Good news, everybody!

Understanding Guile 2.2 Performance
FOSDEM 2016
Andy Wingo
wingo@{igalia,pobox}.com
https://wingolog.org
Good news, everybody!

Guile is faster! Woo hoo!
Bad news, everybody!

“The expert Lisp programmer eventually develops a good ‘efficiency model’.”—Peter Norvig, PAIP

Your efficiency model is out of date
Recap: 1.8

Simple

Approximate efficiency model:

- Cost $O(n)$ in number of reductions

Example:

- Which is slower, $ (+ 1 (+ 2 3)) (+ 1 5)$?

No compilation, so macros cost at run-time
Recap: 2.0

Compilation: macro use has no run-time cost

Partial evaluation (peval)

Cost of \((+ 1 (+ 2 3))\) and 6 are same

Some contification

More on this later

No allocation when building environments
Recap: 2.0

Cost $O(n)$ in number of instructions

- But instructions do not map nicely to Scheme

Inspect the $n$ in $O(n)$:

- optimize (effect of peval)
- disassemble (instructions, effect of contification)
A note on peval

> ,optimize (let lp ((n 5))
>   (if (zero? n)
>       n
>       (+ n (lp (1- n)))))

$6 = 15$

Function inlining, loop unrolling (recursive or iterative), constant folding, constant propagation, beta reduction, strength reduction

Essentially lexical in nature

Understanding peval is another talk
Guile 2.2

Many improvements of degree
Some improvements of kind
Understanding needed to re-develop efficiency model
Improvements of kind

A lambda is not always a closure
Names don’t keep data alive
Unlimited recursion
Dramatically better loops
Lower footprint
Unboxed arithmetic
A lambda is not always a closure

A lambda expression defines a function
That function may or may not exist at run-time
A lambda is not always a closure

Gone
Inlined
Contified
Code pointer
Closure
Lambda: Gone

peval can kill unreachable lambdas

> ,opt (let ((f (lambda ()
  (launch-the-missiles!)))))
  42)
42

> ,opt (let ((launch? #f)
  (f (lambda ()
    (launch-the-missiles!)))))
  (if launch? (f) 'just-kidding))
just-kidding
Lambda: Inlined

peval can inline small or called-once lambdas

> ,opt (let ((launch? #t)
    (f (lambda ()
        (launch-the-missiles!))))
    (if launch? (f) 'just-kidding))
(launch-the-missiles!)
Lambda: Contified

Many of Guile 2.2’s optimizations can’t be represented in Scheme

> (define (count-down n)
  (define loop
    (lambda (n out)
      (let ((out (cons n out)))
        (if (zero? n)
            out
            (loop (1- n) out))))
    (loop n '(())))
> ,x count-down
Disassembly of #<procedure count-down (n)> at

[...]
L1:
10  (cons 2 1 2)
11  (br-if-u64-=-scm 0 1 #f 5) ;; -> L2
14  (sub/immediate 1 1 1)
15  (br -5)                    ;; -> L1
L2:
[...]

loop function was contified: incorporated into body of count-down
Lambda: Contified

Inline : Copy :: Contify : Rewire

Contification always an optimization

Never causes code growth

Enables other optimizations

Can contify a set of functions if

All callers visible to compiler

Always called with same continuation

Reliable: Expect this optimization
(define (thing)
  (define (log what)
    (format #t "Very important log message: ~a\n" what)
    ;; If `log' is too short, it will be inlined. Make it bigger.
    (format #t "Did I ever tell you about my chickens\n")
    (format #t "I was going to name one Donkey\n")
    (format #t "I always wanted a donkey\n")
    (format #t "In the end we called her Raveonette\n")
    (format #t "Donkey is not a great name for a chicken\n")
    (newline) (newline) (newline) (newline) (newline))
  (log "ohai")
  (log "kittens")
  (log "donkeys"))
Lambda: Code pointer

,x thing
Disassembly of #<procedure thing ()> at #x97d704:

[...]

Disassembly of log at #x97d754:

[...]

Two functions, we prevented inlining, whew
Lambda: Code pointer

Disassembly of #<procedure thing ()> at #x97d704:

[...]

12    (call-label 3 2 8)              ;; log at #x97d754

Call procedure at known offset (+8 in this case)

Cheaper call

Precondition: All callers known
Lambda: Code pointer

Disassembly of #<procedure thing ()> at #x97d704:

[...]
12   (call-label 3 2 8)       ;; log at #x97d754

No need for procedure-as-value
Guile currently has a uniform calling convention

❖ Callee-as-a-value passed as arg 0
❖ Arg 0 provides access to free vars, if any
Lambda: Code pointer

If you don’t need the code pointer...

No free vars? Pass any value as arg 0
1 free var? Pass free variable as arg 0
2 free vars? Free vars in pair, pass that pair as arg 0
3 or more? Free vars in vector

Mutually recursive set of procedures?
One free var representation for union of free variables of all functions
Lambda: Closure

Not all callees known? Closure

Closure: an object containing a code pointer and free vars

Though...

0 free variables? Use statically allocated closure

Entry point of mutually recursive set of functions, and all other functions are well-known? Share closure
Lambda: It’s complicated
Names don’t keep data alive

2.0: Named variables kept alive

In particular, procedure arguments and the closure

(define (foo x)
  ;; Should I try to "free" x here?
  ;; (set! x #f)
  (deep-recursive-call)
  #f)

(foo (compute-big-vector))
Names don’t keep data alive

2.2: Only live data is live

User-visible change: less retention...

...though, backtraces sometimes missing arguments

Be (space-)safe out there
Unlimited recursion

Guile 2.0: Default stack size 64K values

Could raise or lower with

GUILE_STACK_SIZE

Little buffer at end for handling errors, but quite flaky
Unlimited recursion

Guile 2.2: Stack starts at one page
Stack grows as needed
When stack shrinks, excess pages returned to OS, at GC
See manual
Recurse away :)

Dramatically better loops

Compiler in Guile 2.2 can reason about loops

Contification produces loops

Improvements of degree: CSE, DCE, etc over loops

Improvements of kind: hoisting
Dramatically better loops

One entry? Hoist effect-free or always-reachable expressions (LICM)

One entry and one exit? Hoisting of all idempotent expressions (peeling)

(define (vector-fill! v x)
  (let lp ((n 0))
    (when (< n (vector-length v))
      (vector-set! v n x)
      (lp (1+ n))))))

Disassembly needed to see.
Footprint

Guile 2.0
☞ 3.38 MiB overhead per process
☞ 13.5 ms startup time
(Overhead: Dirty memory, 64 bit)

Guile 2.2
☞ 2.04 MiB overhead per process
☞ 7.5 ms startup time

ELF shareable static data allocation
Lazy stack growth (per-thread win too!)
Unboxed arithmetic

Guile 2.0

- All floating-point numbers are heap-allocated

Guile 2.2

- Sometimes we can use raw floating-point arithmetic
- Sometimes 64-bit integers are unboxed too
Unboxed arithmetic

> (define (f32vector-double! v)
  (let lp ((i 0))
    (when (< i (bytevector-length v))
      (let ((f32 (bytevector-ieee-single-native-ref v i)))
        (bytevector-ieee-single-native-set! v i (* f32 2))
        (lp (+ i 4)))))
Unboxed arithmetic

> (define v (make-f32vector #e1e6 1.0))
> ,time (f32vector-double! v)

Guile 2.0: 152ms, 71ms in GC
Guile 2.2: 15.2ms, 0ms in GC
10X improvement!
> ,x f32vector-double!

... L1:  
  18  (bv-f32-ref 0 3 1)
  19  (fadd 0 0 0)
  20  (bv-f32-set! 3 1 0)
  21  (uadd/immediate 1 1 4)
  22  (br-if-u64-< 1 4 #f -4) ;; -> L1

Index and f32 values unboxed
Length computation hoisted
Strength reduction on the double
Loop inverted
Unboxed arithmetic

Details gnarly.

Why not:

> (define (f32vector-map! v f)
  (let lp ((i 0))
    (when (< i (f32vector-length v))
      (let ((f32 (f32vector-ref v i)))
        (f32vector-set! v i (f f32))
        (lp (+ i 1))))))
Unboxed arithmetic

(when (< i (f32vector-length v)))
  (let (((f32 (f32vector-ref v i)))
    (f32vector-set! v i (f f32))
    (lp (+ i 1))))

Current compiler limitation: doesn’t understand f32vector-length, which asserts bytevector length divisible by 4

Compiler can’t see through (f f32): has to box f32

... unless f is inlined
Unboxed arithmetic

In practice: 10X speedups, if your efficiency model is accurate

Odd consequence: type checks are back

(unless (= x (logand x #xfffffffff))
  (error "not a uint32"))

Allows Guile to unbox x as integer

Useful on function arguments
Unboxed arithmetic

-- LuaJIT
local uint32 = ffi.new('uint32[1]')
local function to_uint32(x)
    uint32[0] = x
    return uint32[0]
end

For floats, use f64vectors :(
Summary

Guile 2.0: Cost is $O(n)$ in number of instructions

Guile 2.2: Same, but to understand performance

- Mapping from Scheme to instructions more complex
- disassemble necessary to verify

- Pay more attention to allocation

Or just come along for the ride and enjoy the speedups :)}