The Development of

# Chez Scheme

R. Kent Dybvig

ICFP ’06

September 18, 2006

# This Talk

Brief description of Chez Scheme

Contributors

Precursor systems

Early development

Growth of the implementation

Cool hacks

Lessons learned

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

. . . with a few extensions

# Chez Scheme

records, modules, syntax-case, scripts, saved heaps, edge-count profiling, guardians, formatted output, weak pairs, one-shot continuations, pretty printer, generic ports, cafés and waiters, symbol property lists, eval-when, trace package, fasl files, engines, inspector, boot files, case-lambda, incremental compiler, interpreter, compiled files, compiled scripts, generational gc, keyboard interrupt handling, timer interrupts, collect request handlers, identifier aliases, record read syntax, record writers, parameters, fluid variable binding, fluid syntax binding, quasisyntax, meta definitions, include, timing and statistics, parallel processing with threads, synchronization via mutexes and conditions, thread parameters, apropos, additional arithmetic operations, additional input/output operations, input-output ports, compressed input/output, additional list operations, boxes, fixnum arithmetic, flonum arithmetic, inexactnum/flonum arithmetic, breakpoints, file-system operations, user-defined environments, gensyms with globally unique names, expansion-passing style macros, source optimizer, expand/optimize, two's complement logical and shift operations, eq hash tables, environmental queries, sort and merge, random exact integer and inexact number generation, graph printing, visit/revisit, top-level value operators, string ports, void object, source-object correlation, letrec violation detection, Scheme⇔C calls, dynamic loading of C object code, subprocess creation and communication

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

. . . with more than a few extensions

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

. . . with more than a few extensions

Designed for reliability and performance

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

. . . with more than a few extensions

Designed for reliability and performance

Multiple architectures and operating systems

# Chez Scheme

Implementation of Revised$^5$ Report Scheme

. . . with more than a few extensions

Designed for reliability and performance

Multiple architectures and operating systems

Threading/multiple processor support

# System Highlights

Incremental optimizing native-code compiler

Generational garbage collector

Syntax-case macros

Modules (top-level and local)

Scheme⇔C interface

Application delivery and scripts

# Compiler Highlights

Source optimizer/inliner

Closure and assignment conversion

Local call/closure optimization

Destination-driven code generation

Local and global register allocation

Assemblers and linker

Interactive “on the fly” compilation

# Current Applications

Chip layout and testing

Robotic drug testing

Behavioral simulation

Enterprise integration/management

Multi-threaded web services

Computer-assisted art

Research and education

# Developers

Bruce Smith

Bob Hieb

Carl Bruggeman

Mike Ashley

Bob Burger

Oscar Waddell

# Other Major Contributors

George Davidson / Sandia National Labs

Sam Daniel / Motorola

Al Reich, Robert Boone / Freescale

Bob Burger, Mike Ashley / Beckman Coulter

Scott Watson, Thant Tessman / Disney

IU faculty, staff, students

Other colleagues

Many, many others

# Precursor Systems

SDP (with Rex Dwyer)

Z80 Scheme (with George Cohn)

C-Scheme

DG Common Lisp (with Rob Vollum, others)

# Early Development

Motivation: support PhD research

Priorities: speed and reliability

# Design Decisions

Traditional Scheme implementation?

- heap-allocated call frames
- linked environments
- fast closure-creation, continuations
- slow procedure calls, variable references

# Design Decisions

Traditional Algol/C implementation?

- stack-allocated call frames
- variables in frames or registers
- fast procedure calls, variable references
- closures, continuations problematic

# Design Decisions

Abandon traditional Scheme implementation

Adapt traditional Algol/C implementation

$\Rightarrow$ Lesser used operations must pay own way

# Stack-Allocated Call Frames

Problem 1: first-class continuations

- stack may be overwritten

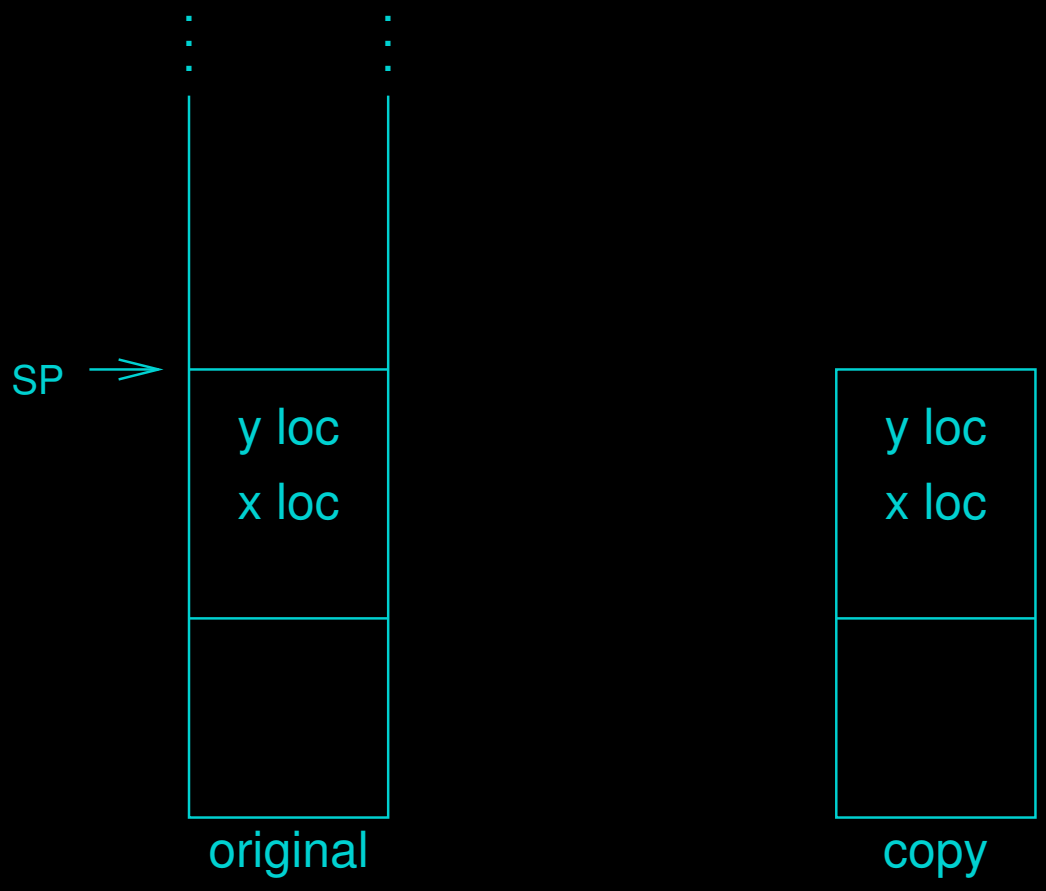
Obvious solution:

- call/cc copies the stack
- throw reinstates the stack
- . . . call/cc “pays its own way”

# Stack-Allocated Call Frames

Problem 2: variables with indefinite extent

- stack copy may replicate locations
- . . . how to track/update copies?



# Display (Flat) Closures

Inspired by Algol 60 *displays* (Randell and Russell):

- bank of pointers replacing static chain

Also by observation:

- most $\lambda$-expressions have few free variables
  (with global variables in static locations)

# Display (Flat) Closures

Closure is like a vector


First element is code pointer

Remaining elements are free variable values

- copied from registers, stack frame
- copied from other closures

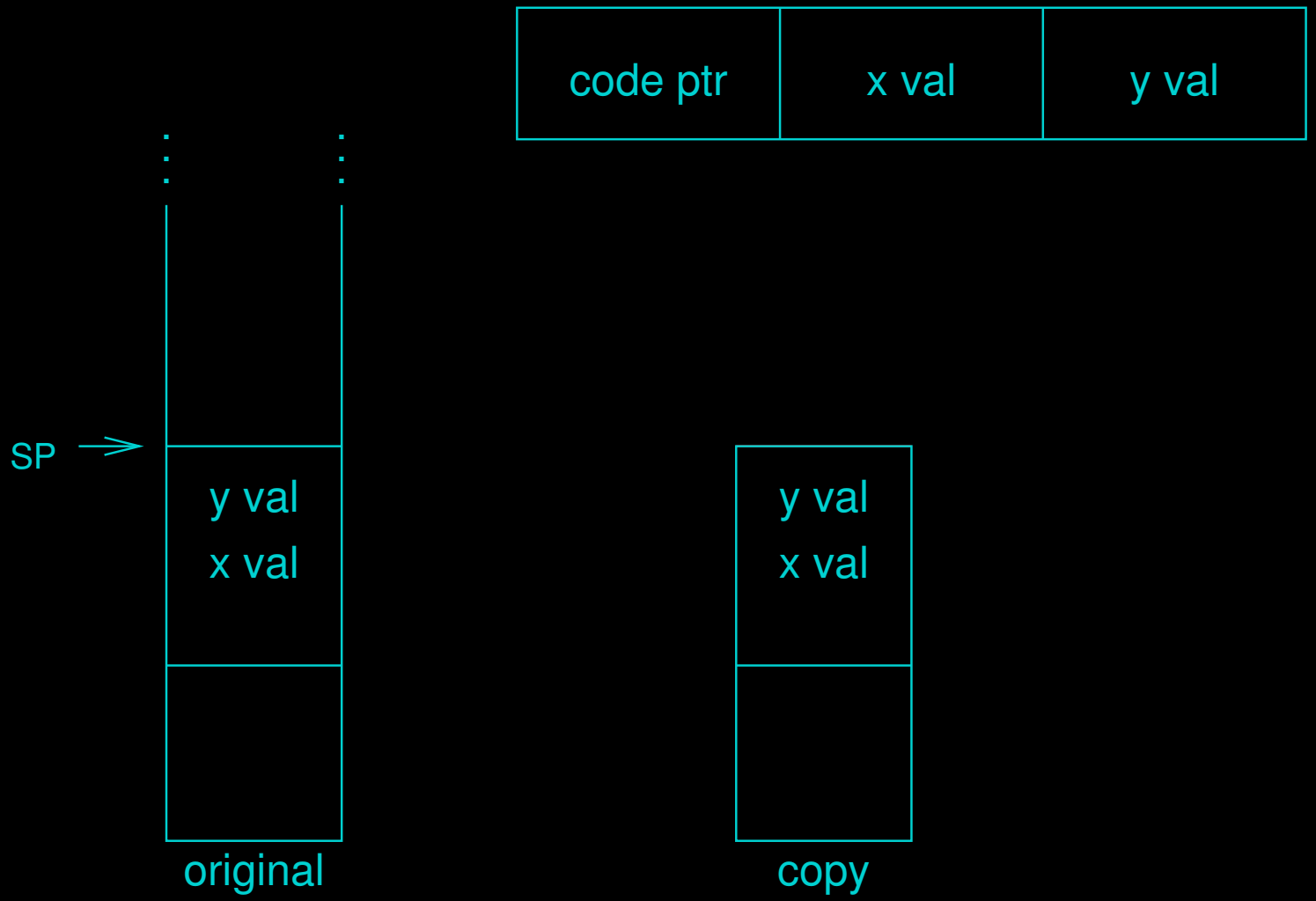
. . . closure creation “pays its own way”

# Assignment Conversion

Problem 2 again: variables with indefinite extent

- stack copy may replicate locations
- display closures may also replicate locations
- . . . how to track/update copies?

# Assignment Conversion



# Assignment Conversion

Observations:

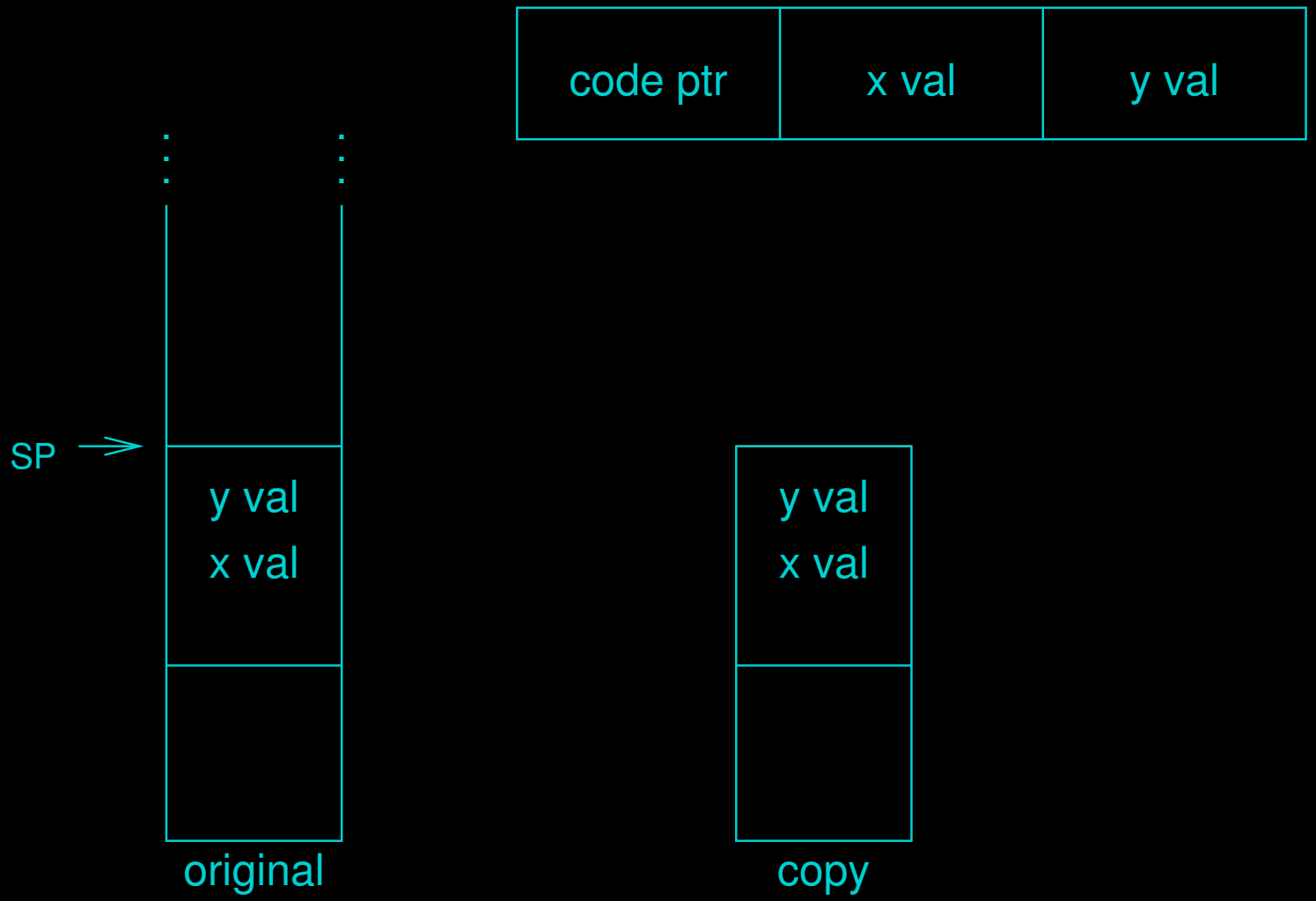
- problem only for assigned variables
- most variables are never assigned
  (in well-written Scheme code)

(Now obvious) solution:

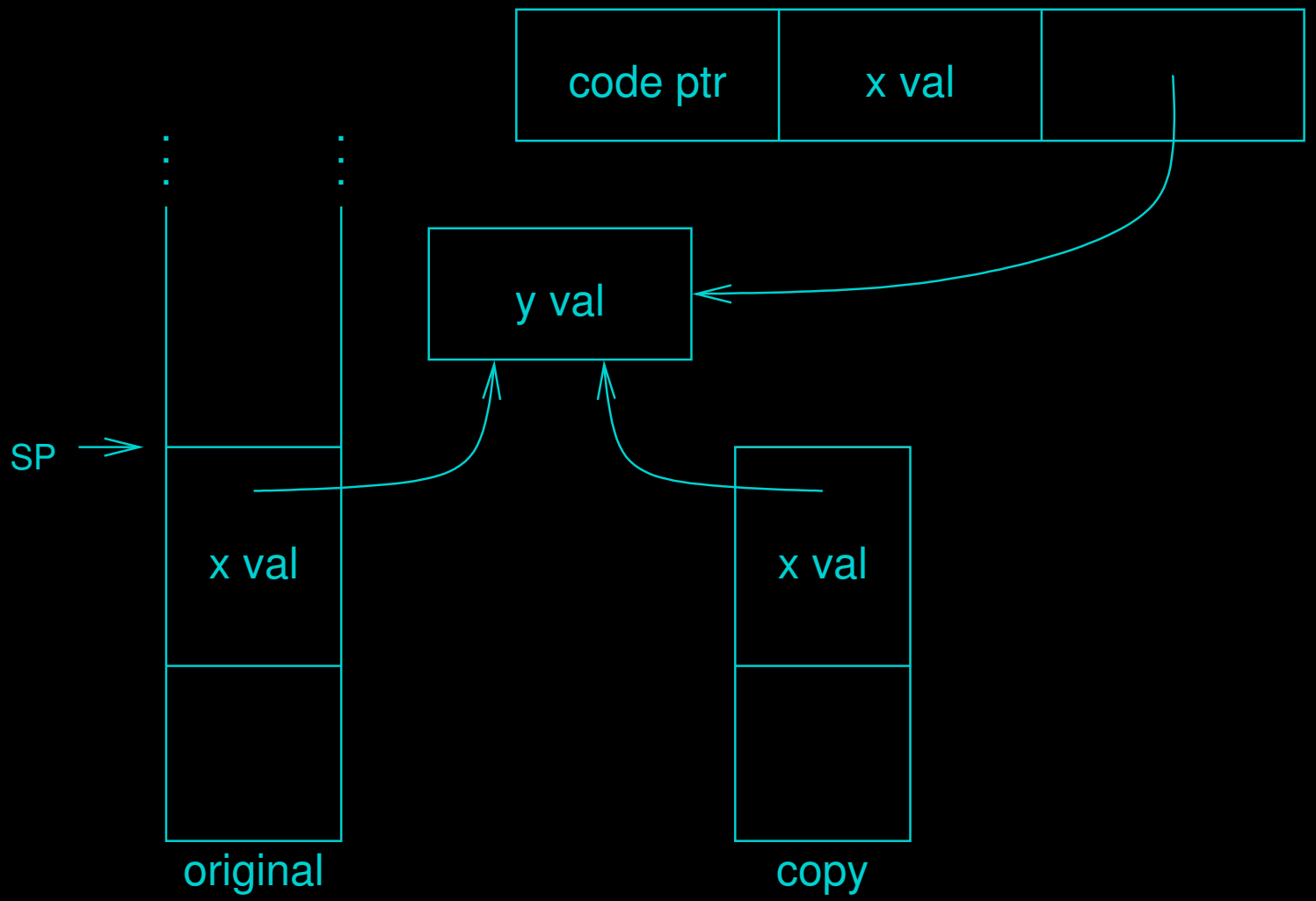
- add an indirect for assigned variables
- don't create multiple copies
- instead store each in heap-allocated box

. . . set! "pays its own way"

# Assignment Conversion



# Assignment Conversion



# Cost/Benefit Analysis

Costs:

- more continuation overhead
- potentially more closure-creation overhead
- assignment overhead (or not)

Benefits:

- less procedure-call overhead
- less variable reference overhead
- closures don’t hold onto garbage

# Getting Sucked In

Primary goal had been:

- to support PhD research

Primary goals became:

- to prove the design
- to create generally useful system

# Getting Sucked In

Led to further design decisions:

- compile “on the fly” ala Cardelli’s ML
- forget interpretation
- support floats, bignums, ratios
- fill out interactive environment
- solid documentation

# Version 1

R2RS compatibility

Simple compiler w/peephole optimization

Stop-and-copy collector

Single architecture, O/S (VAX BSD Unix)

Written mostly in Scheme; bootstrapped

Substantial test suite

Complete documentation

# Growth of the Implementation

Over course of six more major versions:

- adapted as Scheme language changed
- incorprated features demanded by users
- compiler became more sophisticated

Time here only for version highlights

Paper gives more detail

# Version 2

R3RS compatibility

EPS macros, extend-syntax

case-lambda

Destination-driven code generation

Faster code, faster compiler

# Aside: Optimization

Achieved faster code *and* compilation

Not as strange as it seems:

- compiler is written in Scheme
- benefits from its own optimization
- can overshadow additional costs

Actually became a hard rule:

- “optimizations must pay their own way”
- some optimizations didn’t make the cut

# Version 3

Interfaces to other languages, programs

Segmented stack

First RISC architecture ports

Faster code, faster compiler

# Version 4

R4RS and IEEE Scheme compatibility

Hybrid tagged pointers, BiBOP tagging model

Inline allocation

Generational garbage collection

Global (intraprocedural) register allocation

Faster code, faster compiler

# Version 5

syntax-case macros

Multiple return values

Guardians and weak pairs

25X faster gensym

Local call optimizations

Faster code, faster compiler

# Version 6

R5RS compatibility

Modules

Disjoint record types

Source-object correlation

Source optimizer/inliner

Petite Chez Scheme, with fast interpreter

Faster code, faster compiler

# Version 7

Multithreading (via Posix threads)

Scripts and compiled scripts

First 64-bit port

Faster code, faster compiler

# Published Techniques

Major techniques published elsewhere

- segmented stacks
- macros, modules
- multiple return values
- register allocation
- letrec optimization
- inlining
- float printing
- storage management

⋮

# Cool Hacks

Many others are no big deal . . .

. . . but still cool

Present here a small sample

# Lisp$_2$-inspired Hack (V1)

"Lisp$_2$" languages:

- separate namespaces for procedures and other values

Lisp$_2$ symbols typically contain two value cells:

- one for procedure values
- one for other values

Direct procedure calls use procedure-value cell

Cell contains "trap" procedure if no procedure value

# Lisp$_2$-inspired Hack (V1)

Scheme is a "Lisp$_1$" language:

- single namespace for procedures and other values

Scheme symbols require just one value slot

But this leads to procedure? test for every call

# Lisp$_2$-inspired Hack (V1)

Included a procedure-value slot anyway

- define, set! set it to “trap” address
- trap signals error if actual value unbound
- error also if actual value not a procedure
- otherwise, fixes procedure value slot, completes call

# Tail-recursive Trace Hack (V3)

Consider:

```
(define half
  (lambda (x)
    (cond
      [(zero? x) 0]
      [(odd? x) (half (- x 1))]
      [(even? x) (+ (half (- x 1)) 1)])))
```

For (half 3):

- 3 is odd, so half tail-calls itself with 2
- 2 is even, so half nontail-calls itself with 1
- 1 is odd, so half tail-calls itself with 0
- 0 is zero, so half returns 0

# Tail-recursive Trace Hack (V3)

```
(trace-define half
  (lambda (x)
    (cond
      [(zero? x) 0]
      [(odd? x) (half (- x 1))]
      [(even? x) (+ (half (- x 1)) 1)])))
```

Nontail-recursive trace

```
(half 3)
| (half 2)
| | (half 1)
| | | (half 0)
| | | 0
| | 0
| 1
1
```

Tail-recursive trace

```
(half 3)
(half 2)
| (half 1)
| (half 0)
| 0
1
```

# Tail-recursive Trace Hack (V3)

Trick is distinguishing tail from nontail calls

Other untraced calls are occurring

Call from one traced routine might be indirect

# Tail-recursive Trace Hack (V3)

Trick is distinguishing tail from nontail calls

Other untraced calls are occurring

Call from one traced routine might be indirect

Solution:

- eq? on continuations (seriously)

# Tail-recursive Trace Hack (V3)

Trace mechanism adds trace wrapper

Add global (fluid-bound) trace-cont variable

- trace wrapper grabs current continuation
- tail call iff eq? to trace-cont
- if not, rebinds trace-cont during call

Works even in presence of continuations

# Sharing Preservation Hack (V4)

Compiler and linker maintain sharing (and cycles):

```
(eq? '#1=(a) '#1#)  ⇒  #t
(eq? (car '(#1=(a) b)) '#1#)  ⇒  #t
```

Sharing preserved even across file compiles

# Sharing Preservation Hack (V4)

Compiler and linker maintain sharing (and cycles):

```
(eq? '#1=(a) '#1#)  ⇒  #t
(eq? (car '(#1=(a) b)) '#1#)  ⇒  #t
```

Sharing preserved even across file compiles

Allows blind copy propagation of quoted constants

```
(let ((x '(a b))) (eq? (cdr x) (cdr x)))
    → (eq? (cdr '#1=(a b)) (cdr '#1#))
    → (eq? '#2=(b) '#2#)
    ⇒ #t
```

# Efficient Gensym Hack (V5)

Basic idea: delay name generation

Store sequence number in name slot

- much cheaper than generating string
- `(gensym)` can be open coded
- generate name iff printed (lazy evaluation)

Completely transparent to the user

# Efficient Gensym Hack (V5/V7)

Later: stored `#f` in name slot

- cheaper still
- open code for `(gensym)` smaller still
- determine sequence number when printed

Gensyms numbered in print order

```
(let* ([x (gensym)] [y (gensym)] [z (gensym)])
  (list x z x))  ⇒  (#:g0 #:g1)

(let* ([x (gensym)] [y (gensym)] [z (gensym)])
  (list z x z))  ⇒  (#:g2 #:g3 #:g2)
```

This is often a good thing

# Efficient Gensym Hack (V5)

Eliminates name generation for unprinted gensyms

- 25X savings in this case

Also eliminates synchronization when threaded

- much more than 25X savings

Made switch to globally unique names less painful

# Continuation/longjmp Hack (V6)

Since C can call Scheme and Scheme can call C . . .

. . . a continuation may contain nested C/Scheme calls

If such a continuation is grabbed via call/cc:

- outward throw might return to wrong C frame
- inward throw might return to stale C frame
- but not necessarily . . .

# Continuation/longjmp Hack (V6)

longjmp (C throw) to the rescue:

- setjmp (C catch) at each C-to-Scheme call
- longjmp at each Scheme-to-C return
- mark setjmp buffer (and all above) invalid
- second attempted longjmp fails

Arbitrary use of continuations possible on Scheme side

# Lessons Learned

\begin{soapbox}

- General
- Reliability
- Performance

\end{soapbox}

# General

Build incrementally

Don’t overreach

Don’t overbuild

Avoid premature optimization

Avoid complexity when possible

Don’t forget machine-generated programs

Learn to enjoy deleting code

Overhaul when necessary

# Reliability

Learn to love bug reports

Accept no partially correct solutions

Don’t ignore edge cases (if it can happen, it will)

Don’t commit change without adding new tests

Always type tests into a file

Test error cases

# Performance

Speed of generated code important

Other aspects equaly so:

- compile speed
- garbage collection speed
- memory usage
- I/O speed
- run-time libraries

# Performance

Also important:

- uniformity
- continuity
- scalability

# Fast and effective compilation

Optimizations should “pay own way”

Leverage—get the bit level right

Use fast, sub-quadratic algorithms

- 90-95% benefit is okay
- leaves time for other optimization

Realistic whole program compilation possible