Scheme Implementation Techniques

Felix.winkelmann@bevuta.com

bevuta IT | GmbH

Scheme

The language:

- A dialect of Lisp
- Till R5RS designed by consensus
- Wide field of experimentation
- Standards intentionally give leeway to implementors

Implemented in everything, embedded everywhere:

- Implementations in assembler, C, JavaScript, Java
- Runs on Desktops, mobiles, mainframes, microcontrollers, FPGAs
- Embedded in games, desktop apps, live-programming tools, audio software, spambots
- Used as shell-replacement, kernel module

Scheme

Advantages:

- Small, simple, elegant, general, minimalistic
- toolset for implementing every known (and unknown) programming paradigm
- provides the basic tools and syntactic abstractions to shape them to your needs
- code is data
- data is code

Scheme

Major implementations (totally subjective):

- Chez (commercial, very good)
- Bigloo (very fast, but restricted)
- Racket (comprehensive, educational)
- MIT Scheme (old)
- Gambit (fast)
- Guile (mature, emphasis on embedding)
- Gauche (designed to be practical)
- CHICKEN (…)

Other implementations:

MIT Scheme
Chibi
Tinyscheme
Miniscm
S9fes
Schemix
PICOBIT
SHard
Dreme
Mosh-scheme
Wraith Scheme
Sizzle
Larceny
Husk Scheme
BiT
Owl Lisp
LispMe
Kawa
Sisc
JScheme
SCSH
Scheme48
Moshimoshi
Stalin
EdScheme
UMB Scheme
Ypsilon Scheme
SigScheme
librep
CPSCM
BiwaScheme
Pixie Scheme
Bee
Armpit Scheme
Elk
Heist
HScheme
Ikarus
IronScheme
Inlab Scheme
Jaja
Pocket Scheme
Vx-Scheme
SIOD
KSI
Bus-Scheme
OakLisp
QScheme
Llava
Luna
PS3I
Scheme->C
QScheme
Psyche
RScheme
Rhizome/Pi
SCM
XLISP
S7
Saggitarius
KSM
BDC Scheme
Ocs
Schemik
...

Scheme

Interesting language features:

- Dynamic/latent typing
- Garbage collected
- Tail calls
- First-class continuations
- Eval

Interpreters

Implementation — Interpreters

Tree Walking:

- SCM, old Guile
- Slow, unless heavily optimized

Implementation — Interpreters

What you want is this:

```
eval[form;a]=[
     null[form]→NIL;
     numberp[form]→form;
     atom[form]→[get[form;APVAL]→car[apval[1]];
                    T→cdr[sassoc[form;a;λ[[ ];error[A8]]]]];
     eq[car[form];QUOTE]→cadr[form];[2]
     eq[car[form];FUNCTION]→list[FUNARG;cadr[form];a];[2]
     eq[car[form];COND]→evcon[cdr[form];a];
     eq[car[form];PROG]→prog[cdr[form];a];[2]
     atom[car[form]]→[get[car[form];EXPR]→apply[expr;[1] evlis[cdr[form];a];a];
                         get[car[form];FEXPR]→apply[fexpr;[1] list[cdr[form];a];a];
                                             ⎧ spread[evlis[cdr[form];a]]; ⎫
                         get[car[form];SUBR]→⎨ $ALIST:=a;                  ⎬ ;
                                             ⎩ TSX subr,[1]4               ⎭
                                              ⎧ AC:=cdr[form];      ⎫
                         get[car[form];FSUBR]→⎨ MQ:=$ALIST:=a;      ⎬ ;
                                              ⎩ TSX fsubr,[1]4      ⎭
                         T→eval[cons[cdr[sassoc[car[form];a;λ[[];error[A9]]]];
                                              cdr[form]];a]];
     T→apply[car[form];evlis[cdr[form];a];a]]
evcon[c;a]=[null[c]→error[A3];
      eval[caar[c];a]→eval[cadar[a];a];
      T→evcon[cdr[c];a]]
evlis[m;a]=maplist[m;λ[[j];eval[car[j];a]]]
```

Implementation — Interpreters

But what you get is often this:

```
static SCM ceval_1(x)
     SCM x;
{
#ifdef GCC_SPARC_BUG
  SCM arg1;
#else
  struct {SCM arg_1;} t;
# define arg1 t.arg_1
#endif
  SCM arg2, arg3, proc;
  int envpp = 0;  /* 1 means an environment has been
pushed in this
               invocation of ceval_1, -1 means pushed
and then popped. */
#ifdef CAUTIOUS
  SCM xorig;
#endif
  CHECK_STACK;
 loop: POLL;
#ifdef CAUTIOUS
  xorig = x;
#endif
#ifdef SCM_PROFILE
  eval_cases[TYP7(x)]++;
#endif
  switch TYP7(x) {
  case tcs_symbols:
    /* only happens when called at top level */
    x = evalatomcar(cons(x, UNDEFINED), !0);
    goto retx;
  case (127 & IM_AND):
    x = CDR(x);
    arg1 = x;
    while(NNULLP(arg1 = CDR(arg1)))
      if (FALSEP(EVALCAR(x))) {x = BOOL_F; goto retx;}
      else x = arg1;
    goto carloop;
 cdrxbegin:
```

```
  case (127 & IM_CASE):
    x = scm_case_selector(x);
    goto begin;
  case (127 & IM_COND):
    while(NIMP(x = CDR(x))) {
      proc = CAR(x);
      arg1 = EVALCAR(proc);
      if (NFALSEP(arg1)) {
      x = CDR(proc);
      if (NULLP(x)) {
        x = arg1;
        goto retx;
      }
      if (IM_ARROW != CAR(x)) goto begin;
      proc = CDR(x);
      proc = EVALCAR(proc);
      ASRTGO(NIMP(proc), badfun);
      goto evap1;
      }
    }
    x = UNSPECIFIED;
    goto retx;
  case (127 & IM_DO):
    ENV_MAY_PUSH(envpp);
    TRACE(x);
    x = CDR(x);
    ecache_evalx(CAR(CDR(x)));      /* inits */
    STATIC_ENV = CAR(x);
    EXTEND_VALENV;
    x = CDR(CDR(x));
    while (proc = CAR(x), FALSEP(EVALCAR(proc))) {
      for (proc = CAR(CDR(x));NIMP(proc);proc = CDR(proc))
{
      arg1 = CAR(proc); /* body */
      SIDEVAL_1(arg1);
      }
      ecache_evalx(CDR(CDR(x))); /* steps */
      scm_env = CDR(scm_env);
      EXTEND_VALENV;
```

Implementation — Interpreters

Bytecode interpretation:

- Used by Guile (and many others)
- Straightforward to implement
- Relatively fast (up to a certain limit)
- Interesting variant: threaded code (used by Petite Chez)

Implementation — Interpreters

“Closure compilation”:

- Translate source-expressions into closure-tree
- Easy and cleanly implemented in Scheme
- Also hits a performance limit (call-intensive)

Implementation — Interpreters

```
(define (compile exp env)
  (define (walk x e)
    (match x
      ((? symbol?)
       (cond ((lexical-lookup x e) =>
               (lambda (index)
                 (lambda (v) (lexical-ref v index))))
             (else
               (let ((cell (lookup x env)))
                 (lambda (v)
                   (if (bound-cell? cell)
                       (cell-value cell)
                       (error "unbound variable" x)))))))
      (('if x y z)
       (let* ((x (walk x e))
              (y (walk y e))
              (z (walk z e)))
         (lambda (v)
           (if (x v)
               (y v)
               (z v)))))
      (('let ((vars vals) ...) body ...)
       (let* ((e2 (make-lexical-env vars))
              (vals (map (lambda (val) (walk val e)) vals))
              (body (walk `(begin ,@body) e2)))
          (lambda (v)
            (body (add-lexical-env vals v)))))
      ...
      ((proc args ...)
       (let* ((proc (walk proc e))
             (args (map (lambda (arg) (walk arg e)) args)))
         (lambda (v)
           (apply (proc v) (map (lambda (arg) (arg v)) args)))))))
  (walk exp '()))
```

Implementation – Compilers

Compilation:

- For (theoretically) maximum speed
- AOT: Generate executable code before running the program
- JIT: Generate code on the fly

Compilers - Targets

Compiling to machine code:

- MIT Scheme, Chez, Larceny, Ikarus
- Potentially very fast
- Very complex
- Performance-characteristics of modern CPU architectures are difficult to predict
- work-intensive
- Needs backend tailored to target platform

Implementation – Compilers - Targets

Using existing backends:

- Implement gcc frontend
- Or use LLVM
- Code-generation libraries (GNU Lightning, libjit, …)

“Tracing” compilers

- Analyze flow during interpretation, record useful information
- Then compile “hot” loops into native code, using the Recorded data
- Used in modern JavaScript engines, LuaJIT
- Highly complex, but sometimes extremely fast

Compiling to C/C++:

- Easy (the basics)
- Take advantage of optimizations done by C compiler (depending on code)
- Tail-calls and continuations are a challenge
- Extra overhead (compile time)
- But can be done interactively (Goo)

Implementation – Compilers - Targets

Compiling to Java (Source or .class files):

- Done in Kawa, Bigloo
- Takes advantage of large runtime
- Provides GC
- Boxing overhead
- Verification may fail when generating bytecode on the fly
- Generating code at runtime will not work on Dalvik
- Startup overhead

Compiling to JavaScript:

- Done in hop (Bigloo-JS backend), Spock
- Embraces the Web
- JavaScript is flexible, engines are getting faster
- JS has become a processor architecture
- Compile to C, then use emscripten …

Compiling to Common Lisp:

- Done in Rabbit, the first Scheme compiler ever
- Big, rich runtime
- Tail calls not guaranteed, continuations not available
- You might as well code in CL (or use Pseudoscheme)

Implementation – Compilers - Targets

Compiling to other high-level languages:

- SML, Ocaml, Haskell, Erlang, …

- Why not?

Implementation – Compilers - Targets

Compiling to hardware:

- SHard (we'll get to this later …)

Compilers –
Issues when compiling to C

Implementation — Compilers — compiling to C

Direct C generation:

- Scheme->C, Bigloo, Stalin
- Must make compromises regarding tail-calls
- No “downward” continuations (or only with high overhead)
- Or requires sophisticated analyses

Strategies for compiling to C:

- Big switch
- Trampolines
- Cheney-on-the-MTA

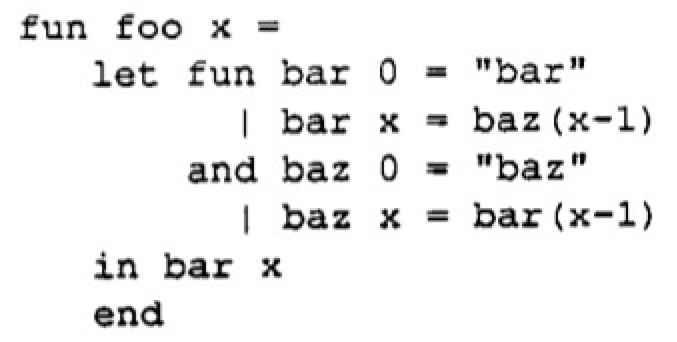
Big switch:

- Generate a big switch statement containing all procedures
- Perform tail calls using goto (or reenter switch contained inside loop)
- GCC extensions (computed goto) can make this very efficient
- Generates large functions, takes time and memory to compile

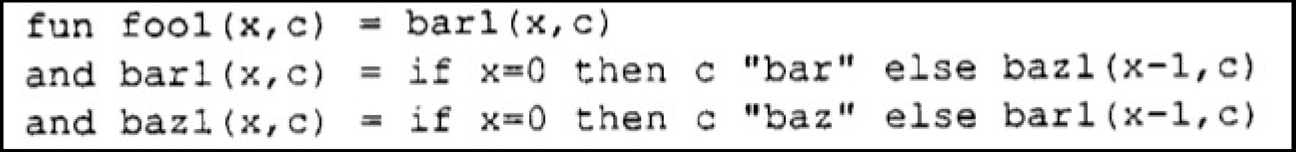
Trampolines:

- Generated functions return functions to be called in tail position
- Outer driver loop

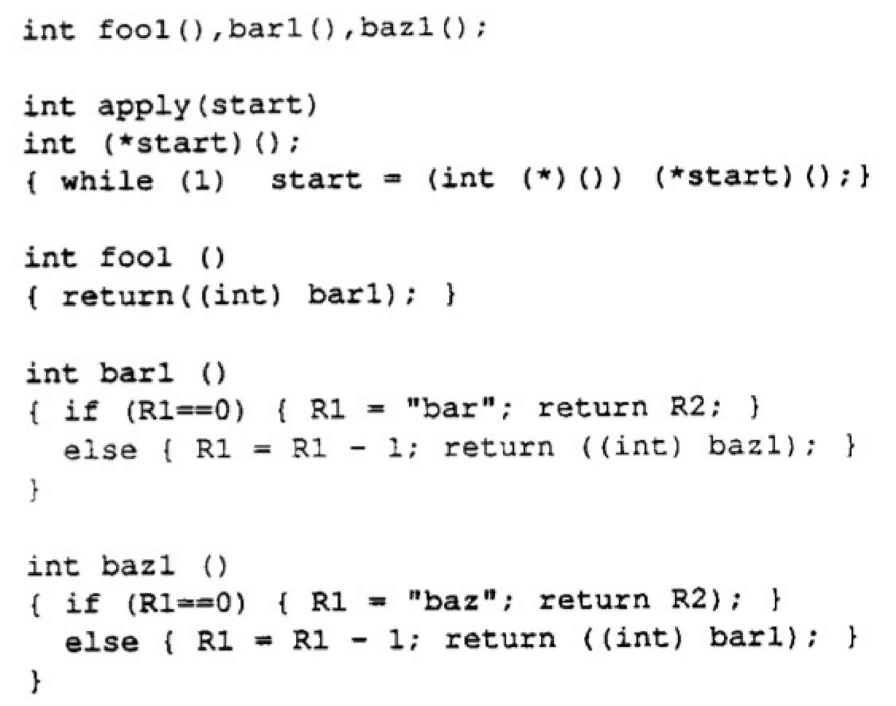
Implementation — Compilers — compiling to C



```
fun foo x =
    let fun bar 0 = "bar"
          | bar x = baz(x-1)
        and baz 0 = "baz"
          | baz x = bar(x-1)
    in bar x
    end
```



```
fun foo1(x,c)  = bar1(x,c)
and bar1(x,c)  = if x=0 then c "bar" else baz1(x-1,c)
and baz1(x,c)  = if x=0 then c "baz" else bar1(x-1,c)
```



```
int foo1(),bar1(),baz1();

int apply(start)
int (*start)();
{ while (1)  start = (int (*)()) (*start)();}

int foo1 ()
{ return((int) bar1); }

int bar1 ()
{ if (R1==0) { R1 = "bar"; return R2; }
  else { R1 = R1 - 1; return ((int) baz1); }
}

int baz1 ()
{ if (R1==0) { R1 = "baz"; return R2); }
  else { R1 = R1 - 1; return ((int) bar1); }
}
```

Implementation — Compilers — compiling to C

Cheney-on-the-MTA:

- Convert to CPS and translate directly to C
- Conses on the stack
- Generated functions never return
- Regularly check stack, GC when stack-limit is hit and perform longjmp(3) (or simply return)

Implementation – Compilers – compiling to C

```
#ifdef stack_grows_upward
#define stack_check(sp) ((sp) >= limit)
#else
#define stack_check(sp) ((sp) <= limit)
#endif
...
object foo(env,cont,a1,a2,a3) environment env; object cont,a1,a2,a3;
{int xyzzy; void *sp = &xyzzy; /* Where are we on the stack? */
 /* May put other local allocations here. */
 ...
 if (stack_check(sp)) /* Check allocation limit. */
  {closure5_type foo_closure; /* Locally allocate closure with 5 slots. */
   /* Initialize foo_closure with env,cont,a1,a2,a3 and ptr to foo code. */
   ...
   return GC(&foo_closure);} /* Do GC and then execute foo_closure. */
 /* Rest of foo code follows. */
 ...
}

object revappend(cont,old,new) object cont,old,new;
{if (old == NIL)
   {clos_type *c = cont;
    /* Call continuation with new as result. */
    return (c->fn)(c->env,new);}
   {cons_type *o = old; cons_type newer; /* Should check stack here. */
    /* Code for (revappend (cdr old) (cons (car old) new)). */
    newer.tag = cons_tag; newer.car = o->car; newer.cdr = new;
    return revappend(cont,o->cdr,&newer);}}
```

http://home.pipeline.com/~hbaker1/CheneyMTA.html

Compilers — Syntax expansion

Syntax expansion:

- Declarative vs. procedural
- Hygiene, referential transparency
- Phase issues (compile-time vs. execution-time)

Implementation – Compilers – Syntax expansion

Defmacro:

- Simple
- Procedural

```
(define-macro (while x . body)
  (let ((loop (gensym)))
    `(let ,loop ()
       (if ,x
           (begin ,@body (,loop))))))
```

Syntax-rules:

- Hygienic, referentially transparent
- Declarative
- Easy to use (for simple things)

```
(define-syntax while
  (syntax-rules ()
    ((_ x body ...)
      (let loop ()
        (if x
            (begin body ... (loop)))))))
```

Implementation — Compilers — Syntax expansion

Explicit renaming:

- Use explicit calls to rename and compare identifiers
- Straightforward but tedious

```
(define-syntax while
  (er-macro-transformer
    (lambda (form rename compare)
      (let ((%if (rename 'if))
            (%let (rename 'let))
            (%begin (rename 'begin))
            (%loop (rename 'loop)))
        `(,%let ,%loop ()
           (,%if ,(cadr form)
                 (,%begin ,@(cddr form) (,%loop))))))))
```

Implicit renaming:

- Similar to explicit-renaming, but assumes renaming is the default mode
- Use explicit calls to "inject" a new identifier
- Invented by Peter Bex, for CHICKEN

```
(define-syntax while
  (ir-macro-transformer
    (lambda (form inject compare)
      `(let loop ()
         (if ,(cadr x)
             (begin ,(cddr x) (loop)))))))
```

Implicit-renaming - Example using injection:

```
(define-syntax while
  (ir-macro-transformer
    (lambda (expr inject compare)
      (let ((test (cadr expr))
            (body (cddr expr))
            (exit (inject 'exit)))
        `(call-with-current-continuation
          (lambda (,exit)
            (let loop
              (if (not ,test) (,exit))
              ,@body
              (loop))))))))
```

Implementation — Compilers — Syntax expansion

Syntactic closures:

- Extends the concept of closing over a lexical environment to syntax

- Conceptually simple

```
(define-syntax loop
  (sc-macro-transformer
    (lambda (form environment)
      `(call-with-current-continuation
         (lambda (exit)
           (let spin ()
             ,@(make-syntactic-closure
                 environment '(exit) (cdr form))
             (spin)))))))
```

Syntax-case:

- Standardized in R6RS
- used in many implementations (Racket, Guile, Chez, Larceny, Ikarus)
- Effectively treats the source code as an abstract data structure

```
(define-syntax while
  (lambda (x)
    (syntax-case x ()
      ((k e ...)
        (with-syntax
          ((exit (datum->syntax-object (syntax k) exit)))
          (syntax (call-with-current-continuation
                    (lambda (exit)
                      (let f () e ... (f))))))))))
```

Portable expanders:

- “alexpander” (syntax-rules + extensions)
- Andre van Tonder's Expander (syntax-case + R6RS module system)
- “psyntax” (syntax-case)

Compilers – Compilation

Intermediate representation:

- Use Scheme
- Straightforward transformations
- Test compilation stages by executing IR directly

Style:

- Direct-style
- CPS (serializes expressions, makes continuations explicit)
- ANF (serializes)

Choices for implementing continuations:

- Take stack-snapshots
- Use makecontext(3), swapcontext(3)
- Stack-reconstruction
- CPS conversion

Stack-reconstruction:

- Maintain a "shadow" activation-frame stack and reconstruct it when the continuation is invoked
- For example used in SCM2JS (targeting JavaScript)

http://florian.loitsch.com/publications

http://cs.brown.edu/~sk/Publications/Papers/Published/pcmkf-cont-from-gen-stack-insp/

Implementation — Compilers — Compilation

```
function sequence(f, g) {
  print('1: ' + f());
  return g();
}
```

```
function sequence(f, g) {
  var tmp1;
  var index = 0;
  var goto = false;
  if (RESTORE.doRestore) {
    var frame = RESTORE.popFrame();
    index = frame.index;
    f = frame.f; g = frame.g;
    tmp1 = frame.tmp1;
    goto = index;
  }
  try {
    switch (goto) {
    case false:
    case 1: goto = false;
      index = 1; tmp1 = f();
      print('1: ' + tmp1);
    case 2: goto = false;
      index = 2; return g();
    }
  } catch(e) {
    if (e instanceof Continuation) {
      var frame = {};
      frame.index = index; // save position
      frame.f = f; frame.g = g;
      frame.tmp1 = tmp1;
      e.pushFrame(frame);
    }
    throw e;
  }
}
```

Implementation — Compilers — Compilation

CPS-conversion:

- Makes continuations explicit
- Trade in procedure-call speed for (nearly) free continuations

Implementation — Compilers — Compilation

```
(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1)))))

(display (fac 10))
(newline)
```

```
(lambda (k1)
  (let ((t7 (lambda (k9 _n_44)
              (zero?
                (lambda (t10)
                  (if t10
                    (k9 '1)
                    (- (lambda (t12)
                         (fac (lambda (t11) (* k9 _n_44 t11)) t12))
                      _n_44
                      '1)))
                _n_44))))
    (let ((t8 (set! fac t7)))
      (let ((t3 '#f))
        (let ((t2 t3))
          (fac (lambda (t6)
                 (display
                   (lambda (t5) (let ((t4 t5)) (newline k1)))
                   t6))
            '10))))))
```

Closure representation:

- Linked environments
- “display” closures
- flat closures

Linked environments:

- Extra-indirection for every reference/update

```
(define (brick-house a b)
  (define (low-rider x y)
    (lambda (q)
      (pick-up-the-pieces b x y q)))
  (low-rider a (thank-you)))
```

```
((brick-house 1 2) 3) -> procedure: [<code>, <e1>]

        <e1> = #(<e2>  <x> <y>)
        <e2> = #(<...> <a> <b>)
```

“Display” closures:

- Add pointers to used environments to a closed-over procedure

```
(define (brick-house a b)
  (define (low-rider x y)
    (lambda (q)
      (pick-up-the-pieces b x y q)))
  (low-rider a (thank-you)))
```

```
((brick-house 1 2) 3) -> procedure: [<code>, <d1>, <d2>]

        <d1> = #(<x> <y>)
        <d2> = #(<a> <b>)
```

Flat closures:

- Add actual values to the closure
- Assigned lexical variables need to be boxed
- Trades memory for access-performance

```
(define (brick-house a b)
  (define (low-rider x y)
    (lambda (q)
      (pick-up-the-pieces b x y q)))
  (low-rider a (thank-you)))
```

```
((brick-house 1 2) 3) ->
  procedure: [<code>, <b>, <x>, <y>]
```

Closure conversion:

- Convert “lambda” forms into explicit closure construction

Implementation — Compilers — Compilation

```
($closure () (k1)
  (let ((t7 ($closure () (k9 _n_44)
              (zero?
               ($closure (($local-ref _n_44) ($local-ref k9)) (t10)
                 (if ($local-ref t10)
                     (($closure-ref 1) '1)
                     (-
                      ($closure (($closure-ref 0) ($closure-ref 1)) (t12)
                        (fac
                         ($closure (($closure-ref 0) ($closure-ref 1)) (t11)
                           (*
                            ($closure-ref 1)
                            ($closure-ref 0)
                            ($local-ref t11)))
                         ($local-ref t12)))
                      ($closure-ref 0)
                      '1)))
               ($local-ref _n_44)))))
     (let ((t8 (set! fac ($local-ref t7))))
       (let ((t3 '#f))
         (let ((t2 ($local-ref t3)))
           (fac
            ($closure (($local-ref k1)) (t6)
              (display
               ($closure (($closure-ref 0)) (t5)
                 (let ((t4 ($local-ref t5)))
                   (newline ($closure-ref 0))))
               ($local-ref t6)))
            '10))))))
```

Safe-for-space-complexity:

- Term coined by Andrew Appel
- CPS + flat closures guarantees minimal data retention

```
(define (flashlight data)
  (superfly data)
  (amen-brother))
```

```
($closure () (k1)
   (let ((t2 ($closure () (k4 _data_44)
               (superfly
                ($closure (($local-ref k4)) (t6)
                  (let ((t5 ($local-ref t6)))
                    (amen-brother
                     ($closure-ref 0))))
                ($local-ref _data_44))))))
      (let ((t3 (set! flashlight ($local-ref t2))))
        (($local-ref k1) '#f))))
```

Assignment elimination:

- Required when using flat closures

```
(define (get-down-on-it x)
  (define (out-of-sight) (+ x y))
  (set! x (* x 2))
  out-of-sight)
```

```
(define get-down-on-it
  ($closure () (_x_44)
    (let ((_x_44 ($box _x_44)))
      (letrec ((out-of-sight
                 ($closure (($local-ref _x_44)) (_y_46)
                   (+
                    ($unbox ($closure-ref 0))
                    ($local-ref _y_46)))))
         ($box-set!
           ($local-ref _x_44)
           (* ($unbox ($local-ref _x_44)) '2))
         ($local-ref out-of-sight)))))
```

Compilers - Optimizations

Implementation — Compilers — Optimizations

Inlining:

- The most important optimization
- Reduce procedure-call overhead
- Reduce overhead of intrinsic operations

Primitive procedures:

- Every global variable may be redefined at any time, even primitives
- Unless this is solved, all optimizations are moot
- Use flow-analysis or module-systems (or cheat)
- (eval (read)) breaks everything

Implementation — Compilers — Optimizations

The usual optimizations:

- CSE
- Constant-folding
- Variable/value-propagation

Lambda-lifting:

- Lift local procedures to toplevel adding free variables as extra arguments
- Used in Larceny (incremental lambda-lifting) and Gambit

Implementation — Compilers — Optimizations

```
((lambda ()
    (begin
     (set! reverse-map
           (lambda (.f_2 .l_2)
             (define .loop_3
               (lambda (.l_5 .x_5)
                 (if (pair? .l_5)
                     (.loop_3 (cdr .l_5)
                              (cons (.f_2 (car .l_5)) .x_5))
                     .x_5)))
             (.loop_3 .l_2 '())))
     'reverse-map)))
```

```
((lambda ()
    (define .loop_3
      (lambda (.f_2 .l_5 .x_5)
        (if (pair? .l_5)
            (.loop_3 .f_2
                     (cdr .l_5)
                     (cons (.f_2 (car .l_5)) .x_5))
            .x_5)))
    (begin
     (set! reverse-map
           (lambda (.f_2 .l_2)
             (.loop_3 .f_2 .l_2 '())))
     'reverse-map)))
```

Implementation — Compilers — Optimizations

Type analysis:

- Hindley-Milner type-inference (variables have one type)
- Flow-Analysis (more powerful, but more complex, and only Complete when doing whole-program compilation)

Compilers using type-analysis:

- schlep (declare types or associates variable names with certain types)
- PreScheme (H&M)
- Softscheme (frontend)
- Bigloo, Stalin, CHICKEN and probably many others

Implementation — Compilers — Optimizations

- An example (from CHICKENs "scrutinizer")

```
(define (think-about-it x y)
  (let ((z (+ x 1))                                                       ; z: number, x: number
        (u (cons x y))                                                      ; u: (pair number *)
        (q (if (vector? y)
               (begin                                                                ; y: vector
                 (shake-everything-you-got u)                              ; now u is (pair * *)
                 (set! z
                   (string-append
                     (vector-ref y (car u))
                     ": it's a new day"))                                            ; z: string
                 (string-ref z x))                                        ; z: string, x: number
               (error "St. Louis breakdown"))))           ; q: (still) string, y: (still) vector
    (tighten-it-up
      (string-append (vector-ref y x) z q))))                  ; y: vector, z: string, q: string
```

Stalin:

- "Stalin brutally optimizes"
- Probably the smartest compiler
- R4RS, with caveats
- Very long compile-times
- Needs a lot of experience to use well
- Not actively developed
- Sometimes gives rather useless error messages

```
(define (fuck-up) (panic "This shouldn't happen"))
```

Unboxing of floating-point numbers:

- Required for number crunching
- Done by a Gambit, Bigloo, Racket

Implementation — Compilers — Optimizations

```
(let ((Temp_0 (fl- (fl* W_0 a_J4)
                   (fl* W_1 a_J5)))
      (Temp_1 (fl+ (fl* W_0 a_J5)
                   (fl* W_1 a_J4)))
      (Temp_2 (fl- (fl* W_0 a_J6)
                   (fl* W_1 a_J7)))
      (Temp_3 (fl+ (fl* W_0 a_J7)
                   (fl* W_1 a_J6))))
  (let ((a_J0 (fl+ a_J0 Temp_0))
        (a_J1 (fl+ a_J1 Temp_1))
        (a_J2 (fl+ a_J2 Temp_2))
        (a_J3 (fl+ a_J3 Temp_3))
        (a_J4 (fl- a_J0 Temp_0))
        (a_J5 (fl- a_J1 Temp_1))
        (a_J6 (fl- a_J2 Temp_2))
        (a_J7 (fl- a_J3 Temp_3)))
    (let ((W_0 W_2)
          (W_1 W_3)
          (W_2 (fl- 0. W_3))
          (W_3 W_2))
      ...
```

```
___F64V13=((___F64V11)*(___F64V2));
___F64V14=((___F64V12)*(___F64V1));
___F64V15=((___F64V14)+(___F64V13));
___F64V16=((___F64V11)*(___F64V1));
___F64V17=((___F64V12)*(___F64V2));
___F64V18=((___F64V17)-(___F64V16));
___F64V19=((___F64V11)*(___F64V4));
___F64V20=((___F64V12)*(___F64V3));
___F64V21=((___F64V20)+(___F64V19));
___F64V22=((___F64V11)*(___F64V3));
___F64V23=((___F64V12)*(___F64V4));
___F64V24=((___F64V23)-(___F64V22));
___F64V25=((___F64V5)-(___F64V15));
___F64V26=((___F64V6)-(___F64V18));
___F64V27=((___F64V7)-(___F64V21));
___F64V28=((___F64V8)-(___F64V24));
___F64V29=((___F64V5)+(___F64V15));
___F64V30=((___F64V6)+(___F64V18));
___F64V31=((___F64V7)+(___F64V21));
___F64V32=((___F64V8)+(___F64V24));
___F64V33=((0.)-(___F64V9));
```

Implementation — Compilers — Optimizations

Speculative inlining:

- Check procedures and arguments at runtime

```
(f (car (g 5)))  -->  (f (let ((x (g 5)))
                           (if (and (’##eq? car ’car)
                                    (’##pair? x))
                               (’##car x)
                               (car x))))
```

Runtime — Garbage collection

Implementation — Runtime - Garbage collection

Garbage collection:

- Conservative
- Reference counting
- Mark & Sweep
- Stop & Copy
- Generation collectors

Implementation – Runtime - Garbage collection

Conservative GC:

- Scan registers, stack, heap-memory for pointers
- Simple, straightforward to use
- libgc (BDW)
- Works surprisingly well (but not always)
- No extra work, may increase performance
- Simplifies embedding and FFI considerably

Implementation – Runtime - Garbage collection

Reference counting GC:

- Maintain count of references
- Simple (at first glance, but gets complicated quickly)
- Doesn't handle cycles

Mark & sweep GC:

- Mark live data, then scan heap and collect unmarked items
- Simple
- Speed depends on size of heap
- May need extra compaction logic

Stop & copy GC:

- Trades in memory for speed
- Uses two heaps, copying from one to the other, then swaps
- Automatic compaction
- Non-recursive using Cheney algorithm (also breadth-first)
- Speed depends on amount of live data

Generational GC:

- Use multiple heap-generations, collected independently
- Usually variations of S&C
- Currently the state of the art

Runtime – Data representation

Data representation:

- Tagging (encode type-information in value)
- String-representation
- Heap organization

Tagging of immediate (or non-immediate) values:

- Small integers, booleans, characters
- Need to be distinguished from data block pointers

Tag bits:

- Use low-bit(s) to mark immediates (pointers are usually even)

  XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX1

- Store additional type-information in non-immediate Object header

- Endless variations possible

Preallocated (boxed) fixnums:

- Used in PDP10 MacLisp
- Fixnum objects in low heap (for a limited range)
- If done right, arithmetic can be performed directly on pointers

Strings (unicode):

- With ASCII everything was easy
- UCS-2/4: needs more memory, but has O(1) access
- UTF-8: slow access, but saves space, simplifies access To foreign code
- Or use hybrid approach (as used in Larceny): 8-bit String + lookup-table for non-Latin1 chars

https://trac.ccs.neu.edu/trac/larceny/wiki/StringRepresentations

Implementation – Runtime – Data representation

BIBOP:

- BIg Bag Of Pages
- Use different heap-areas for differently typed data
- Calculate type from address (reduces need for object header)

Foreign function interfaces

Interfacing to foreign code:

- dynamic: generate glue-code on the fly (usually done when generating machine code or in interpreters)

- static: generate glue-code during compilation (i.e. when you compile to C)

- Some libraries do the glue-code generation for you

  libffi
  even better: dyncall

http://dyncall.org

Implementation – Foreign-function interfaces

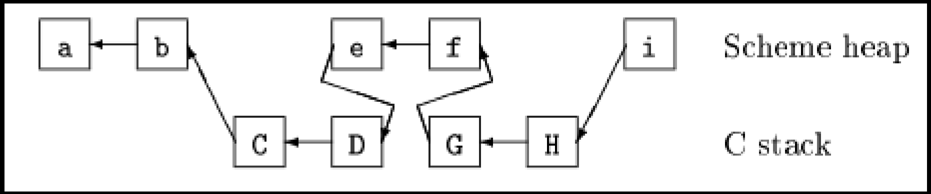
Preprocessor issues:

- Data-sizes are hidden in macro- and struct-definitions
- Solution: run C-compiler on the fly to extract information
- Used in Larceny

Implementation — Foreign-function interfaces

Continuation issues:

- Continuations are not preserved in foreign code
- Use threads?



Other interesting things

Other interesting things

Compile to Lua VM:

- Fast and small VM
- Supports tail-calls
- No continuations but co-routines
- VM not officialy documented
- "A No-Frills Introduction to Lua 5.1 VM Instructions"

http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf

Other interesting things

Implement Scheme in Scheme:

- Denotational semantics by Anton van Straaten
- GRASS

http://www.appsolutions.com/SchemeDS/ds.html

http://www.call-with-current-continuation.org/grass.tar.gz

Other interesting things

Schemix

- Scheme as a kernel module

```
$ echo "(display (+ 1 2 3))" > /dev/schemix
$ cat /dev/schemix
6
$ cat > /dev/schemix
(define foo (kernel-lambda (char*) printk))
(foo "Blah, blah, blah")
^D
$ dmesg | tail -n 1
Blah, blah, blah
```

http://www.abstractnonsense.com/schemix/

Other interesting things

PICOBIT:

- Targets microcontrollers

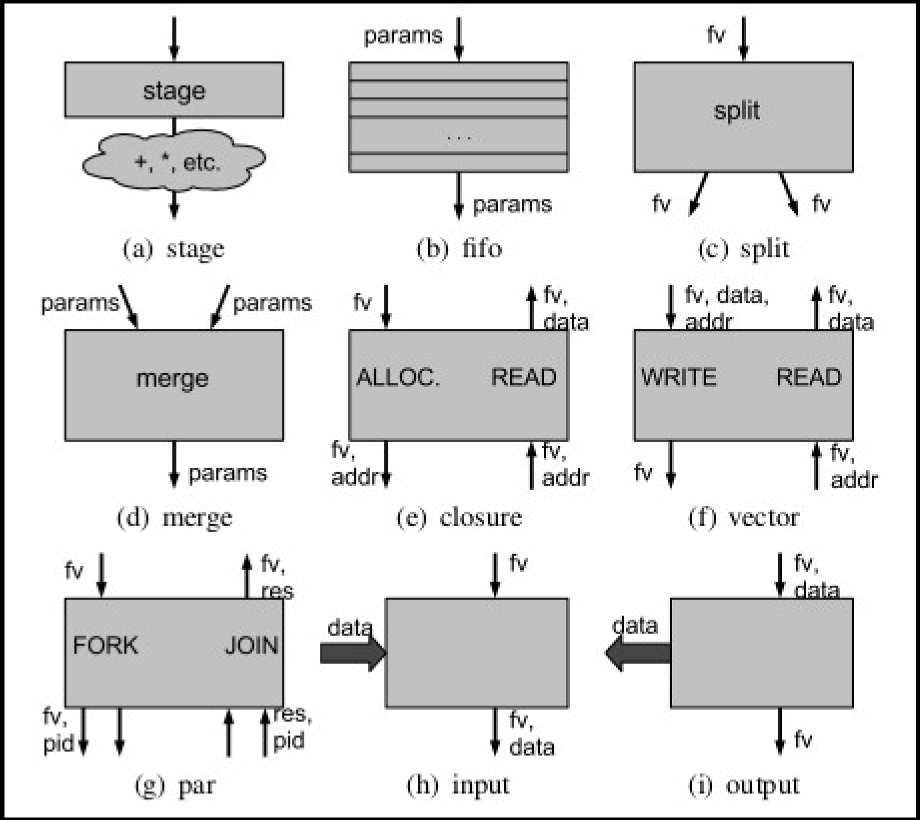
http://www.iro.umontreal.ca/~feeley/papers/sw03.pdf

Other interesting things

Compile to FPGA:

- SHard (University Montreal)
- Very restricted Scheme subset

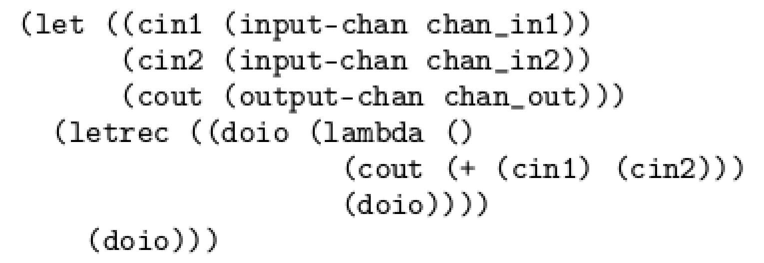
Other interesting things



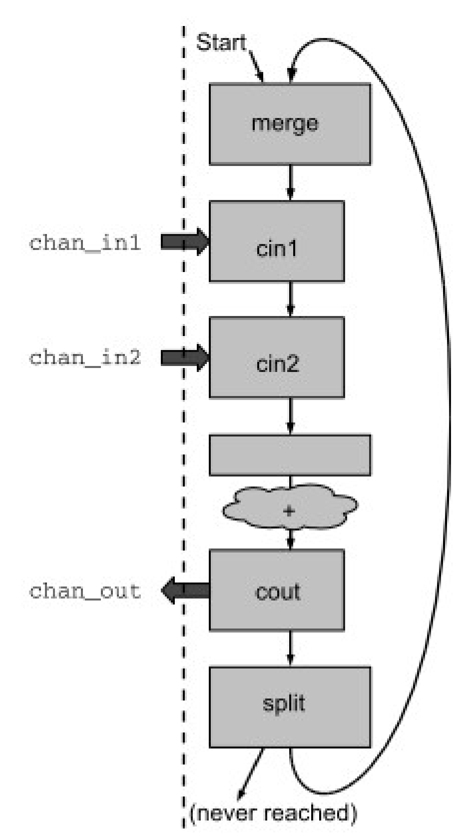
(a) stage (b) fifo (c) split

(d) merge (e) closure (f) vector

(g) par (h) input (i) output



```
(let ((cin1 (input-chan chan_in1))
      (cin2 (input-chan chan_in2))
      (cout (output-chan chan_out)))
  (letrec ((doio (lambda ()
                   (cout (+ (cin1) (cin2)))
                   (doio))))
    (doio)))
```



What I have not covered:

- Embedding Scheme into other applications
- Runtime- and library-design
- Bootstrapping
- Countless other things …

So:

- Implement Scheme!
- It's the true way of learning the language (and any language)
- Experiment, and don't worry about standard conformance

Required reading:

- The Multics MacLisp compiler
- The acknowledgements section of the Scheme-Shell manual
- The “Lambda” papers

http://www.multicians.org/lcp.html

http://www.scsh.net/docu/html/man.html

http://library.readscheme.org/

Links:

http://library.readscheme.org

http://www.schemers.org

http://www.scheme-reports.org

http://wiki.call-cc.org

Books:

- "Compiling with Continuations"
- "Lisp In Small Pieces"
- "Essentials of Programming Languages"
- “Garbage Collection”

Thank you.