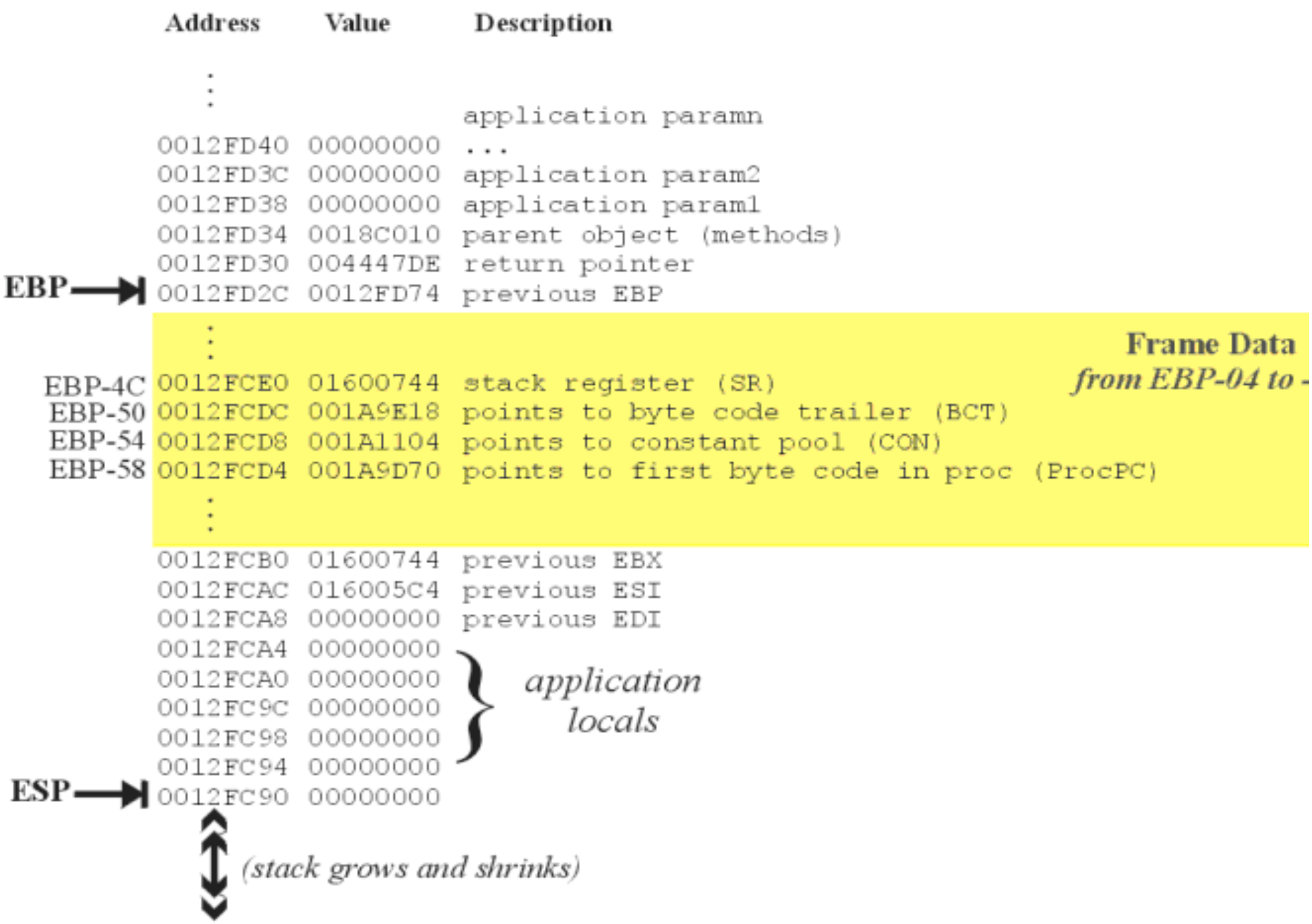
Essentials of Garbage Collection :-)

Gregory Cooper, Arjun Guha, and Shriram Krishnamurth


Address Value Description
application paramn
0012FD40 00000000 ...
0012FD3C 00000000 application param2
0012FD38 00000000 application param1
0012FD34 0018C010 parent object (methods)
0012FD30 004447DE return pointer
EBP
0012FD2C 0012FD74 previous EBP
Frame Data
from EBP-04 to -
EBP-4C 0012FCE0 01600744 stack register (SR)
EBP-50 0012FCDC 001A9E18 points to byte code trailer (BCT)
EBP-54 0012FCD8 001A1104 points to constant pool (CON)
EBP-58 0012FCD4 001A9D70 points to first byte code in proc (ProcPC)
0012FCB0 01600744 previous EBX
0012FCAC 016005C4 previous ESI
0012FCA8 00000000 previous EDI
0012FCA4 00000000
0012FCA0 00000000
0012FC9C 00000000
0012FC98 00000000
application locals
0012FC94 00000000
ESP
0012FC90 00000000
(stack grows and shrinks)

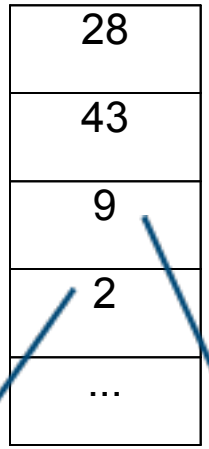

Address
Value
Description
application paramn
0012FD40 00000000 ...
0012FD3C 00000000 application param2
0012FD38 00000000 application param1
0012FD34 0018C010 par (methods)
0012FD30 004447DE
EBP
0012FD2C 0012FD74
Frame Data
from EBP-04 to -
EBP-4C 0012FCE0 0160 ister (
EBP-50 0012FCDC 001A byte co ailer (BCT)
EBP-54 0012FCD8 001A nstan (CON)
EBP-58 0012FCD4 001A de in proc (ProcPC)
0012FCB0 016007 ious EBX
0012FCAC 016005C4
0012FCA8 00000000
0012FCA4 00000000
0012FCA0 00000000
0012FC9C 00000000
0012FC98 00000000
0012FC94 00000000
application
locals
ESP
0012FC90 00000000
(stack grows and shrinks)

Goals

- Minimal curricular dependencies
- Minimal architectural detail
- Abstraction of *physical* memory
- Easy testing and debugging

Abstractions

Continuation



Store

'atom	'empty	'atom	1	'cons
0	2	'atom	2	'cons
7	4	...	...	...

Writing a collector isn't so bad now.

But what about testing and debugging?

Exercising Collectors (Manual Approach)

```
(define (tree-add1 tree)
  (cond
    [(empty? tree) empty]
    [(number? tree) (add1 tree)]
    [(cons? tree)
     (cons (tree-add1 (car tree))
           (tree-add1 (cdr tree)))]))
```

Exercising Collectors (Manual Approach)

```
(define (tree-add1 tree)
  (cond
    [(gc:empty? tree) (gc:empty)]
    [(gc:number? tree) (gc:add1 tree)]
    [(gc:cons? tree)
     (gc:cons (tree-add1 (gc:car tree))
              (tree-add1 (gc:cdr tree)))]))
```

Exercising Collectors (Manual Approach)

```
(define (tree-add1/k tree k)
  (cond
    [(gc:empty? tree) (k (gc:empty))]
    [(gc:number? tree) (k (gc:add1 tree))]
    [(gc:cons? tree)
     (tree-add1/k (gc:car tree)
       (lambda (lv)
         (tree-add1/k (gc:cdr tree)
           (lambda (rv)
             (k (gc:cons lv rv)))))))]))
```

Exercising Collectors (Manual Approach)

```
(define (tree-add1/k tree)
  (cond
    [(gc:empty? tree) (pop (gc:empty))]
    [(gc:number? tree) (gc:add1 tree)]
    [(gc:cons? tree)
     (push (make-left-frame (gc:cdr tree)))
     (tree-add1/k (gc:car tree))]))

(define (pop val)
  (match (stack-pop!)
    [(struct left-frame rt)
     (push (make-right-frame val))
     (tree-add1/k rt)]
    [(struct right-frame lv)
     (pop (gc:cons lv val))]))
```

Exercising Collectors (Manual Approach)

```
(define (tree-add1/k tree)
  (cond
    [(gc:empty? tree) (pop (gc:empty))]
    [(gc:number? tree) (gc:add1 tree)]
    [(gc:cons? tree)
     (push (make-left-frame (gc:cdr tree)))
     (tree-add1/k (gc:car tree))]))

(define (pop val)
  (match (stack-pop!)
    [(struct left-frame rt)
     (push (make-right-frame val))
     (tree-add1/k rt)]
    [(struct right-frame lv)
     (pop (gc:cons lv val))]))
```

Manual Transformation: Consequences

- Contorted program structure
- Curricular dependency on CPS
- Tedious and time-consuming
- Tight coupling of collector-mutator pairs
- "Test data problem"

Alternative

Compilation or interpretation:

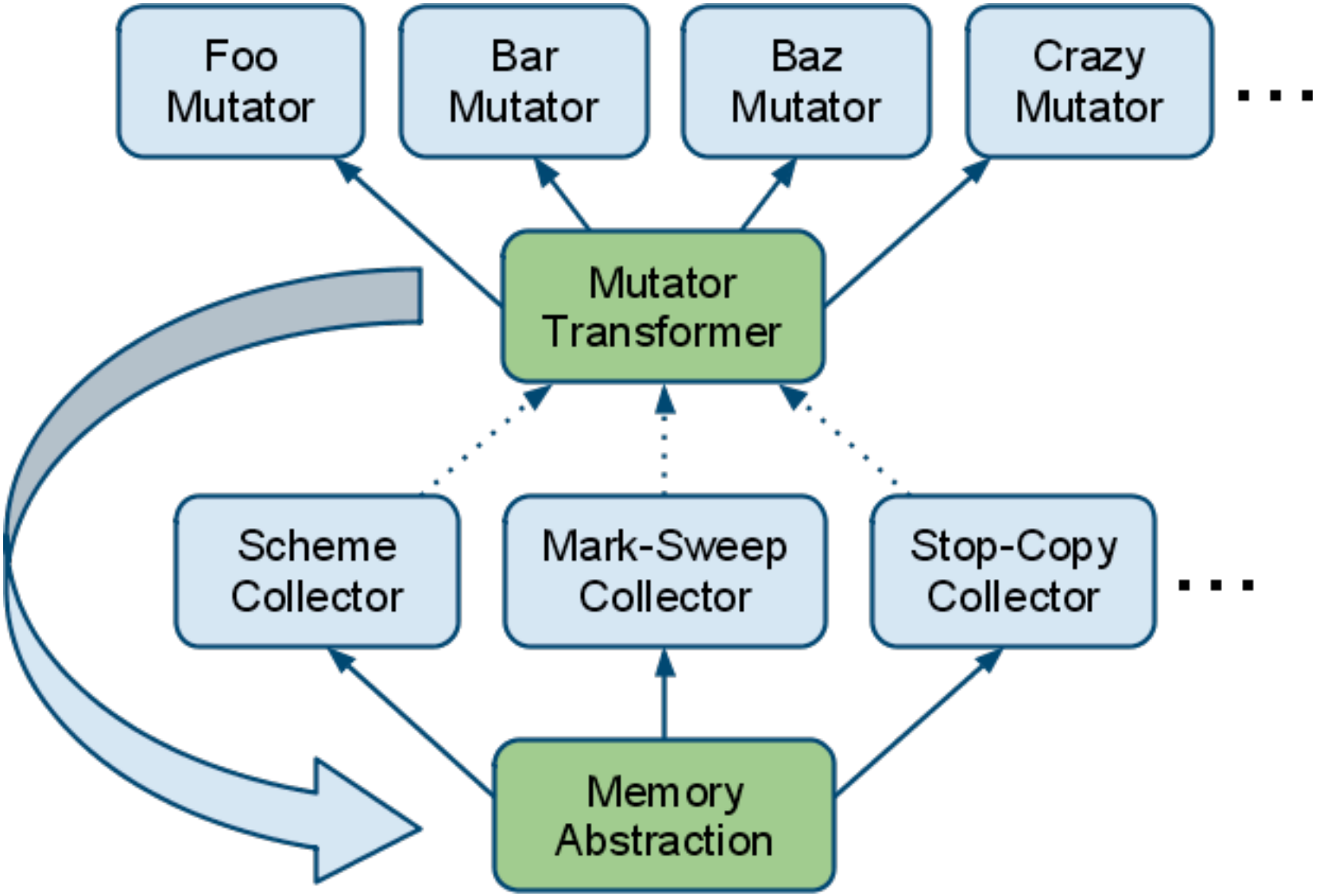
- Dependence on CPS
- Difficulty debugging
- Restricted mutators

Solution Approach

Lightweight compilation and stack-inspection:

- Ordinary Scheme syntax
- Normal control flow

Solution Architecture



Continuation Marks

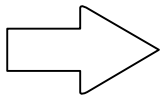
```
(define (traverse tree)
  (with-continuation-mark tree
    (cond
      [(empty? tree)
       (display (current-continuation-marks))
       empty)]
      [(cons? tree)
       (cons (traverse (car tree))
             (traverse (cdr tree))]
      [else tree])))

(traverse '(1 2)) ; prints: (() (2) (1 2))
```

- Continuation marks use the *real* stack, so no more CPS.

Representing & Recording Roots

```
(let ([x (cons 1 2)])
  (tree-add1 x)
```

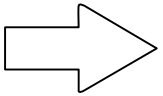


```
(let ([x (cons 1 2)])
  (with-continuation-mark
    (list
      (case-lambda
        [() x]
        [(v) (set! x v)]))
    (tree-add1 x)))
```

- Track variables in scope, wrap each non-tail-position procedure application in a **`w-c-m`** that records roots.
- Implementation: **`syntax-case`** and **`#%module-begin`**.

Handling Anonymous Roots

```
(cons
  (cons 1 empty)
  (cons 2 empty))
```



```
(let* ([t1 (cons 1 empty)]
       [t2 (cons 2 empty)])
  (cons t1 t2))
```

- As a separate pre-processing step, convert all procedure applications to ANF.

Wrapping Primitive Operators

```
;; gc:add1 : addressof number -> addressof number
(define (gc:add1 addr)
  (gc:alloc-flat (add1 (gc:deref addr))))

;; In general:
(define ((lift prim) . args)
  (gc:alloc-flat (apply prim (map gc:deref args))))
```

Putting It Together

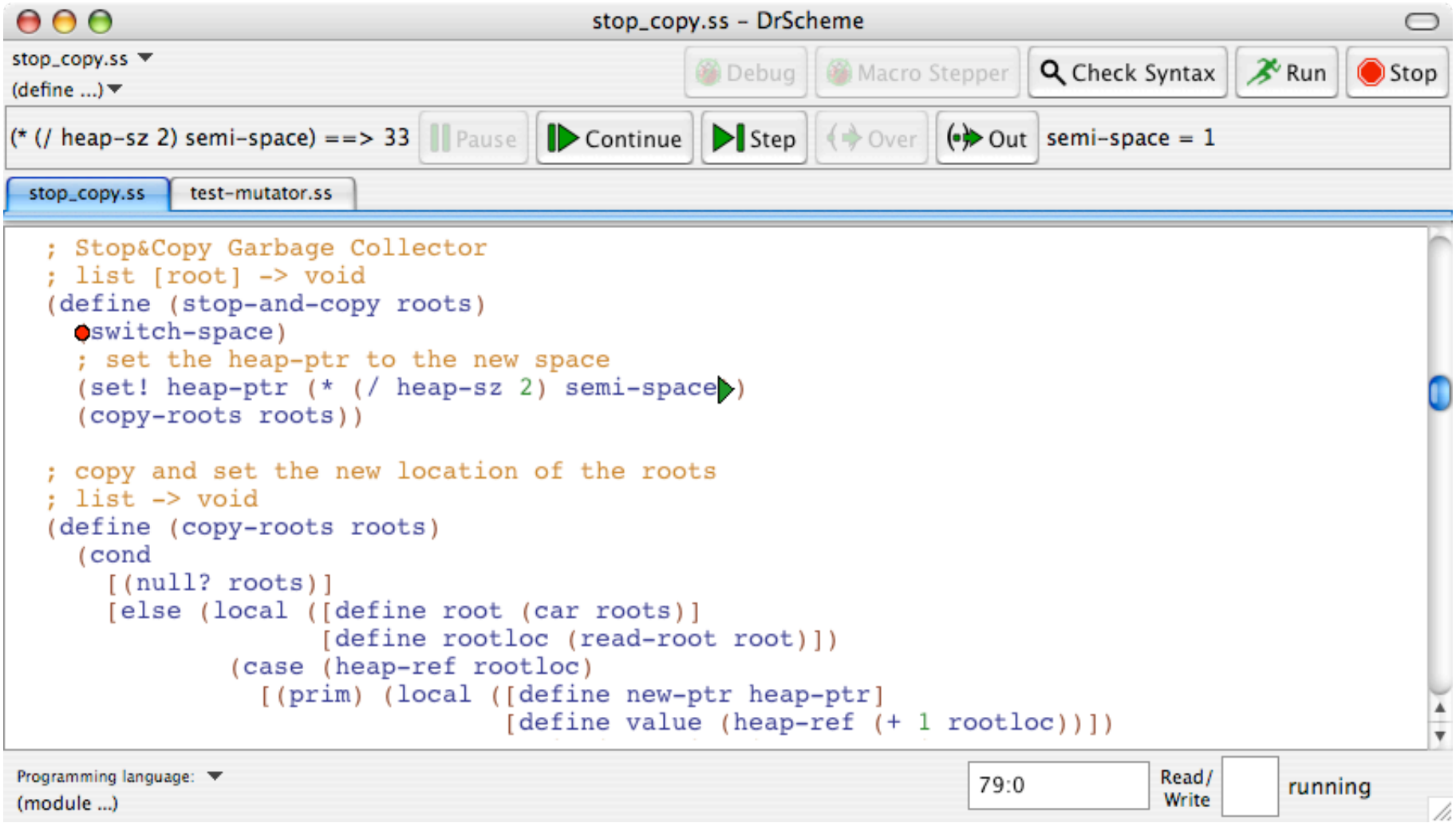
```
(module mutator-lang scheme

  (define-syntax gc:app ...)
  (define-syntax (gc:module-begin stx)
    (syntax-case (expand-syntax stx) ...))
  (define-values (gc:add1 gc:+ gc:- gc:* ...)
    ...)
  (provide
    (rename gc:app #%app)
    (rename gc:module-begin #%module-begin))
    (rename gc:add1 add1)
    (rename gc:+ +)
    ...))
```

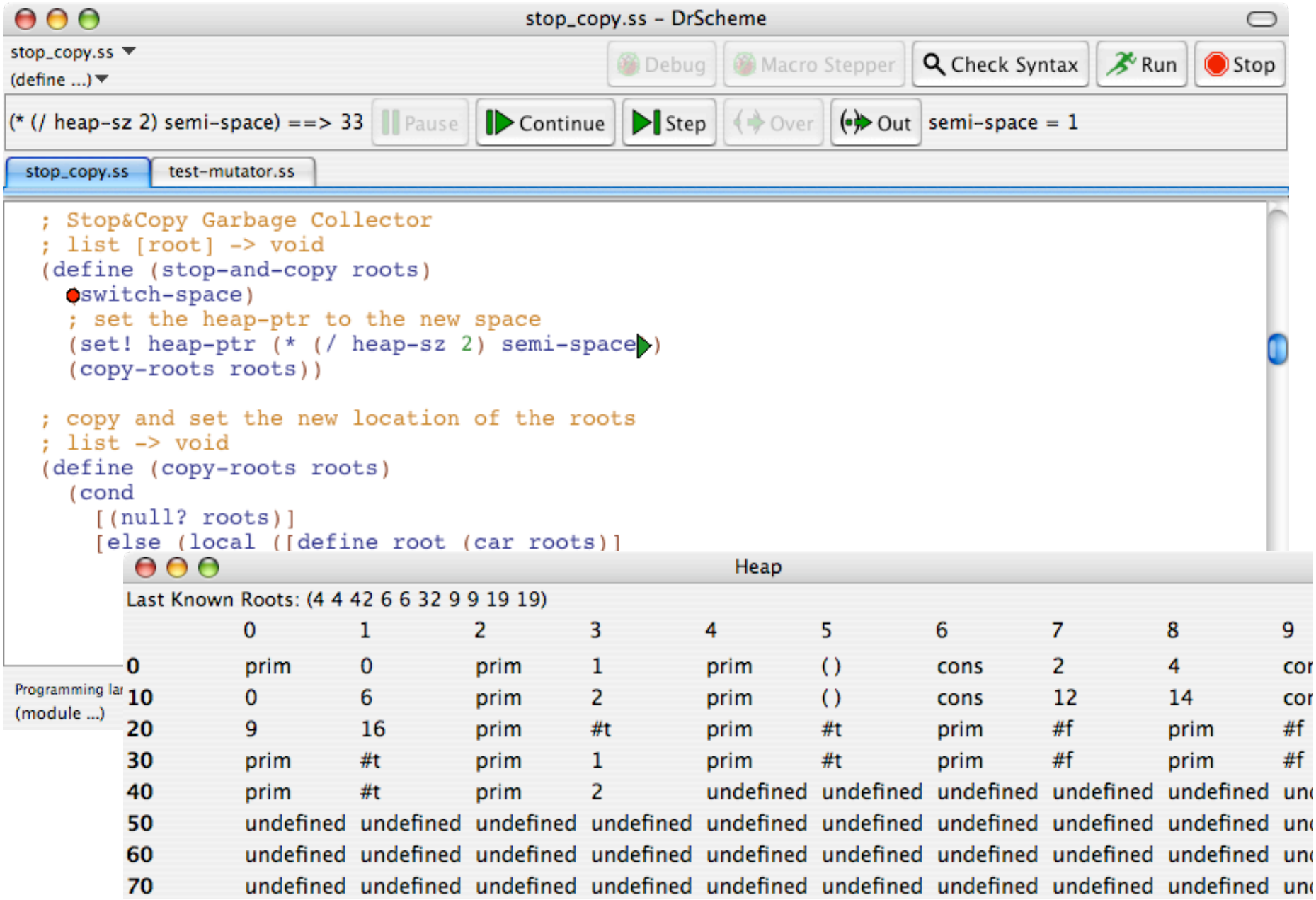
Putting It Together

```
#lang mutator-lang
(allocator-setup 50 "stop-copy.ss")
(define (tree-add1 tree)
  (cond
    [(empty? tree) empty]
    [(number? tree) (add1 tree)]
    [(cons? tree)
     (cons (tree-add1 (car tree))
           (tree-add1 (cdr tree)))]))
```

Debugging Support



Debugging Support



	0	1	2	3	4	5	6	7	8	9
0	prim	0	prim	1	prim	()	cons	2	4	cor
10	0	6	prim	2	prim	()	cons	12	14	cor
20	9	16	prim	#t	prim	#t	prim	#f	prim	#f
30	prim	#t	prim	1	prim	#t	prim	#f	prim	#f
40	prim	#t	prim	2	undefined	undefined	undefined	undefined	undefined	und
50	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	und
60	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	und
70	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	und

Omitted Details (provided in the paper)

- Top-level variables
- Closures
- Tail calls

Conclusions

- Well-designed macro and module systems can implement powerful transformations with surprising ease.
- Implementing and testing a garbage collector can be fun.
- Students do seem to learn better by implementing.

http://www.plai.org/