do it in 90 minutes
- Non-goals
  - RnRS compatibility, C interoperability, etc
  - optimizations, performance, etc
  - explain optimizations, Gambit-C, etc
- Target audience
  - people who know Scheme/Lisp
  - helps to know higher-order functions

Why is it difficult?

- Scheme has, and C does not have
  - tail-calls   a.k.a. tail-recursion opt.
  - first-class continuations
  - closures of indefinite extent
  - automatic memory management   i.e. GC
- Implications
  - can’t translate (all) Scheme calls into C calls
  - have to implement continuations
  - have to implement closures
  - have to organize things to allow GC
- The rest is easy!

Tail-calls and GC

- In Scheme, this function runs in constant space, regardless of the value of n (and ignoring the space for the numbers computed)

```
(define f
  (lambda (n x)
    (if (= n 0)
        (car x)
        (f (- n 1)
           (cons (cdr x)
                 (+ (car x)
                    (cdr x)))))))

(f 20 (cons 1 1))  ; => 10946
```

- recursive call is a **tail call** i.e. `f` is a **loop**
- unused pairs are reclaimed by the GC

Closures (1)

- In Scheme functions can be nested and variables are lexically scoped

```
(define add-all
  (lambda (n lst)
    (map (lambda (x) (+ x n)) lst)))

(add-all 1 '(10 20 30))  ; => (11 21 31)

(add-all 5 '(10 20 30))  ; => (15 25 35)
```

- In the body of `(lambda (x) (+ x n))`
  - x is a *bound* occurrence of x
  - n is a *free* occurrence of n
- A variable bound in the closest enclosing `lambda`-expression = a slot of the current activation frame (easy)

Closures (2)

- Closures may also outlive their parent

```
(define make-adder
  (lambda (n)
    (lambda (x) (+ x n))))

(map (make-adder 1)
     '(10 20 30))  ; => (11 21 31)
```

- Traditional (contiguous) stack allocation of activation frames will not work
- A closure must “remember” the parent closure’s activation frame and the GC must reclaim the activation frames only when they are not required anymore

First-class continuations (1)

- First-class continuations allow arbitrary transfer of control
- A **continuation** denotes a suspended computation that is awaiting a value
- For example, when this program is run at the REPL

  ```
  > (sqrt (+ (read) 1))
  ```

  the program will wait at the call to `read` for the user to enter an number.
  The continuation of the call to `read` denotes a computation that takes a value, adds 1 to it, computes its square-root, prints the result and goes to the next REPL interaction.

First-class continuations (2)

- `call/cc` turns the continuation into a function which, when called, causes that suspended computation to resume
- In `(call/cc` *f*`)`, the function *f* will be called with the continuation

```
> (sqrt (+ (call/cc
             (lambda (cont)
               (* 2 (cont 8))))
           1))
3
```

- With first-class continuations it is easy to do: **backtracking**, **coroutining**, **multithreading**, **non-local escapes** (for exception handling)

First-class continuations (3)

- Example 1: non-local escape

```
(define (map-/ lst)
  (call/cc
   (lambda (return)
     (map (lambda (x)
            (if (= x 0)
                (return #f)
                (/ 1 x)))
          lst))))

(map-/ ’(1 2 3))  ; => (1 1/2 1/3)

(map-/ ’(1 0 3))  ; => #f
```

First-class continuations (4)

- Example 2: backtracking
- We want to find X, Y and Z such that $2 <= X, Y, Z <= 9$ and $X^2 = Y^2 + Z^2$

```
(let ((x (in-range 2 9))
      (y (in-range 2 9))
      (z (in-range 2 9)))
  (if (= (* x x)
         (+ (* y y) (* z z)))
      (list x y z)
      (fail)))  ; => (5 3 4)
```

- What is the definition of `in-range` and `fail`?

First-class continuations (5)

```
(define fail
  (lambda () (error "no solution")))

(define in-range
  (lambda (a b)
    (call/cc
     (lambda (cont)
       (enumerate a b cont)))))

(define enumerate
  (lambda (a b cont)
    (if (> a b)
        (fail)
        (let ((save fail))
          (set! fail
            (lambda ()
              (set! fail save)
              (enumerate (+ a 1) b cont)))
          (cont a)))))
```

Approach to compiling Scheme to C

- We use **source-to-source** transformations to do most of the compilation work
- A **source-to-source** transformation is a compiler whose input and output are in the **same language**, in this case Scheme
- The output of the transformations will be “easier to compile” than the input (i.e. there will be less reliance on powerful features)
- The final Scheme code will be straightforward to translate to C
- Two source-to-source transformations: **closure-conversion** and **CPS-conversion**

Scheme subset

- To highlight the difficult aspects of compiling Scheme, only a subset of Scheme is handled by the compiler:
  - Very few primitives (`+`, `-`, `*`, `=`, `<`, `display` (for integers only), and `call/cc`)
  - Only small exact integers and functions (and `#f`=0/`#t`=1)
  - Only the main special forms and no macros
  - `set!` only to global variables
  - No variable-arity functions
  - No error checking

- Exercise: implement the rest of Scheme...

Closure-conversion (1)

- The problem: access to free variables

```
(lambda (x y z)
  (let ((f (lambda (a b)
             (+ (* a x) (* b y)))))
    (- (f 1 2) (f 3 4))))
```

- How are the values of `x` and `y` obtained in the body of `f`?

Closure-conversion (2)

- First idea: pass the values of the free-variables as parameters

```
(lambda (x y z)
  (let ((f (lambda (x y a b)
             (+ (* a x) (* b y)))))
    (- (f x y 1 2) (f x y 3 4))))
```

- This transformation, known as **lambda lifting** works well in this case, but not in general:

```
(lambda (x y z)
  (let ((f (lambda (a b)
             (+ (* a x) (* b y)))))
    f))
```

- The values of the free-variables have to be packaged into an object which also gives the function's code: the **closure**

Closure-conversion (3)

- Second idea: build a structure containing the free-variables and pass it to the function as a parameter when the function is called

```
(lambda (x y z)
  (let ((f (vector
             (lambda (self a b)
               (+ (* a (vector-ref self 1))
                  (* b (vector-ref self 2))))
             x
             y)))
    (- ((vector-ref f 0) f 1 2)
       ((vector-ref f 0) f 3 4))))
```

- Eliminates free-variables
- Each `lambda`-expression now denotes a block of instructions (just like in C)

Closure-conversion rules

- $\boxed{\texttt{(lambda (}P_1 \ldots P_n\texttt{) }E\texttt{)}}$ =
  `(vector (lambda (self` $P_1 \ldots P_n$`)` $\boxed{E}$`)` $\boxed{v}\ldots$`)`
  where $v \ldots$ is the list of free-variables of
  `(lambda (`$P_1 \ldots P_n$`)` E`)`

- $\boxed{v}$ = `(vector-ref self` i`)`
  where v is a free-variable and i is the position of v in the list of free-variables of the enclosing `lambda`-expression

- $\boxed{\texttt{(}f\ E_1 \ldots E_n\texttt{)}}$ = `((vector-ref` $\boxed{f}$ `0)` $\boxed{f}$ $\boxed{E_1}\ldots\boxed{E_n}$`)`
  NOTE: this is valid when f is a variable and this will be the case after CPS-conversion, except when f=`(lambda...)` which is handled specially

- Use `closure` and `closure-ref` for dynamic typing

CPS-conversion (1)

- The problem: continuations have
  - indefinite extent (because of `call/cc`)
  - can be invoked more than once ($X^2 = Y^2 + Z^2$ example)
- Continuations can’t be reclaimed when a function returns
- The GC has to be responsible for reclaiming continuations
- “Simple” solution: transform the program so that continuations are objects explicitly manipulated by the program (closures) and let the GC deal with those

CPS-conversion (2)

- Basic idea of CPS-conversion
  - The evaluation of an expression **produces** a value that is **consumed** by the continuation
  - If we represent the continuation with a function we can use **function call** to express “sending a value to the continuation”

CPS-conversion (3)

- For example in the program

```
(let ((square (lambda (x) (* x x))))
  (write (+ (square 10) 1)))
```

  the continuation of `(square 10)` is a computation that expects a value that it will add one to and then write

- That continuation is represented with the function

```
(lambda (r) (write (+ r 1)))
```

CPS-conversion (4)

- This continuation needs to be passed to `square` so that it can send the result to it (CPS=**Continuation-Passing Style**)
- So we must add a continuation parameter to all `lambda`-expressions, change the function calls to pass the continuation function, and use the continuation when a function needs to return a result

```
(let ((square (lambda (k x) (k (* x x)))))
  (square (lambda (r) (write (+ r 1)))
          10))
```

CPS-conversion (5)

- Notice that tail-calls can be expressed simply by passing the current continuation to the called function

- For example

```
(let ((mult (lambda (a b) (* a b))))
  (let ((square (lambda (x) (mult x x))))
    (write (+ (square 10) 1))))
```

becomes

```
(let ((mult (lambda (k a b) (k (* a b)))))
  (let ((square (lambda (k x) (mult k x x))))
    (square (lambda (r) (write (+ r 1)))
            10)))
```

because the call to `mult` in `square` is a **tail-call**, `mult` has the **same continuation** as `square`

CPS-conversion (6)

- When the CPS-conversion is done systematically on all the program
  - all function calls become tail-calls [a]
  - non-tail-calls create a closure for the continuation of the call
- The function calls can simply be translated to “jumps”

[a] calls to primitive operations like `+` and `vector` are not considered to be function calls

CPS-conversion rules (1)

- We define the notation

$$\frac{\boxed{E}}{\mathcal{C}}$$

  to mean the Scheme expression that is the CPS-conversion of the Scheme expression E where the Scheme expression $\mathcal{C}$ represents E's continuation

- Note that E is a source expression (it may contain non-tail-calls) and $\mathcal{C}$ is an expression in CPS form (it contains tail-calls only)

- $\mathcal{C}$ is either a variable or a `lambda`-expression

CPS-conversion rules (2)

- The first rule is

$$\boxed{\textit{program}} = \texttt{(lambda (}r\texttt{) }\boxed{\textit{program}}\texttt{ (\%halt }r\texttt{))}$$

It says that the **primordial continuation** of the program takes r, the result of the program, and calls the primitive operation `(%halt` r`)` which terminates the execution [a]

[a] in the actual compiler it also displays the result

CPS-conversion rules (3)

- $\underset{\mathcal{C}}{\boxed{c}} = (\mathcal{C}\ \ c)$

- $\underset{\mathcal{C}}{\boxed{v}} = (\mathcal{C}\ \ v)$

- $\underset{\mathcal{C}}{\boxed{\texttt{(set! } v\ E_1\texttt{)}}} = \underset{\texttt{(lambda (}r_1\texttt{) (}\mathcal{C}\ \texttt{(set! } v\ r_1\texttt{)))}}{\boxed{E_1}}$

- $\underset{\mathcal{C}}{\boxed{\texttt{(if } E_1\ E_2\ E_3\texttt{)}}} = \underset{\texttt{(lambda (}r_1\texttt{) (if } r_1\ \underset{\mathcal{C}}{\boxed{E_2}}\ \underset{\mathcal{C}}{\boxed{E_3}}\texttt{))}}{\boxed{E_1}}$

CPS-conversion rules (4)

- $$\underset{\mathcal{C}}{\boxed{\texttt{(begin}\ E_1\ \ E_2\texttt{)}}} = \underset{\texttt{(lambda (}r_1\texttt{)}\ \underset{\mathcal{C}}{\boxed{E_2}}\texttt{)}}{\boxed{E_1}}$$

- $$\underset{\mathcal{C}}{\boxed{\texttt{(+}\ E_1\ \ E_2\texttt{)}}} =$$

  $$\underset{\texttt{(lambda (}r_1\texttt{)}\ \underset{\texttt{(lambda (}r_2\texttt{) (}\mathcal{C}\ \texttt{(+}\ r_1\ \ r_2\texttt{)))}}{\boxed{E_2}}\texttt{)}}{\boxed{E_1}}$$

- $$\underset{\mathcal{C}}{\boxed{\texttt{(lambda (}P_1 \ldots P_n\texttt{)}\ \ E_0\texttt{)}}} =$$

  $$\texttt{(}\mathcal{C}\ \texttt{(lambda (}k\ \ P_1 \ldots P_n\texttt{)}\ \underset{k}{\boxed{E_0}}\texttt{))}$$

CPS-conversion rules (5)

- $$\underset{\mathcal{C}}{\boxed{(E_0)}} = \underset{(\texttt{lambda}\ (r_0)\ (r_0\ \mathcal{C}))}{\boxed{E_0}}$$

- $$\underset{\mathcal{C}}{\boxed{(E_0\ \ E_1)}} = \underset{(\texttt{lambda}\ (r_0)\ \underset{(\texttt{lambda}\ (r_1)\ (r_0\ \mathcal{C}\ r_1))}{\boxed{E_1}})}{\boxed{E_0}}$$

- $$\underset{\mathcal{C}}{\boxed{(E_0\ \ E_1\ \ E_2)}} =$$

  $$\underset{(\texttt{lambda}\ (r_0)\ \underset{(\texttt{lambda}\ (r_1)\ \underset{(\texttt{lambda}\ (r_2)\ (r_0\ \mathcal{C}\ r_1\ r_2))}{\boxed{E_2}})}{\boxed{E_1}})}{\boxed{E_0}}$$

- etc.

CPS-conversion rules (6)

- $$\boxed{\texttt{((lambda ()}\ E_0\texttt{))}}_{\mathcal{C}} = \boxed{E_0}_{\mathcal{C}}$$

- $$\boxed{\texttt{((lambda (}P_1\texttt{)}\ E_0\texttt{)}\ E_1\texttt{)}}_{\mathcal{C}} = \boxed{E_1}_{\texttt{(lambda (}P_1\texttt{)}\boxed{E_0}_{\mathcal{C}}\texttt{)}}$$

- $$\boxed{\texttt{((lambda (}P_1\ P_2\texttt{)}\ E_0\texttt{)}\ E_1\ E_2\texttt{)}}_{\mathcal{C}} = \boxed{E_1}_{\texttt{(lambda (}P_1\texttt{)}\ \boxed{E_2}_{\texttt{(lambda (}P_2\texttt{)}\boxed{E_0}_{\mathcal{C}}\texttt{)}}\ \texttt{)}}$$

- etc.

What about `call/cc`?

- In CPS form, `call/cc` is simply

```
(define call/cc
  (lambda (k f)
    (f k (lambda (dummy-k result)
           (k result)))))
```

- The CPS-converter adds this definition to the CPS-converted program if `call/cc` is used

Compiler structure

- Less than 800 lines of Scheme
- Does
  - Parsing and expansion of forms (e.g. `let`)
  - CPS-conversion
  - Closure-conversion
  - C code generation
- Runtime has
  - One heap section (and currently no GC!)
  - A table of global variables
  - A small stack for parameters, local variables and primitive expression evaluation

Example

```
------------------- SOURCE CODE:

(define square
  (lambda (x)
    (* x x)))

(+ (square 5) 1)

------------------- AST:

(begin
  (set! square (lambda (x.1) (%* x.1 x.1)))
  (%+ (square 5) 1))

------------------- AST AFTER CPS-CONVERSION:

(let ((r.5 (lambda (k.6 x.1)
             (k.6 (%* x.1 x.1)))))
  (let ((r.3 (set! square r.5)))
    (square (lambda (r.4)
              (let ((r.2 (%+ r.4 1)))
                (%halt r.2)))
            5)))
```

Example (cont)

```
------------------ AST AFTER CPS-CONVERSION:

(let ((r.5 (lambda (k.6 x.1)
             (k.6 (%* x.1 x.1)))))
  (let ((r.3 (set! square r.5)))
    (square (lambda (r.4)
              (let ((r.2 (%+ r.4 1)))
                (%halt r.2)))
            5)))

------------------ AST AFTER CLOSURE-CONVERSION:

(lambda ()
  (let ((r.5 (%closure
              (lambda (self.7 k.6 x.1)
                ((%closure-ref k.6 0)
                 k.6
                 (%* x.1 x.1)))))))
    (let ((r.3 (set! square r.5)))
      ((%closure-ref square 0)
       square
       (%closure
        (lambda (self.8 r.4)
          (let ((r.2 (%+ r.4 1)))
            (%halt r.2))))
       5))))
```

Example (cont)

```
------------------ C CODE:

case 0: /* (lambda () (let ((r.5 (%closure (lambda (self.7 k.6 x.1) .

 BEGIN_CLOSURE(1,0); END_CLOSURE(1,0);
 PUSH(LOCAL(0/*r.5*/)); GLOBAL(0/*square*/) = TOS();
 PUSH(GLOBAL(0/*square*/));
 BEGIN_CLOSURE(2,0); END_CLOSURE(2,0);
 PUSH(INT2OBJ(5));
 BEGIN_JUMP(3); PUSH(LOCAL(2)); PUSH(LOCAL(3)); PUSH(LOCAL(4)); END_JU

case 2: /* (lambda (self.8 r.4) (let ((r.2 (%+ r.4 1))) (%halt r.2)))

 PUSH(LOCAL(1/*r.4*/)); PUSH(INT2OBJ(1)); ADD();
 PUSH(LOCAL(2/*r.2*/)); HALT();

case 1: /* (lambda (self.7 k.6 x.1) ((%closure-ref k.6 0) k.6 (%* x..

 PUSH(LOCAL(1/*k.6*/));
 PUSH(LOCAL(2/*x.1*/)); PUSH(LOCAL(2/*x.1*/)); MUL();
 BEGIN_JUMP(2); PUSH(LOCAL(3)); PUSH(LOCAL(4)); END_JUMP(2);
```

Conclusion

- Powerful transformations:
  - **CPS-conversion**
  - **Closure-conversion**

- Performance is not so bad with NO optimizations (about 6 times slower than Gambit-C with full optimization)

- Many improvements are possible...