Nested Functions & Memory Management

Eelco Visser

TU Delft

CS4200 | Compiler Construction | December 10, 2020

This Lecture

Functions, Revisited

- activation records
- nested functions
- static links

Miscellaneous

- Statements
- String Constants
- Execution Environment

Memory Management

- memory safety
- garbage collection algorithms

Functions, Revisited

Functions in ChocoPy

function name

local variables

return to caller

call function

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```

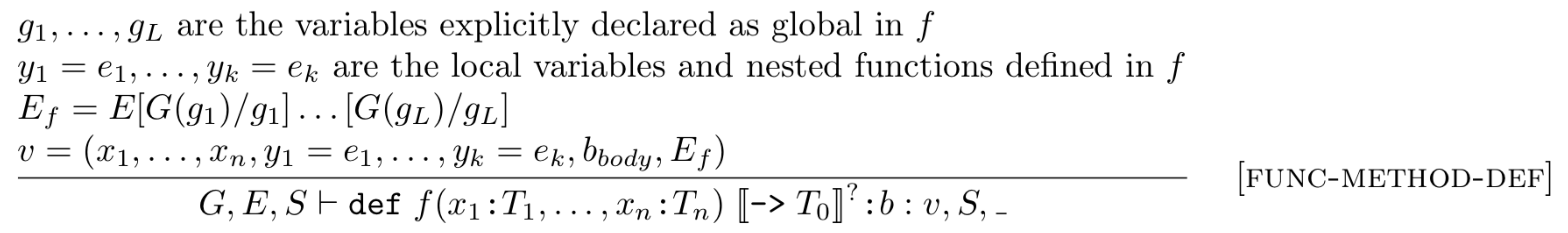
formal parameters

actual parameters

Operational Semantics: Invoke Function

$$\frac{\begin{array}{l}
S_0(E(f)) = (x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\
n, k \geq 0 \\
G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\
\vdots \\
G, E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\
l_{x1}, \ldots, l_{xn}, l_{y1}, \ldots, l_{yk} = newloc(S_n, n+k) \\
E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\
G, E', S_n \vdash e'_1 : v'_1, S_n, \_ \\
\vdots \\
G, E', S_n \vdash e'_k : v'_k, S_n, \_ \\
S_{n+1} = S_n[v_1/l_{x1}] \ldots [v_n/l_{xn}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\
G, E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R \\
R' = \begin{cases} None, & \text{if } R \text{ is } \_ \\ R, & \text{otherwise} \end{cases}
\end{array}}{G, E, S_0 \vdash f(e_1, \ldots, e_n) : R', S_{n+2}, \_} \quad [\text{INVOKE}]$$

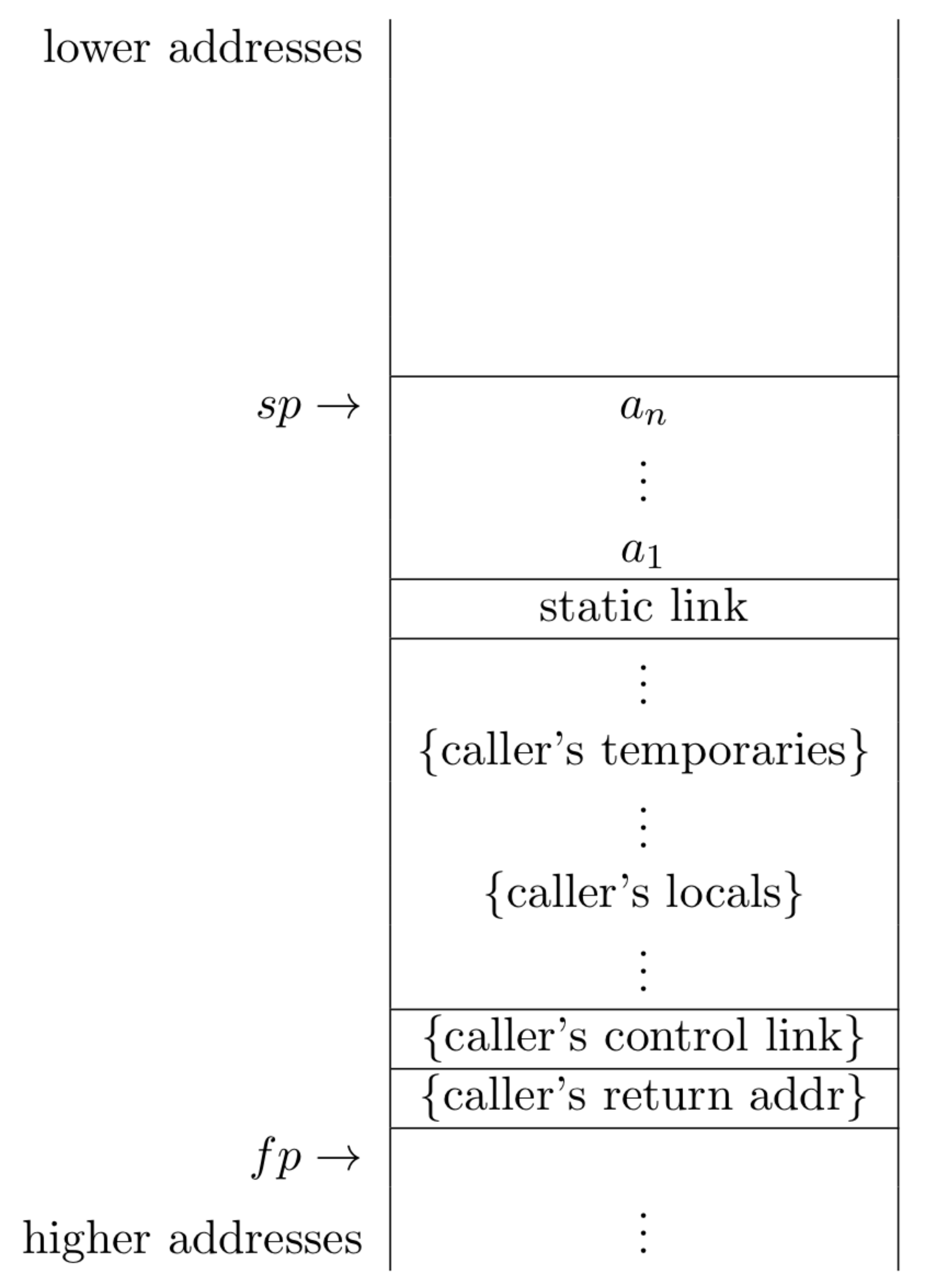
Operational Semantics: Define Function



$$\frac{\begin{array}{l} g_1, \ldots, g_L \text{ are the variables explicitly declared as global in } f \\ y_1 = e_1, \ldots, y_k = e_k \text{ are the local variables and nested functions defined in } f \\ E_f = E[G(g_1)/g_1] \ldots [G(g_L)/g_L] \\ v = (x_1, \ldots, x_n, y_1 = e_1, \ldots, y_k = e_k, b_{body}, E_f) \end{array}}{G, E, S \vdash \texttt{def}\ f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\texttt{->}\ T_0]\!]^? : b : v, S, \_} \quad [\text{FUNC-METHOD-DEF}]$$

Activation Records

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```



lower addresses

$sp \rightarrow$

a_n

$\vdots$

a_1

static link

$\vdots$

{caller's temporaries}

$\vdots$

{caller's locals}

$\vdots$

{caller's control link}

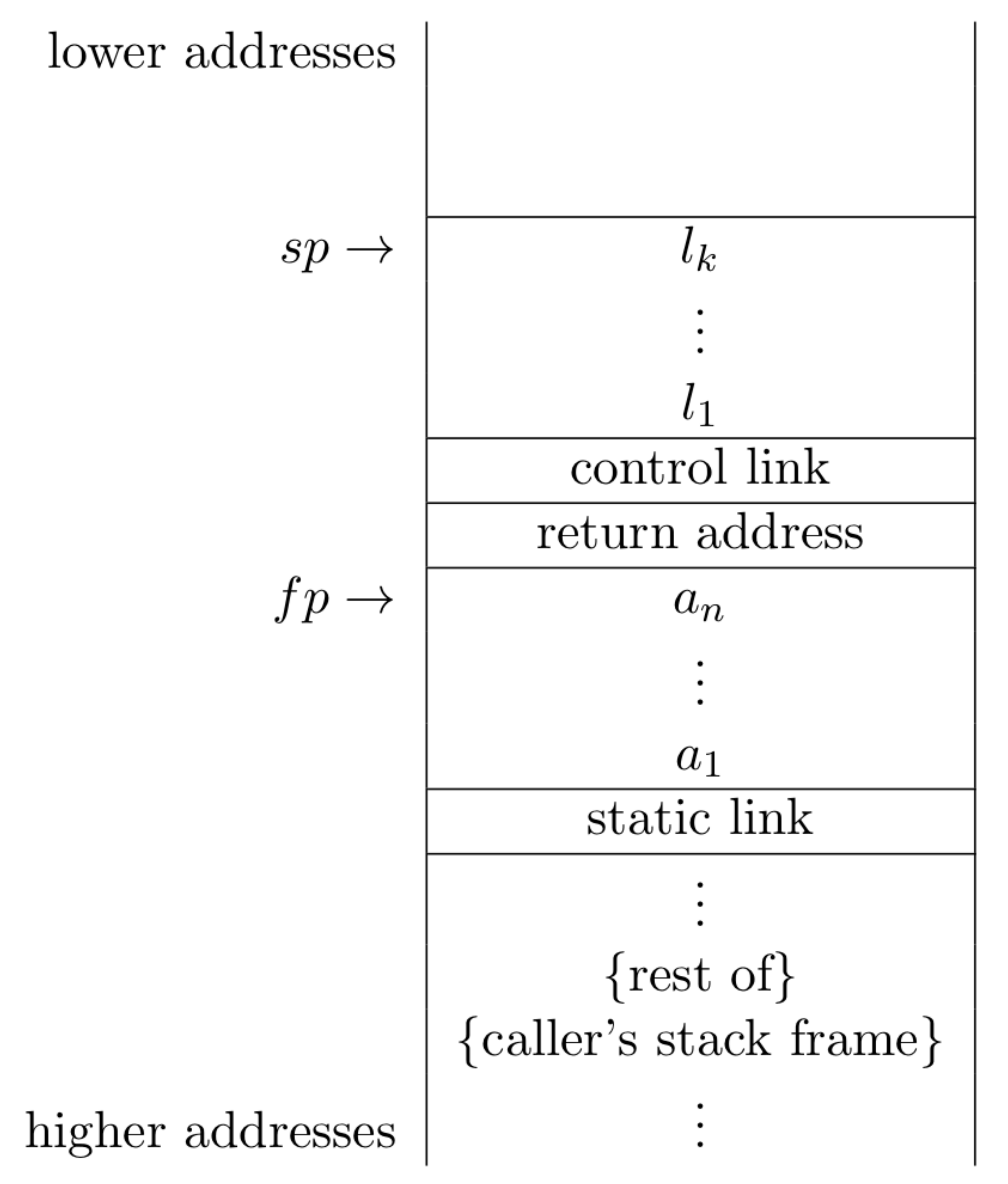
{caller's return addr}

$fp \rightarrow$

higher addresses

$\vdots$

before/after invocation



lower addresses

$sp \rightarrow$

l_k

$\vdots$

l_1

control link

return address

$fp \rightarrow$

a_n

$\vdots$

a_1

static link

$\vdots$

{rest of}

{caller's stack frame}

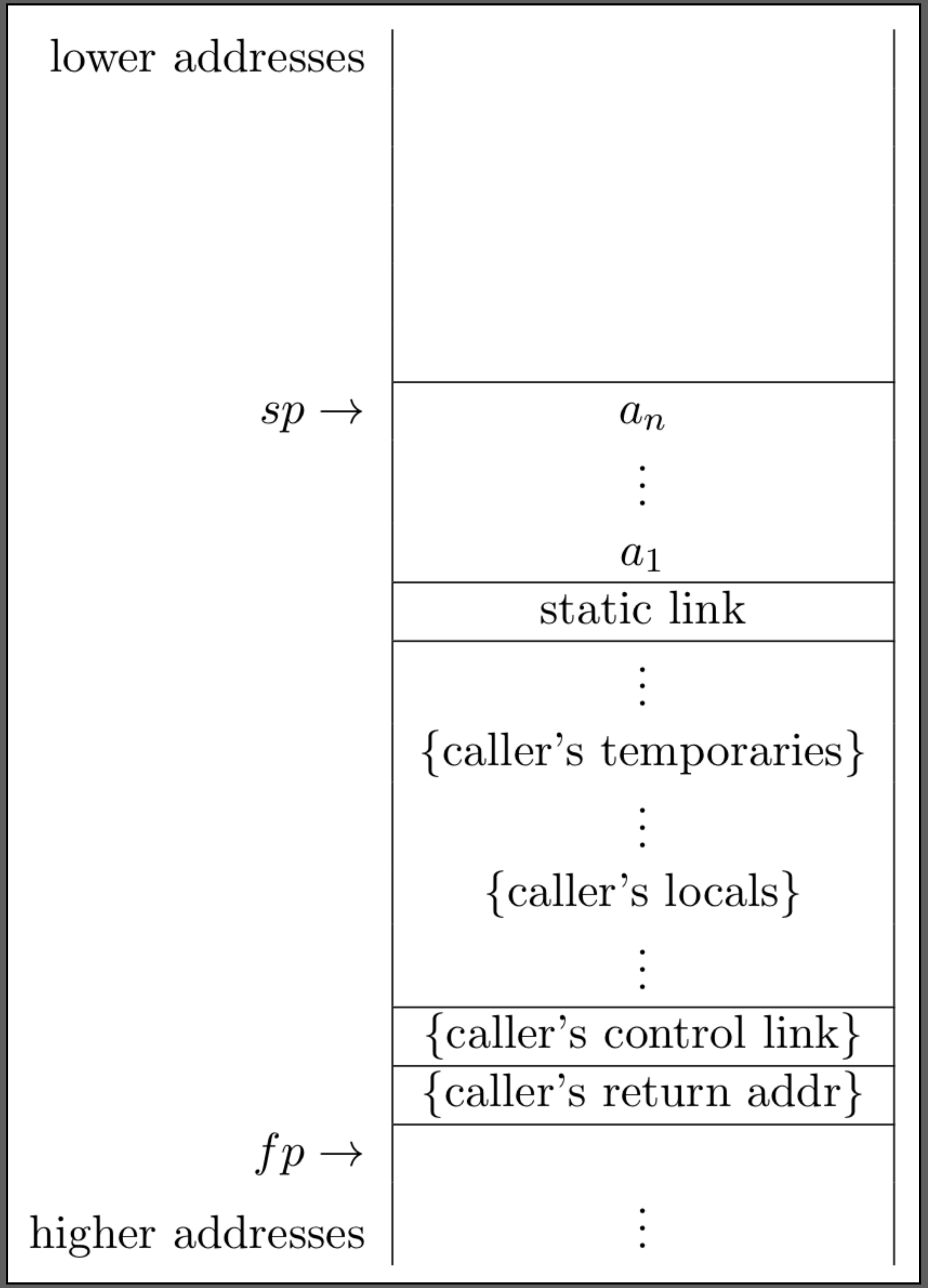
higher addresses

$\vdots$

during callee's execution

Calling Convention: Caller

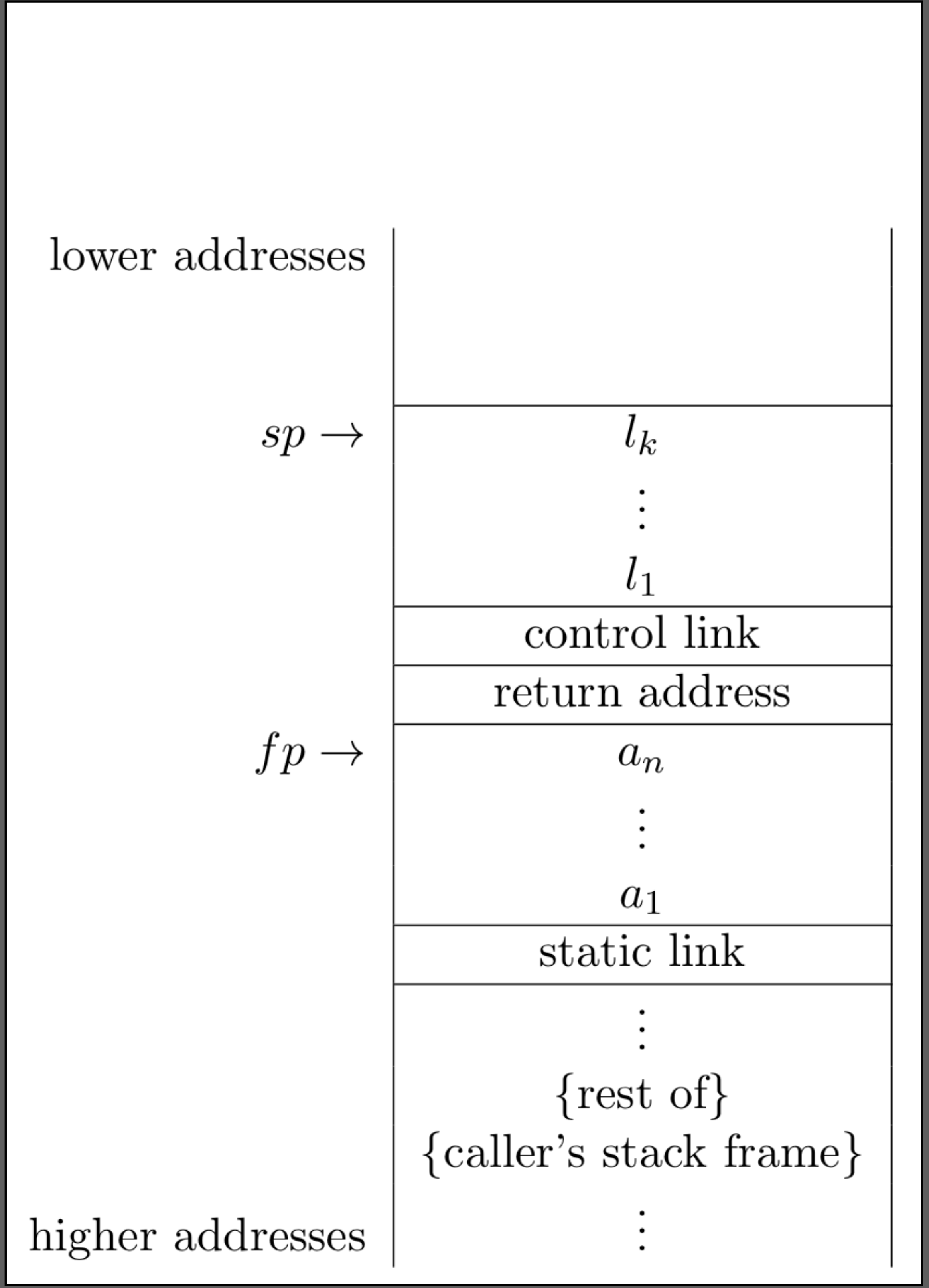
```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```



```
.globl $caller

$caller:
  addi    sp, sp, -@$caller.size     # Reserve space for stack frame
  sw      ra, @$caller.size-4(sp)    # Save return address
  sw      fp, @$caller.size-8(sp)    # Save control link (fp)
  addi    fp, sp, @$caller.size      # New fp is at old SP.
  li      a0, 0                      # Load integer constant 0
  sw      a0, -12(fp)                # init local variable $caller.d
  addi    sp, sp, -12                # allocate space for actual arguments
  li      a0, 235                    # Load integer constant 235
  sw      a0, 0(sp)                  # push argument on stack
  li      a0, 4357                   # Load integer constant 4357
  sw      a0, 4(sp)                  # push argument on stack
  li      a0, 345                    # Load integer constant 345
  sw      a0, 8(sp)                  # push argument on stack
  jal     $callee                    # call function $callee
  addi    sp, fp, -@$caller.size     # restore stack pointer
  sw      a0, -12(fp)                # write local variable $caller.d
label_97:
.equiv @$caller.size, 12             # Epilogue of $caller
  lw      ra, -4(fp)                 # Restore return address
  lw      fp, -8(fp)                 # Restore caller's fp
  jr      ra                         # Return to caller
```

Calling Convention: Callee

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```



```
$callee:
  addi    sp, sp, -@$callee.size    # Reserve space for stack frame
  sw      ra, @$callee.size-4(sp)   # Save return address
  sw      fp, @$callee.size-8(sp)   # Save control link (fp)
  addi    fp, sp, @$callee.size     # New fp is at old SP.
  li      a0, 1                     # Load integer constant 1
  sw      a0, -12(fp)               # init local variable $callee.a
  li      a0, 2                     # Load integer constant 2
  sw      a0, -16(fp)               # init local variable $callee.b
  lw      a0, 8(fp)                 # read formal parameter $callee.x
  lw      t1, 4(fp)                 # read formal parameter $callee.y
  add     a0, a0, t1                # Addition
  lw      t1, 0(fp)                 # read formal parameter $callee.z
  add     a0, a0, t1                # Addition
  lw      t1, -12(fp)               # read local variable $callee.a
  add     a0, a0, t1                # Addition
  lw      t1, -16(fp)               # read local variable $callee.b
  add     a0, a0, t1                # Addition
  j       label_96
label_96:
.equiv @$callee.size, 16            # Epilogue of $callee
  lw      ra, -4(fp)                # Restore return address
  lw      fp, -8(fp)                # Restore caller's fp
  jr      ra                        # Return to caller
```

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) -> int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) -> int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

```
def callee(x : int, y : int, z : int) -> int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) -> int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

problem: callee overwrites registers for temporaries

solution: lift calls from call expressions
store result in local variable

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

```
def callee(x : int, y : int, z : int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

solution: lift calls from call expressions
store result in local variable

```
$caller:
  addi    sp, sp, -@$caller.size     # Reserve space for stack frame
  sw      ra, @$caller.size-4(sp)    # Save return address
  sw      fp, @$caller.size-8(sp)    # Save control link (fp)
  addi    fp, sp, @$caller.size      # New fp is at old SP.
  li      a0, 0                      # Load integer constant 0
  sw      a0, -12(fp)                # init local variable $caller.d
  li      a0, 0                      # Load integer constant 0
  sw      a0, -16(fp)                # init local variable $caller.temp_2
  addi    sp, sp, -4                 # allocate space for actual arguments
  li      a0, 13                     # Load integer constant 13
  sw      a0, 0(sp)                  # push argument on stack
  jal     $inc                       # call function $inc
  addi    sp, fp, -@$caller.size     # restore stack pointer
  sw      a0, -16(fp)                # write local variable $caller.temp_2
  addi    sp, sp, -12                # allocate space for actual arguments
  li      a0, 235                    # Load integer constant 235
  sw      a0, 0(sp)                  # push argument on stack
  li      a0, 4357                   # Load integer constant 4357
  sw      a0, 4(sp)                  # push argument on stack
  li      a0, 345                    # Load integer constant 345
  addi    a0, a0, 81                 # Add with constant 81
  lw      t1, -16(fp)                # read local variable $caller.temp_2
  add     a0, a0, t1                 # Addition
  sw      a0, 8(sp)                  # push argument on stack
  jal     $callee                    # call function $callee
  addi    sp, fp, -@$caller.size     # restore stack pointer
  sw      a0, -12(fp)                # write local variable $caller.d

label_98:
.equiv @$caller.size, 16             # Epilogue of $caller
  lw      ra, -4(fp)                 # Restore return address
  lw      fp, -8(fp)                 # Use control link to restore caller's fp
  jr      ra                         # Return to caller
```

Shadowing

Shadowing

```
a : int = 10

def foo(a: int) -> int:
  def foo(b : int) -> int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```

problem: identifier can be used for multiple declarations

Shadowing

```
a : int = 10

def foo(a: int) -> int:
  def foo(b : int) -> int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```

```
$a : int = 10

def $foo($foo.a: int) -> int:
  def $foo.foo($foo.foo.b : int) -> int:
    $foo.foo.a : int = 20
    return $foo.foo.a + $foo.foo.b
  return $foo($foo.a + 10)

print($foo($a))
```

problem: identifier can be used for multiple declarations

solution: rename identifiers so that declarations have unique names

Shadowing

```
a : int = 10

def foo(a: int) -> int:
  def foo(b : int) -> int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```

```
$a : int = 10

def $foo($foo.a: int) -> int:
  def $foo.foo($foo.foo.b : int) -> int:
    $foo.foo.a : int = 20
    return $foo.foo.a + $foo.foo.b
  return $foo.foo($foo.a + 10)

print($foo($a))
```

problem: identifier can be used for multiple declarations

solution: rename identifiers so that declarations have unique names

implementation: dynamic rule to rename function and variable names (sketch)

```
f2 := $[[<Parent>].[f1]];
rules(
  FunctionName : f1 -> f2
)
```

Nested Functions

Closed Nested Functions are Just Functions

global variable

nested function definition

```
x : int = 10

def foo(y : int) -> int:
  def bar(z : int) -> int:
    return z + 10
  return bar(y + 10)

print(foo(x))
```

reference to local variable

reference to local variable

reference to global variable

Closed Nested Functions are Just Functions

```
x : int = 10

def foo(y : int) -> int:
  def bar(z : int) -> int:
    return z + 10
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size      # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp)     # Save return address
  sw     fp, @$foo.size-8(sp)     # Save control link (fp)
  addi   fp, sp, @$foo.size       # New fp is at old SP.
  addi   sp, sp, -4               # allocate space for actual arguments
  lw     a0, 0(fp)                # read formal parameter $foo.y
  addi   a0, a0, 10               # Add with constant 10
  sw     a0, 0(sp)                # push argument on stack
  jal    $foo.bar                 # call function $foo.bar
  addi   sp, fp, -@$foo.size      # restore stack pointer
  …
  jr     ra                       # Return to caller

.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  lw     a0, 0(fp)                # read formal parameter $foo.bar.z
  addi   a0, a0, 10               # Add with constant 10
  …
  jr     ra                       # Return to caller
```

nested function name is hidden from context

but otherwise it is a normal function

Nested Functions with ‘Free’ Variables

global variable

nested function definition

```
x : int = 10

def foo(y : int) -> int:
  def bar(z : int) -> int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

reference to variable in enclosing function

reference to local variable

reference to global variable

Accessing Lexically Enclosing Frame via Static Link

```python
x : int = 10

def foo(y : int) -> int:
  def bar(z : int) -> int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo.bar
  addi   sp, fp, -@$foo.size  # restore stack pointer
  j      label_105
label_105:
.equiv @$foo.size, 8          # Epilogue of $foo
  lw     ra, -4(fp)           # Restore return address
  lw     fp, -8(fp)           # Use control link to restore caller's fp
  jr     ra                   # Return to caller
```

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  lw     t0, 0(fp)                # load static link 1
  lw     a0, 0(t0)                # read variable $foo.y
  lw     t1, 4(fp)                # read formal parameter $foo.bar.z
  add    a0, a0, t1               # Addition
  j      label_106
label_106:
.equiv @$foo.bar.size, 8          # Epilogue of $foo.bar
  lw     ra, -4(fp)               # Restore return address
  lw     fp, -8(fp)               # Use control link to restore caller's fp
  jr     ra                       # Return to caller
```

Accessing Lexically Enclosing Frame via Static Link

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo.bar
  addi   sp, fp, -@$foo.size  # restore stack pointer
  …
  jr     ra                   # Return to caller
```

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  lw     t0, 0(fp)                # load static link 1
  lw     a0, 0(t0)                # read variable $foo.y
  lw     t1, 4(fp)                # read formal parameter $foo.bar.z
  add    a0, a0, t1               # Addition
  …
  jr     ra                       # Return to caller
```

Offset in Activation Record

```
x : int = 10

def foo(y : int) -> int:
  a : int = 0

  def bar(z : int) -> int:
    b : int = 0
    b = z
    return a + b + x

  a = y + 1
  return bar(y + 10)

print(foo(x))
```

Offset in Activation Record

```
x : int = 10

def foo(y : int) → int:
  a : int = 0

  def bar(z : int) → int:
    b : int = 0
    b = z
    return a + b + x

  a = y + 1
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  li     a0, 0                # Load integer constant 0
  sw     a0, -12(fp)          # init local variable $foo.a
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 1            # Add with constant 1
  sw     a0, -12(fp)          # write local variable $foo.a
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo
  addi   sp, fp, -@$foo.size  # restore stack poin
  …
  jr     ra                   # Return to caller
```

offset from frame pointer

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  li     a0, 0                  # Load integer constant 0
  sw     a0, -12(fp)            # init local variable $foo.bar.b
  lw     a0, 4(fp)              # read formal parameter $foo.bar.z
  sw     a0, -12(fp)            # write local variable $foo.bar.b
  lw     t0, 0(fp)              # load static link 1
  lw     a0, -12(t0)            # read variable $foo.a
  lw     t1, -12(fp)            # read local variable $foo.bar.b
  add    a0, a0, t1             # Addition
  lw     t1, $x                 # read global variable $x
  add    a0, a0, t1             # Addition
  …
  jr     ra                     # Return to caller
```

same offset from static link

Recursive Nested Functions

nested function definition

```
def exp(base: int, n: int) -> int:
  def aux(x: int) -> int:
    if x == 0:
      return 1
    else:
      return base * aux(x - 1)
  return aux(n)

print(exp(2, 4))
```

reference to variable in lexically enclosing function

Recursive Nested Functions

```python
def exp(base: int, n: int) -> int:
  def aux(x: int) -> int:
    if x == 0:
      return 1
    else:
      return base * aux(x - 1)
  return aux(n)

print(exp(2, 4))
```

nested function definition

```
.globl $exp.aux
$exp.aux:
  addi   sp, sp, -@$exp.aux.size  # Reserve space for stack frame
  sw     ra, @$exp.aux.size-4(sp) # Save return address
  sw     fp, @$exp.aux.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$exp.aux.size   # New fp is at old SP.
  li     a0, 0                    # Load integer constant 0
  sw     a0, -12(fp)              # init local variable temp_29
  lw     a0, 4(fp)                # read formal parameter $exp.aux.x
  li     t1, 0                    # Load integer constant 0
  xor    a0, a0, t1               # Test integer equality
  seqz   a0, a0
  beqz   a0, false_3
  li     a0, 1                    # Load integer constant 1
  j      label_110
  j      end_3
false_3:
  addi   sp, sp, -8               # allocate space for actual arguments
  lw     t0, 0(fp)                # load static link 1
  sw     t0, 0(sp)                # pass static link as parameter
  lw     a0, 4(fp)                # read formal parameter $exp.aux.x
  li     t1, 1                    # Load integer constant 1
  sub    a0, a0, t1               # Subtraction
  sw     a0, 4(sp)                # push argument on stack
  jal    $exp.aux                 # call function $exp.aux
  addi   sp, fp, -@$exp.aux.size  # restore stack pointer
  sw     a0, -12(fp)              # write local variable temp_29
  lw     t0, 0(fp)                # load static link 1
  lw     a0, 4(t0)                # read variable $exp.base
  lw     t1, -12(fp)              # read local variable temp_29
  mul    a0, a0, t1
  …
  jr     ra                       # Return to caller
```

Nested Functions: Calling Up

```python
def f(a: int) -> int:
  z : int = 17
  def g(b: int) -> int:
    def h(c: int) -> int:
      def i(d: int) -> int:
        print(d)
        if d == 1:
          return g(d - 1)
        else:
         return d
      print(c)
      return i(c - 1)
    print(b)
    if b == 0:
      return z
    else:
      return h(b - 1)
  print(a)
  return g(a - 1)

print(f(4))
```

Nested Functions: Calling Up

```
def f(a: int) -> int:
  z : int = 17
  def g(b: int) -> int:
    def h(c: int) -> int:
      def i(d: int) -> int:
        print(d)
        if d == 1:
          return g(d - 1)
        else:
          return d
      print(c)
      return i(c - 1)
    print(b)
    if b == 0:
      return z
    else:
      return h(b - 1)
  print(a)
  return g(a - 1)

print(f(4))
```

```
.globl $f.g
$f.g:
  addi    sp, sp, -@$f.g.size  # Reserve space for stack f
  sw      ra, @$f.g.size-4(sp) # Save return address
  sw      fp, @$f.g.size-8(sp) # Save control link (fp)
  addi    fp, sp, @$f.g.size   # New fp is at old SP.
  addi    sp, sp, -4           # allocate space for actual
  lw      a0, 4(fp)            # read formal parameter $f.
  sw      a0, 0(sp)            # push argument on stack
  jal     $printInt            # call function $printInt
  addi    sp, fp, -@$f.g.size  # restore stack pointer
  lw      a0, 4(fp)            # read formal parameter $f.
  li      t1, 0                # Load integer constant 0
  xor     a0, a0, t1           # Test integer equality
  seqz    a0, a0
  beqz    a0, false_28
  lw      t0, 0(fp)            # load static link 1
  lw      a0, -12(t0)          # read variable $f.z
  j       label_153
  j       end_28
false_28:
  addi    sp, sp, -8           # allocate space for actual
  mv      t0, fp               # load static link
  sw      t0, 0(sp)            # pass static link as param
  lw      a0, 4(fp)            # read formal parameter $f.
  li      t1, 1                # Load integer constant 1
  sub     a0, a0, t1           # Subtraction
  sw      a0, 4(sp)            # push argument on stack
  jal     $f.g.h               # call function $f.g.h
  addi    sp, fp, -@$f.g.size  # restore stack pointer
  …
  jr      ra                   # Return to caller
```

```
.globl $f.g.h.i
$f.g.h.i:
  addi    sp, sp, -@$f.g.h.i.size  # Reserve space for stack frame
  sw      ra, @$f.g.h.i.size-4(sp) # Save return address
  sw      fp, @$f.g.h.i.size-8(sp) # Save control link (fp)
  addi    fp, sp, @$f.g.h.i.size   # New fp is at old SP.
  addi    sp, sp, -4               # allocate space for actual argu
  lw      a0, 4(fp)                # read formal parameter $f.g.h.i
  sw      a0, 0(sp)                # push argument on stack
  jal     $printInt                # call function $printInt
  addi    sp, fp, -@$f.g.h.i.size  # restore stack pointer
  lw      a0, 4(fp)                # read formal parameter $f.g.h.i
  li      t1, 1                    # Load integer constant 1
  xor     a0, a0, t1               # Test integer equality
  seqz    a0, a0
  beqz    a0, false_29
  addi    sp, sp, -8               # allocate space for actual argu
  lw      t0, 0(fp)                # load static link 1
  lw      t0, 0(t0)                # load static link 2
  lw      t0, 0(t0)                # load static link 3
  sw      t0, 0(sp)                # pass static link as parameter
  lw      a0, 4(fp)                # read formal parameter $f.g.h.i
  li      t1, 1                    # Load integer constant 1
  sub     a0, a0, t1               # Subtraction
  sw      a0, 4(sp)                # push argument on stack
  jal     $f.g                     # call function $f.g
  addi    sp, fp, -@$f.g.h.i.size  # restore stack pointer
  j       label_155
  j       end_29
false_29:
  lw      a0, 4(fp)                # read formal parameter $f.g.h.i
  …
  jr      ra                       # Return to caller
```

identify call frame of function g

Nested Functions: Mutual Recursion

```
def pred(x: int) -> bool:
  true : bool = True
  false : bool = False

  def even(a : int) -> bool:
    if a == 0:
      return true
    else:
      return odd(a - 1)

  def odd(b : int) -> bool:
    if b == 0:
      return false
    else:
      return even(b - 1)

  return even(x)

print(pred(2))
```

what is the static link?

what is the static link?

Making Nesting Explicit

Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) -> int:
  b : int = 0

  def aux(i : int) -> int:
    return b + i

  def bar(y : int) -> int:
    c : int = 0

    def baz(z : int) -> int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```

how many static links should we follow to find a variable or (static link of) a function?

Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b + i

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b/1 + i/0

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux/2(c/1 + 1)
      return a/0 + x/2 + y/1 + z/0

    return baz/0(a/0 + b/1 + x/1)

  b = aux/0(x/0)
  return bar/0(b/0 + 10)

print(foo/0(a/0))
```

how many static links should we follow to find a variable or (static link of) a function?

difference between nesting level of occurrence and nesting level of definition

Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b + i

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b/1 + i/0

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux/2(c/1 + 1)
      return a/0 + x/2 + y/1 + z/0

    return baz/0(a/0 + b/1 + x/1)

  b = aux/0(x/0)
  return bar/0(b/0 + 10)

print(foo/0(a/0))
```

```
Return(
  AddInt(
    AddInt(
      AddInt(
        Var(("$a", 0))
      , Var(("$foo.x", 2))
      )
    , Var(("$foo.bar.y", 1))
    )
  , Var(("$foo.bar.baz.z", 0))
  )
)
```

how many static links should we follow to find a variable or (static link of) a function?

difference between nesting level of occurrence and nesting level of definition

transformation pairs levels with variables

Functions as First-Class Citizens

Challenge: Closures

Static link only works with nested functions

- the environment is still on the stack

Functions as first-class citizens

- `map((x: int) ⇒ x + 1, [1, 2, 3])`
- anonymous functions (lambdas)

Function values

- function value may escape the call frame in which it is created
- formal parameters + function body + values of free variables
- encoding in OO languages as objects with apply function

Challenge

- Extend ChocoPy with first-class functions

Statements

Statements

```
a : int = 3
b : int = 4

if a == b :
 a = 1
else:
 b = 2
```

```
main:
  …
  lw      a0, $a        # read global variable $a
  lw      t1, $b        # read global variable $b
  xor     a0, a0, t1    # Test integer equality
  seqz    a0, a0
  beqz    a0, false_30
  li      a0, 1         # Load integer constant 1
  sw      a0, $a, t0    # write global variable $a
  j       end_30
false_30:
  li      a0, 2         # Load integer constant 2
  sw      a0, $b, t0    # write global variable $b
  …
```

```
rules

  stat-to-instrs-(|r, regs) :
    IfElse(e, Block(stats1), Else(Block(stats2))) -> <concat>[
      …
    ]
    with <exp-to-instrs(|r, regs)> e => instrs0
    with <stats-to-instrs(|r, regs)> stats1 => instrs1
    with <stats-to-instrs(|r, regs)> stats2 => instrs2
```

String Constants

String Constants

```
message : str = "hello"
target : str = "world"

print(message + " " + target)
```

```
.globl temp_48
temp_48:
.word const_288

.globl temp_49
temp_49:
.word const_288

.globl $target
$target:
.word const_287

.globl $message
$message:
.word const_286
```

global variables

constant objects

```
.globl const_286
const_286:
.word 3
.word 6
.word $str$dispatchTable
.word 5
.string "hello"
.align 2

.globl const_287
const_287:
.word 3
.word 6
.word $str$dispatchTable
.word 5
.string "world"
.align 2

.globl const_288
const_288:
.word 3
.word 5
.word $str$dispatchTable
.word 0
.string ""
.align 2

.globl const_289
const_289:
.word 3
.word 5
.word $str$dispatchTable
.word 1
.string " "
.align 2
```

```
main:
  …
  addi   sp, sp, -8              # allocate space for actual arguments
  la     a0, const_289           # load string constant
  sw     a0, 0(sp)               # push argument on stack
  lw     a0, $message            # read global variable $message
  sw     a0, 4(sp)               # push argument on stack
  jal    strcat                  # call function strcat
  addi   sp, fp, -@..main.size   # restore stack pointer
  sw     a0, temp_49, t0         # write global variable temp_49
  addi   sp, sp, -8              # allocate space for actual arguments
  lw     a0, $target             # read global variable $target
  sw     a0, 0(sp)               # push argument on stack
  lw     a0, temp_49             # read global variable temp_49
  sw     a0, 4(sp)               # push argument on stack
  jal    strcat                  # call function strcat
  addi   sp, fp, -@..main.size   # restore stack pointer
  sw     a0, temp_48, t0         # write global variable temp_48
  addi   sp, sp, -4              # allocate space for actual arguments
  lw     a0, temp_48             # read global variable temp_48
  sw     a0, 0(sp)               # push argument on stack
  jal    $printString            # call function $printString
  addi   sp, fp, -@..main.size   # restore stack pointer
  …
```

loading string constants

Boxed vs Unboxed

Boxed vs Unboxed

String Values

- represented as objects with string as attribute

Integers and Booleans

- ChocoPy reference implementation:
  - represent as objects with value as attribute (= boxed)
- My implementation
  - unboxed representation of integers and booleans
  - where does this go wrong?

Execution Environment

Execution Environment: Built-In Functions

```
message : str = "hello"
target : str = "world"

print(message + " " + target)
```

string concatenation

```
.globl strcat
strcat:
  addi    sp, sp, -12
  sw      ra, 8(sp)
  sw      fp, 4(sp)
  addi    fp, sp, 12
  lw      t0, 4(fp)
  lw      t1, 0(fp)
  lw      t0, @.__len__(t0)
  beqz    t0, strcat_4
  lw      t1, @.__len__(t1)
  beqz    t1, strcat_5
  add     t1, t0, t1
  sw      t1, -12(fp)
  addi    t1, t1, 4
  srli    t1, t1, 2
  addi    a1, t1, @listHeaderWords
  la      a0, $str$prototype
  jal     alloc2
  lw      t0, -12(fp)
  sw      t0, @.__len__(a0)
  addi    t2, a0, 16
  lw      t0, 4(fp)
  lw      t1, @.__len__(t0)
  addi    t0, t0, @.__str__
```

printString : type specialized

```
.globl $printString
$printString:
  addi    sp, sp, -@printString.size
  sw      ra, @printString.size-4(sp)
  sw      fp, @printString.size-8(sp)
  addi    fp, sp, @printString.size
  lw      a1, 0(fp)
  addi    a1, a0, @.__str__
  li      a0, @print_string
  ecall
  li      a1, @newline
  li      a0, @print_char
  ecall
.equiv @printString.size, 8
  lw      ra, -4(fp)
  lw      fp, -8(fp)
  addi    sp, sp, @printString.size
  jr      ra
```

```
main:
  …
  addi    sp, sp, -8              # allocate space for actual arguments
  la      a0, const_281
  sw      a0, 0(sp)               # push argument on stack
  lw      a0, $message            # read global variable $message
  sw      a0, 4(sp)               # push argument on stack
  jal     strcat                  # call function strcat
  addi    sp, fp, -@..main.size   # restore stack pointer
  sw      a0, temp_46, t0         # write global variable temp_46
  addi    sp, sp, -8              # allocate space for actual arguments
  lw      a0, $target             # read global variable $target
  sw      a0, 0(sp)               # push argument on stack
  lw      a0, temp_46             # read global variable temp_46
  sw      a0, 4(sp)               # push argument on stack
  jal     strcat                  # call function strcat
  addi    sp, fp, -@..main.size   # restore stack pointer
  sw      a0, temp_45, t0         # write global variable temp_45
  addi    sp, sp, -4              # allocate space for actual arguments
  lw      a0, temp_45             # read global variable temp_45
  sw      a0, 0(sp)               # push argument on stack
  jal     $printString            # call function $printString
  addi    sp, fp, -@..main.size   # restore stack pointer
  …
```

get implementations from ChocoPy reference compiler

Memory Management

Garbage Collection

Reference counting

- deallocate records with count 0

Mark & sweep

- mark reachable records
- sweep unmarked records

Copying collection

- copy reachable records

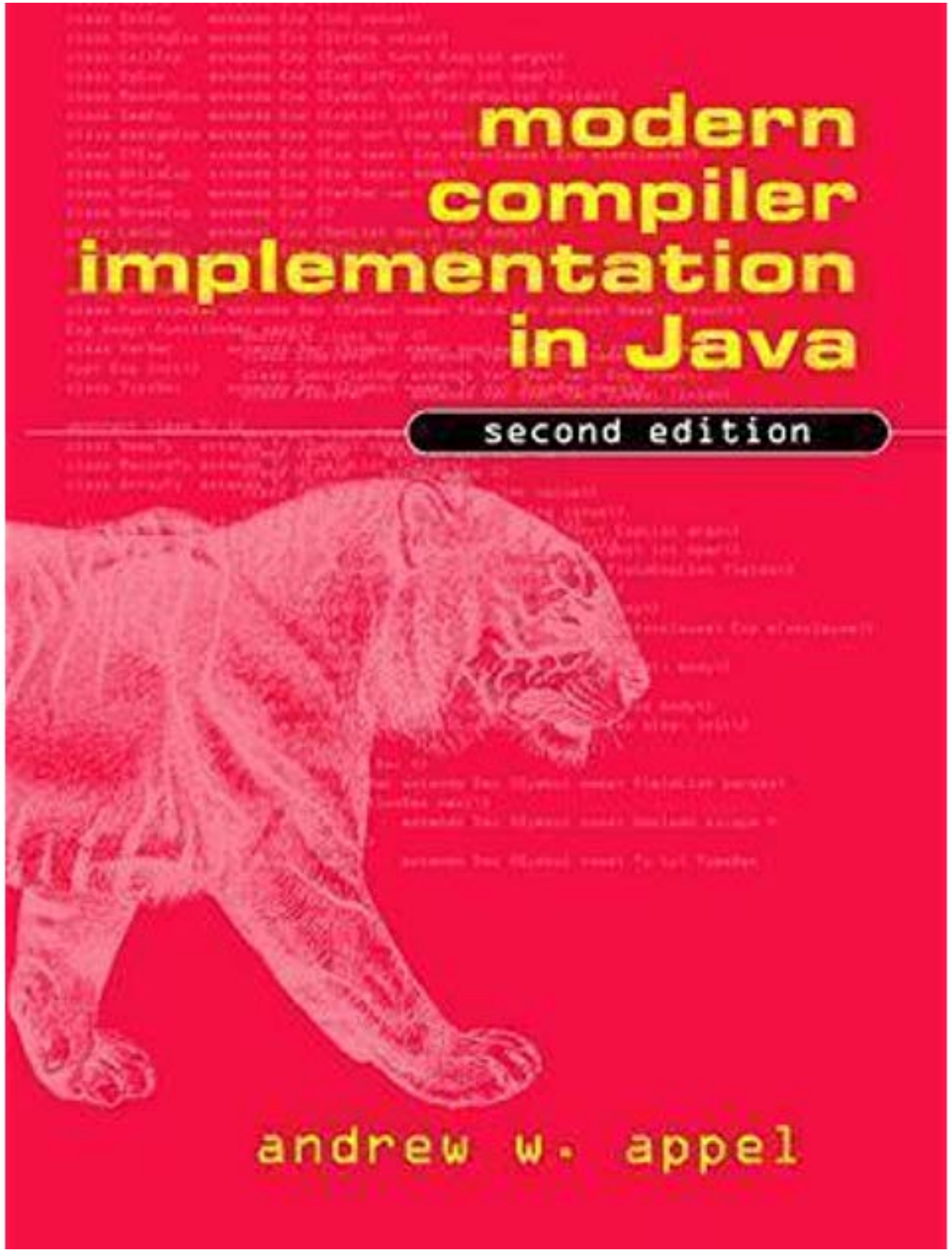
Generational collection

- collect only in young generations of records

Reading Material

Andrew W. Appel and Jens Palsberg (2002). Garbage Collection. Chapter In Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press.

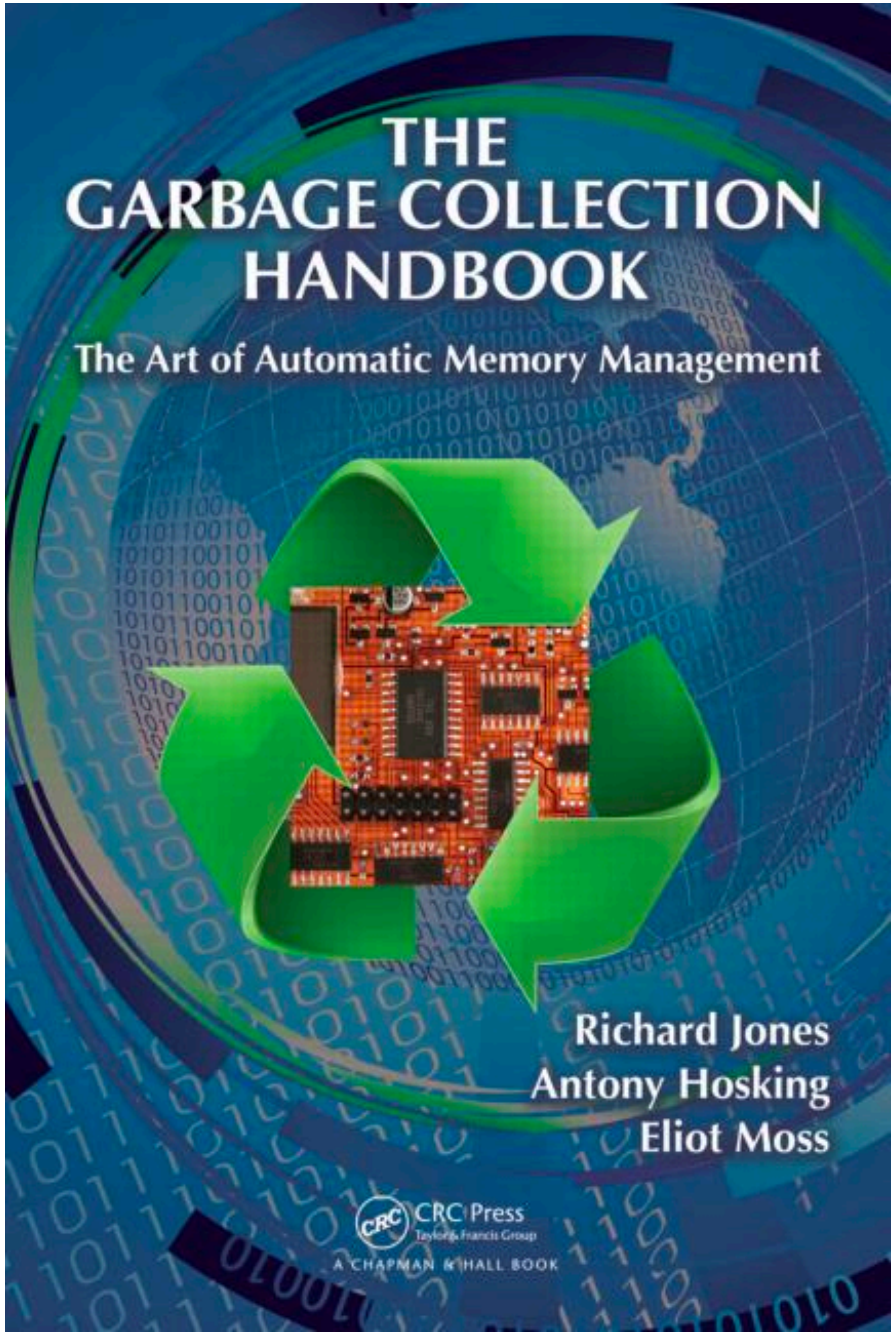
The lecture closely follows the discussion of mark-and-sweep collection, reference counts, copying collection, and generational collection in this chapter. This chapter also provides detailed cost analyses and discusses advantages and disadvantages of the different approaches to garbage collection.



Richard Jones, Antony Hosking, Eliot Moss. The Garbage Collection Handbook. The Art of Automatic Memory Management.

A systematic overview of garbage collection algorithms.

Dig deeper



Memory Safety &
Memory Management

Memory Safety

A program execution is memory safe if

- It only creates valid pointers through standard means
- Only uses a pointer to access memory that belongs to that pointer

Combines temporal safety and spatial safety

Spatial Safety

Access only to memory that pointer owns

View pointer as triple (p, b, e)

- p is the actual pointer
- b is the base of the memory region it may access
- e is the extent (bounds of that region)

Access allowed iff

- b <= p <= e - sizeof(typeof(p))

Allowed operations

- Pointer arithmetic increments p, leaves b and e alone
- Using &: e determined by size of original type

Temporal Safety

No access to undefined memory

Temporal safety violation: trying to access undefined memory

- Spatial safety assures it was to a legal region
- Temporal safety assures that region is still in play

Memory region is defined or undefined

Undefined memory is

- unallocated
- uninitialized
- deallocated (dangling pointers)

Memory Management

Manual memory management

- malloc, free in C
- Easy to accidentally free memory that is still in use
- Pointer arithmetic is unsafe

Automated memory management

- Spatial safety: references are opaque (no pointer arithmetic)
- (+ array bounds checking)
- Temporal safety: no dangling pointers (only free unreachable memory)

Garbage Collector

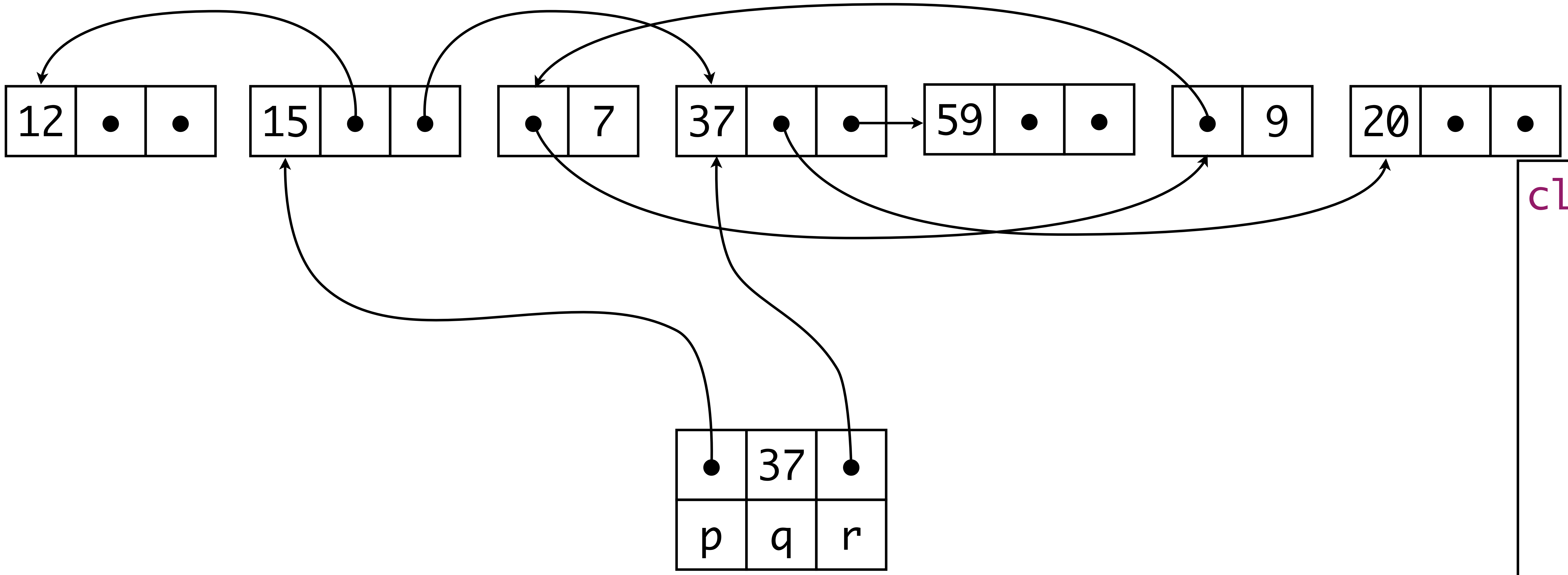
Terminology

- objects that are referenced are live
- objects that are not referenced are dead (garbage)
- objects are allocated on the heap

Responsibilities

- allocating memory
- ensuring live objects remain in memory
- garbage collection: recovering memory from dead objects

An Example Program



```
class List {
  List link;
  int key;
}

class Tree {
  int key;
  tree left;
  tree right;
}
```

```
class Main {
  static Tree makeTree() { … }
  static void showTree() { … }
  static void main() {
    {
      List x = new List(nil, 7);
      List y = new List(x, 9);
      x.link = y;
    }
    {
       Tree p = maketree();
       Tree r = p.right;
       int q = r.key;
       // garbage-collect here
       showtree(p)
    }
  }
}
```

Reference Counting

Reference Counting

Counts

- how many pointers point to each record?
- store count with each record

Counting

- extra instructions

Deallocate

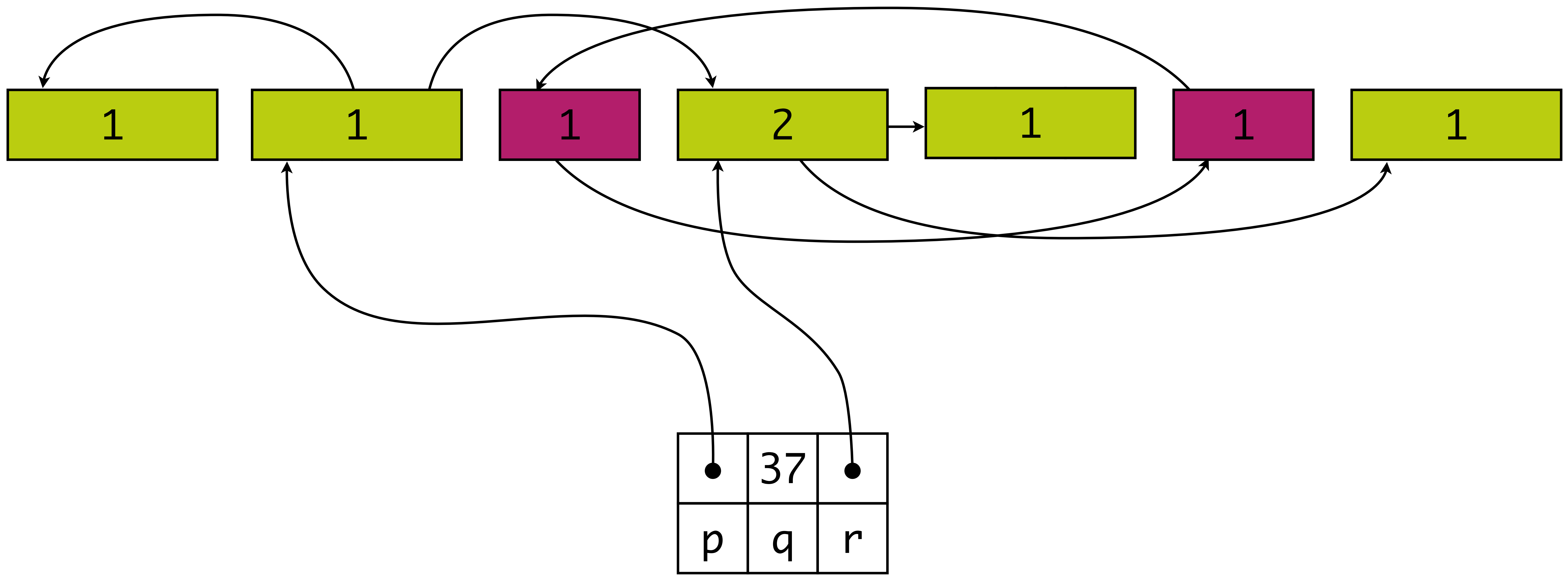
- put on freelist
- recursive deallocation on allocation

Reference Counting: Instrumentation

```
x.f := p
```

```
z        := x.f
c        := z.count
c        := c - 1
z.count  := c
if (c == 0) put z on free list
x.f      := p
c        := p.count
c        := c + 1
p.count  := c
```

Reference Counting



Reference Counting: Notes

Cycles

- memory leaks
- break cycles explicitly
- occasional mark & sweep collection

Expensive

- fetch, decrease, store old reference counter
- possible deallocation
- fetch, increase, store new reference counter

Programming Languages using Reference Counting

Languages with automatic reference counting

- Objective-C, Swift

Dealing with cycles

- strong reference: counts as a reference
- weak reference: can be nil, does not count
- unowned references: cannot be nil, does not count

Mark & Sweep

Mark & Sweep: Idea

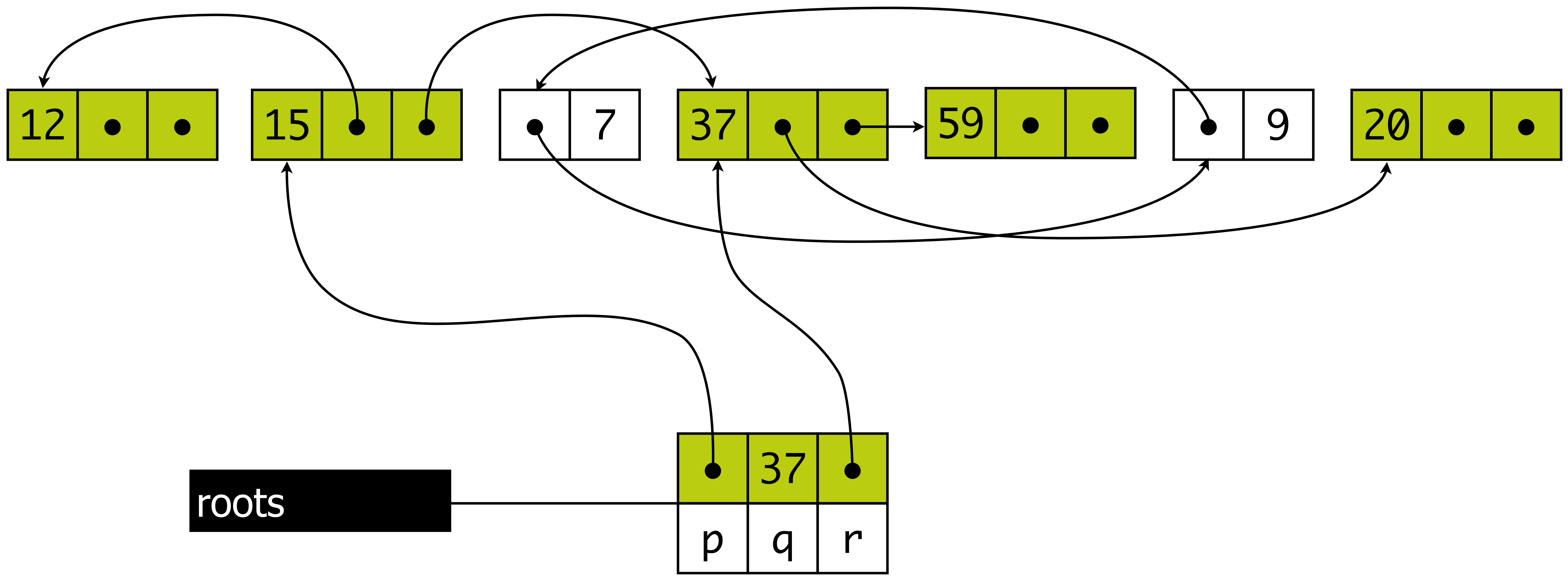
Mark

- mark reachable records
- start at variables (roots)
- follow references

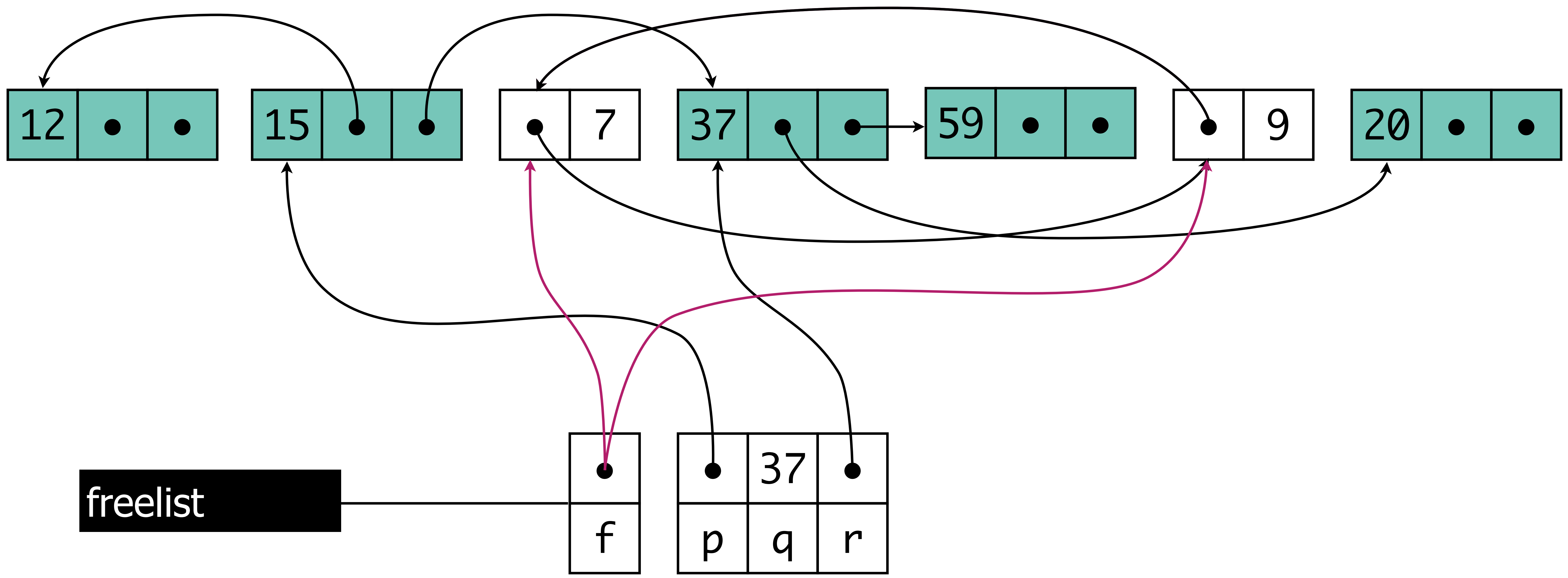
Sweep

- marked records: unmark
- unmarked records: deallocate
- linked list of free records

Marking



Sweeping



Mark & Sweep: Algorithms

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked := true

    foreach f in fields(x)

      DFS(f)
```

```
Sweep phase:

  p := first address in heap

  while p < last address in heap
    if p.marked

      p.marked := false

    else

      f1 := first field in p
      p.f1 := freelist
      free list := p

    p := p + sizeof( p )
```

Mark & Sweep: Costs

Instructions

- R reachable words in heap of size H
- Mark: c1 * R
- Sweep: c2 * H
- Reclaimed: H - R words
- Instructions per word reclaimed: (c1 * R + c2 * H) / (H - R)
- if (H >> R) cost per allocated word ~ c2

Mark & Sweep: Costs

Memory

- DFS is recursive
- maximum depth: longest path in graph of reachable data
- worst case: H
- | stack of activation records | > H

Measures

- explicit stack
- pointer reversal

Marking: DFS with Explicit Stack: Algorithms

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked = true
    t = 1 ; stack[t] = x

    while t > 0

      x = stack[t] ; t = t - 1

      foreach f in fields(x)
        if pointer(f) & !f.marked

          f.marked = true
          t = t + 1 ; stack[t] = f
```

Marking: DFS with Pointer Reversal

```
function DFS(x)
 if pointer(x) & x.done < 0
  x.done = 0 ; t = nil

  while true
   if x.done < x.fields.size
    y = x.fields[x.done]
    if pointer(y) & y.done < 0
     x.fields[x.done] = t ; t = x ; x = y ; x.done = 0
    else
     x.done = x.done + 1

   else
    y = x; x = t
    if t = nil then return
    t = x.fields[x.done]; x.fields[x.done] = y
    x.done = x.done + 1
```

marking without memory overhead

Mark & Sweep

Sweeping

- independent of marking algorithm
- several freelists (per record size)
- split free records for allocation

Fragmentation

- external: many free records of small size
- internal: too-large record with unused memory inside

Copying Collection

Copying Collection: Idea

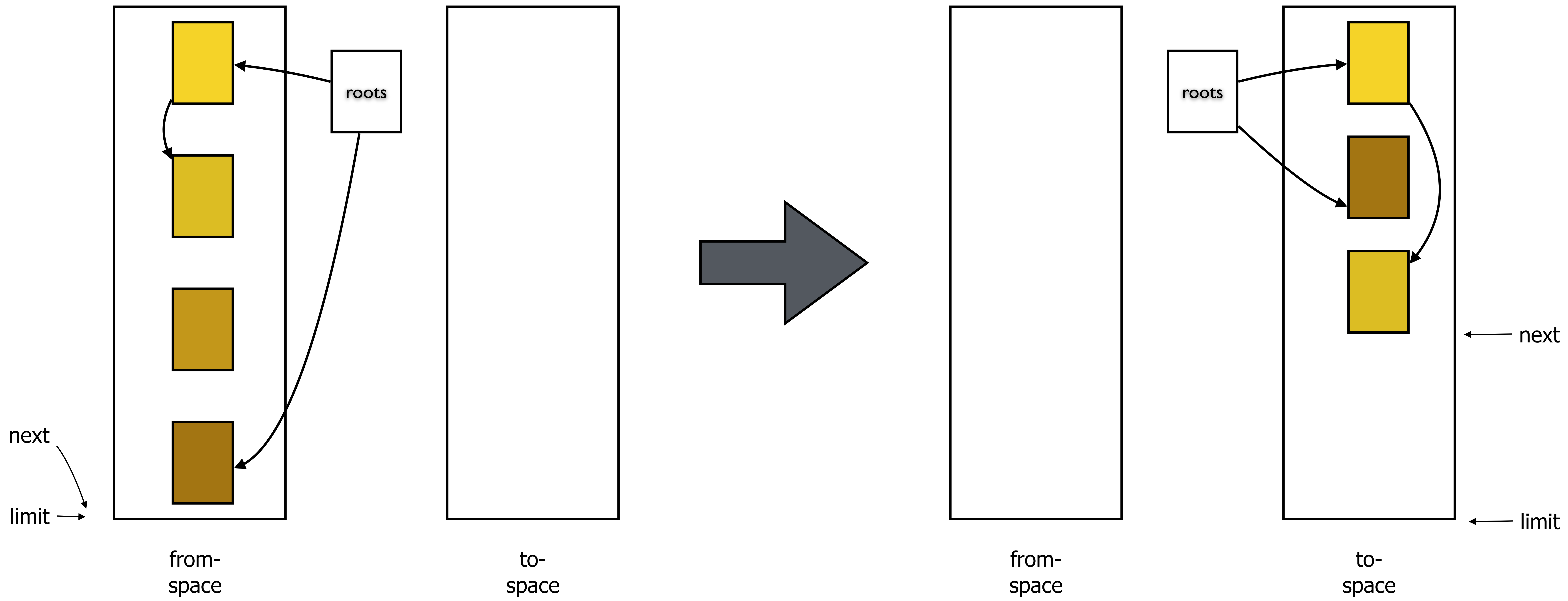
Spaces

- fromspace & tospace
- switch roles after copy

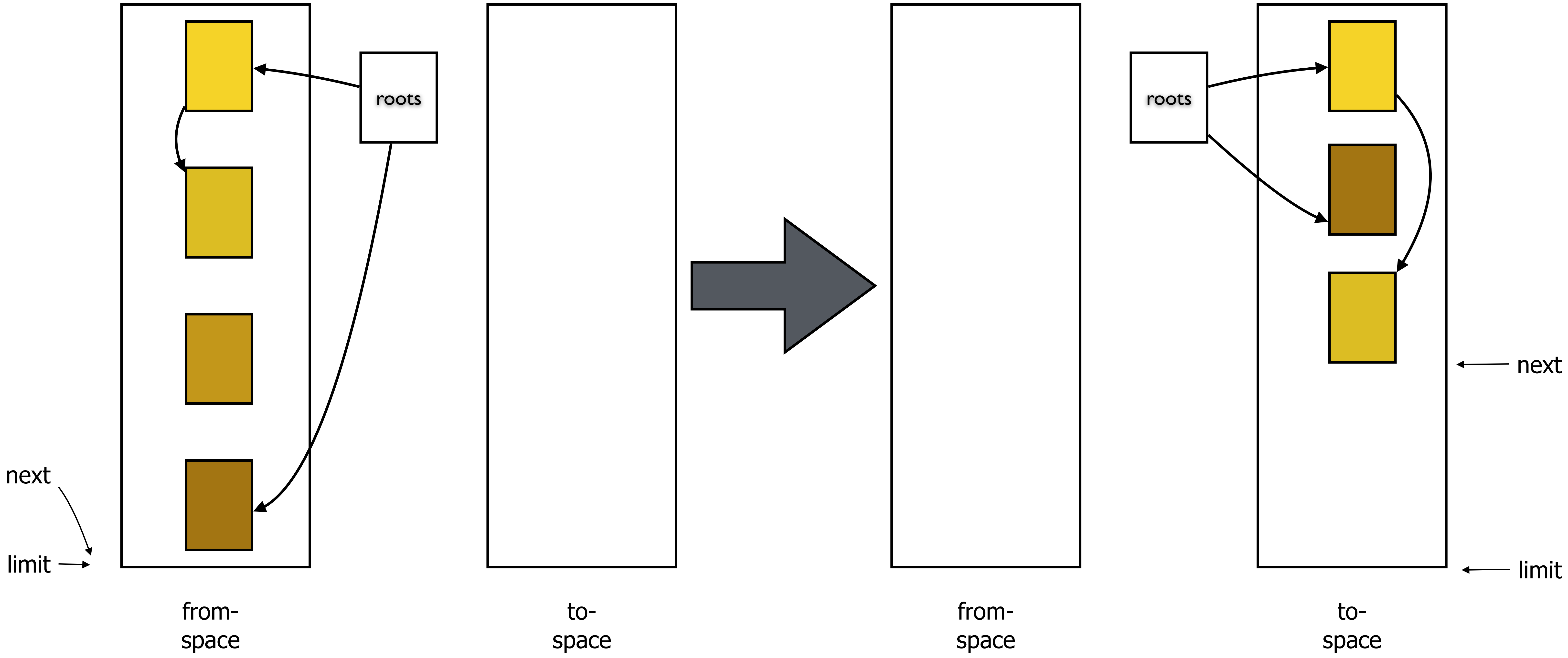
Copy

- traverse reachability graph
- copy from fromspace to tospace
- fromspace unreachable, free memory
- tospace compact, **no fragmentation**

Copying Collection: Idea

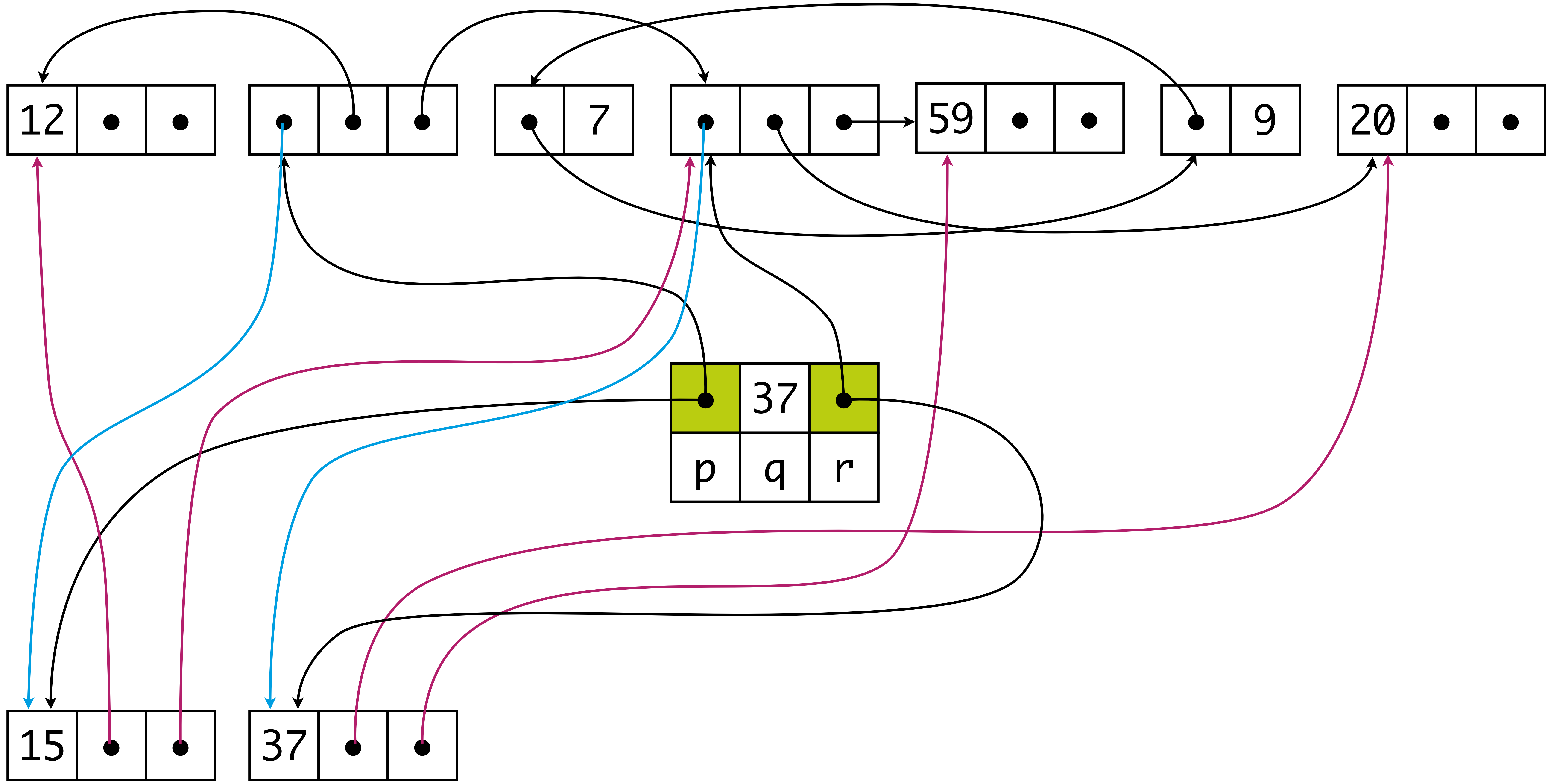


Copying Collection: Algorithm

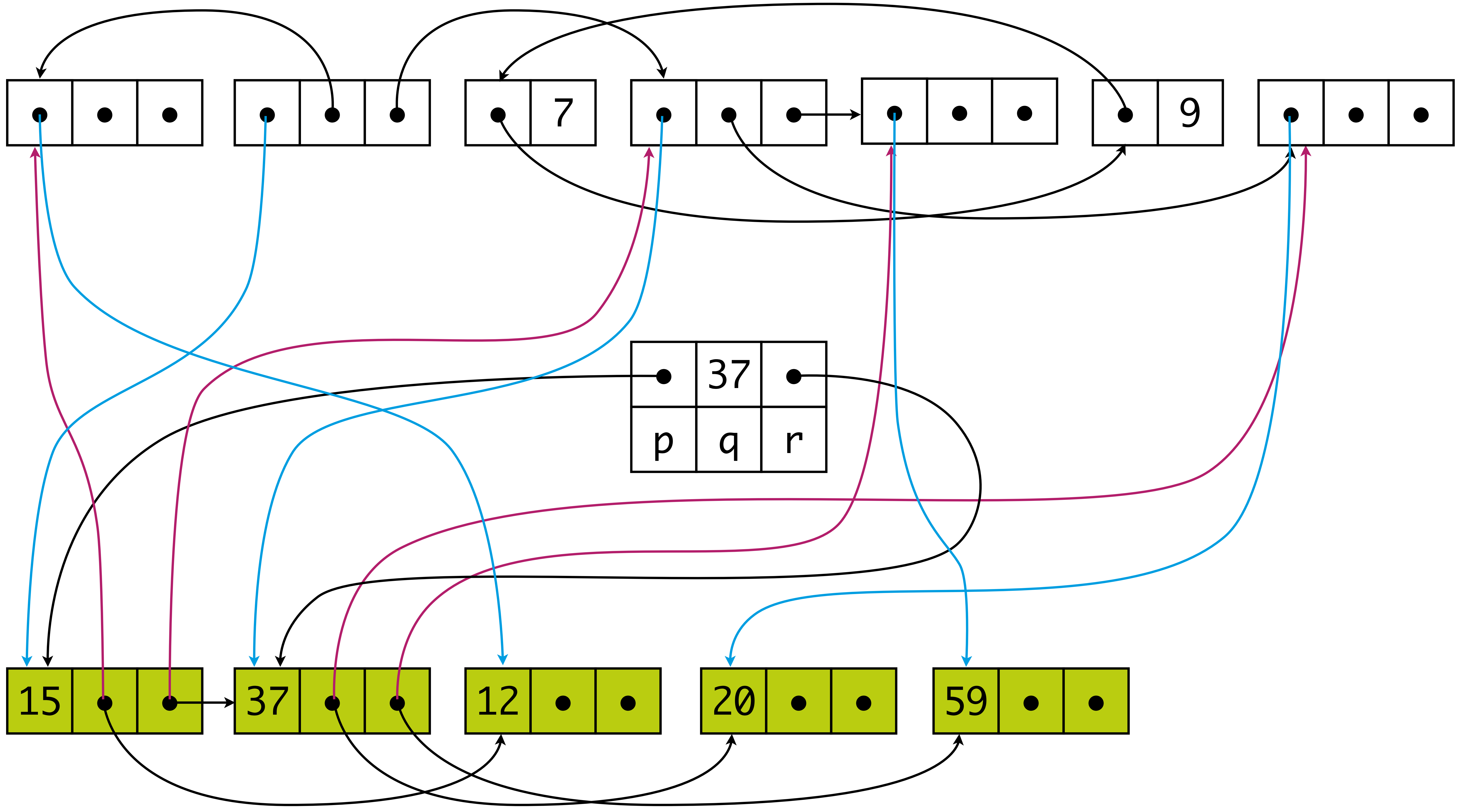


```
function BFS()

  next := scan := start(tospace)

  foreach r in roots
    r = Forward(r)

  while scan < next

    foreach f in fields of scan
      scan.f = Forward(scan.f)

    scan = scan + sizeof(scan)
```

Copying Collection: Example



Copying Collection: Example



Copying Collection: Issues

Adjacent records

- likely to be unrelated

Pointers to records in records

- likely to be accessed
- likely to be far apart

Solution

- depth-first copy: slow pointer reversals
- hybrid copy algorithm

Copying Collection: Costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- Reclaimed: H/2 - R words
- Instructions per word reclaimed: (c3 * R) / (H/2 - R)
- If (H >> R) : cost per allocated word => 0
- If (H = 4R) : c3 instructions per word allocated
- Solution: reduce portion of R to inspect => generational collection

Generational Collection

Generational Collection

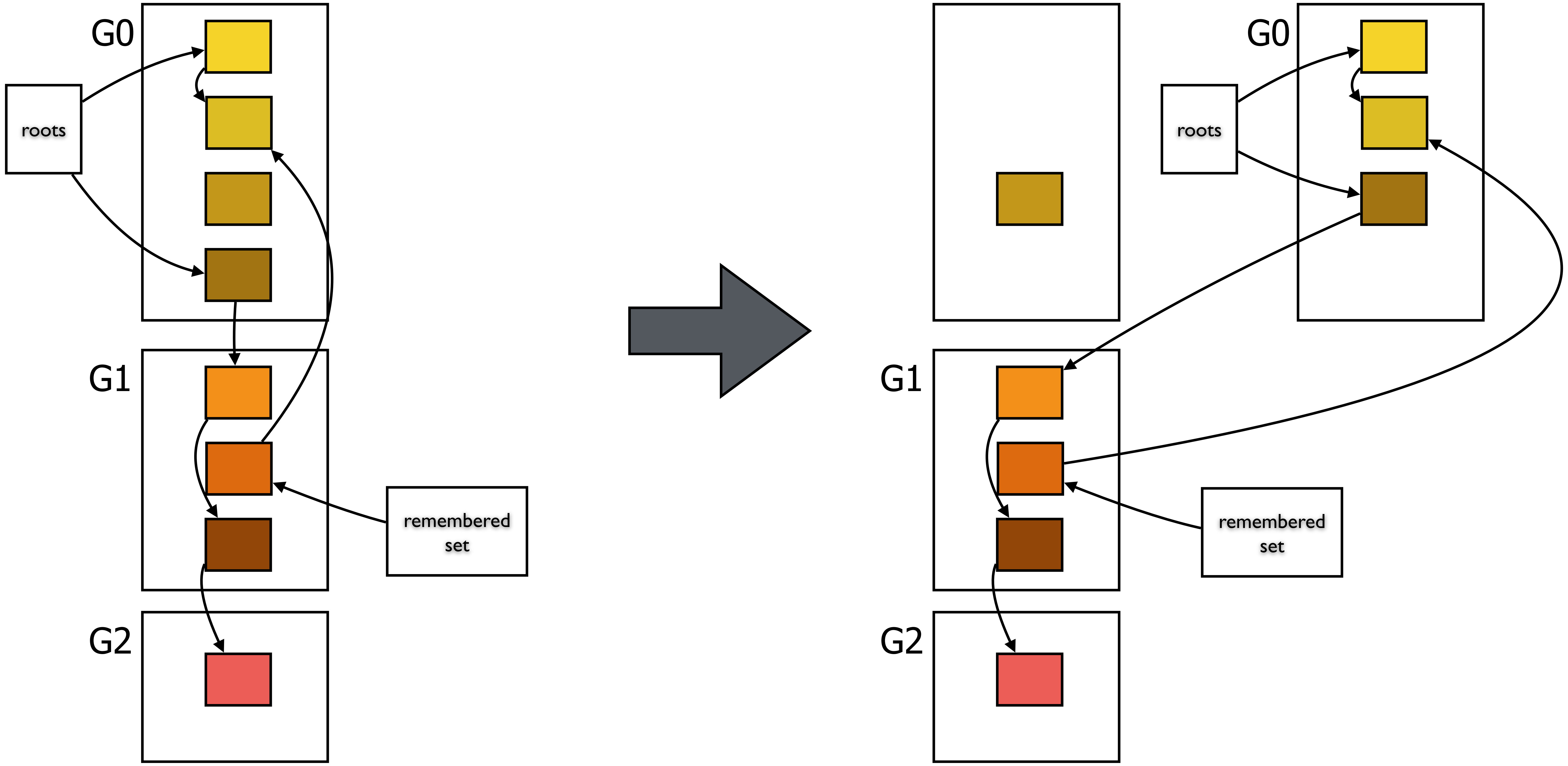
Generations

- young data: likely to die soon
- old data: likely to survive for more collections
- divide heap, collect younger generations more frequently

Collection

- roots: variables & pointers from older to younger generations
- preserve pointers to old generations
- promote objects to older generations

Generational Collection



Generational Collection: Costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- 10% of youngest generation is live: H/R = 10
- Instructions per word reclaimed:
  (c3 * R) / (H - R) = (c3 * R) / (10R - R) ~= c3/10
- Adding to remembered set: 10 instructions per update

Incremental Collection

Interrupt by garbage collector undesirable

- interactive, real-time programs

Incremental / concurrent garbage collection

- interleave collector and mutator (program)
- incremental: per request of mutator
- concurrent: in between mutator operations

Tricolor marking

- White: not visited
- Grey: visited (marked or copied), children not visited
- Black: object and children marked

Summary

Algorithms

How can we collect unreachable records on the heap?

- reference counts
- mark reachable records, sweep unreachable records
- copy reachable records

How can we reduce heap space needed for garbage collection?

- pointer-reversal
- breadth-first search
- hybrid algorithms

Design Choices

Serial vs Parallel

- garbage collection as sequential or parallel process

Concurrent vs Stop-the-World

- concurrently with application or stop application

Compacting vs Non-compacting vs Copying

- compact collected space
- free list contains non-compacted chunks
- copy live objects to new space; from-space is non-fragmented

Performance Metrics

Throughput

– percentage of time not spent in garbage collection

GC overhead

– percentage of time spent in garbage collection

Pause time

– length of time execution is stopped during garbage collection

Frequency of collection

– how often collection occurs

Footprint

– measure of (heap) size

Garbage Collection in Java HotSpot VM

Serial collector

- young generation: copying collection
- old generation: mark-sweep-compact collection

Parallel collector

- young generation: stop-the-world copying collection in parallel
- old generation: same as serial

Parallel compacting collector

- young generation: same as parallel
- old generation: roots divided in threads, marking live objects in parallel, …

Concurrent Mark-Sweep (CMS) collector

- stop-the-world initial marking and re-marking
- concurrent marking and sweeping

Literature

Literature

- Andrew W. Appel, Jens Palsberg. Modern Compiler Implementation in Java, 2nd edition, 2002.
- Sun Microsystems. Memory Management in the Java HotSpotTM Virtual Machine, April 2006.
- Richard Jones, Antony Hosking, Eliot Moss. The Garbage Collection Handbook. The Art of Automatic Memory Management.

Language-Parametric Memory Management?

Language-Parametric Memory Management?

Garbage collectors are language-specific

- Representation of objects in memory
- Roots of heap in stack

Can we derive garbage collector from language definition?

A uniform model for memory layout

- Scopes describe static binding structure
- Frames instantiate scopes at run time
- Language-parametric memory management
- Language-parametric type safety

Language-Parametric Type Safety?

Type Safety: Well-typed programs don't go wrong

- A program that type checks does not have run-time type errors
- Preservation
  - `e : t & e -> v => v : t`
- Progress
  - `e -> e'  =>  e' is a value || e' -> e''`
- (Slightly different for big step semantics as in definitional interpreters)

Proving type safety

- Easier to establish with an interpreter
- Bindings complicate proof
- How to maintain?
- Can we automate verification of type safety?

Traditionally, operational semantics specifications use ad hoc mechanisms for representing the binding structures of programming languages.

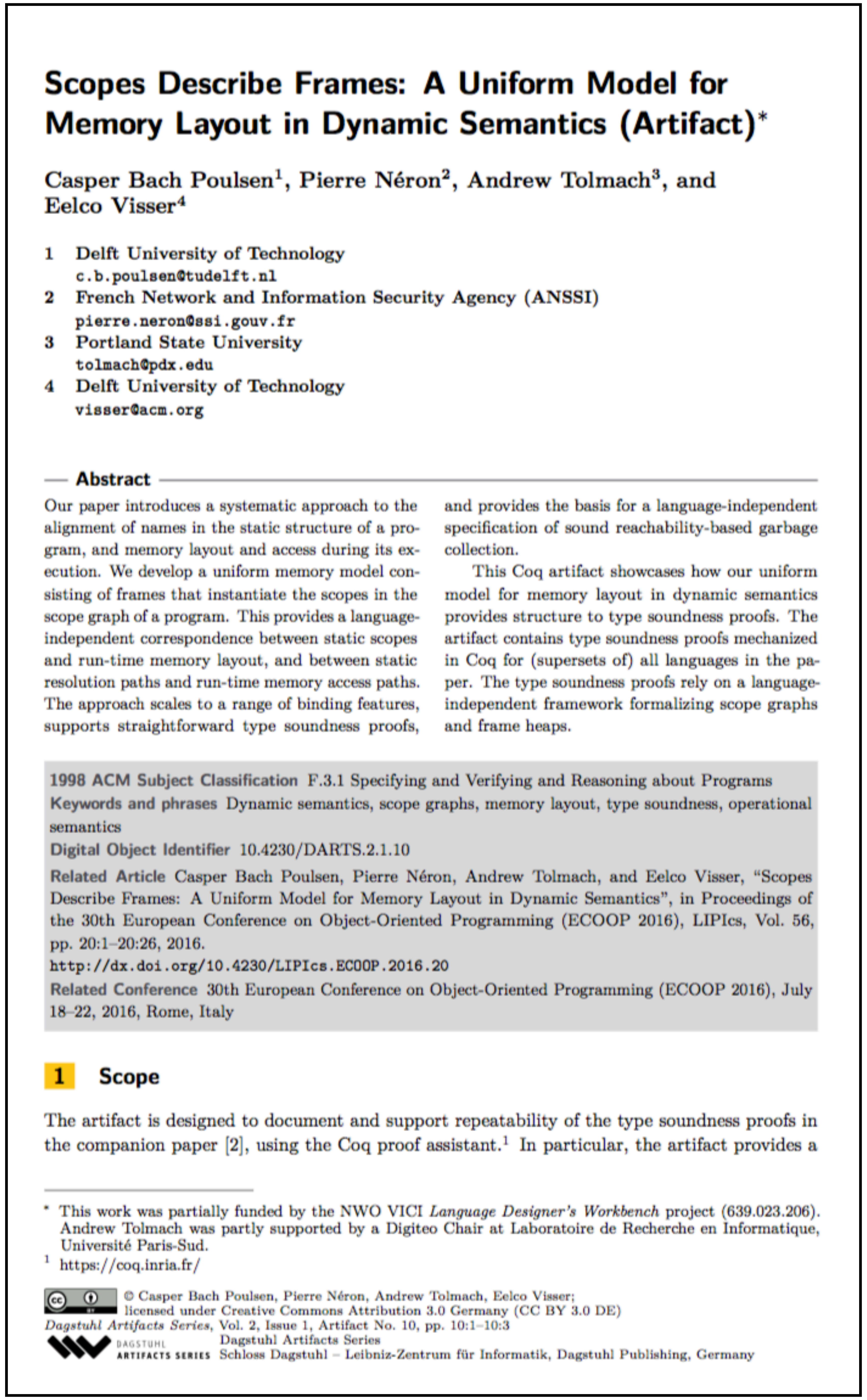
This paper introduces frames as the dynamic counterpart of scopes in scope graphs.

This provides a uniform model for the representation of memory at run-time.

We are currently experimenting with specializing DynSem interpreters using scopes and frames using Truffle/Graal with encouraging results (200x speed-ups).

ECOOP 2016

http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20



Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics (Artifact)*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[3], and Eelco Visser[4]

1 Delft University of Technology
c.b.poulsen@tudelft.nl
2 French Network and Information Security Agency (ANSSI)
pierre.neron@ssi.gouv.fr
3 Portland State University
tolmach@pdx.edu
4 Delft University of Technology
visser@acm.org

Abstract

Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics

Digital Object Identifier 10.4230/DARTS.2.1.10

Related Article Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser, "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics", in Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LIPIcs, Vol. 56, pp. 20:1–20:26, 2016.
http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20

Related Conference 30th European Conference on Object-Oriented Programming (ECOOP 2016), July 18–22, 2016, Rome, Italy

1 Scope

The artifact is designed to document and support repeatability of the type soundness proofs in the companion paper [2], using the Coq proof assistant.[1] In particular, the artifact provides a

---

* This work was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

[1] https://coq.inria.fr/

Dagstuhl Artifacts Series, Vol. 2, Issue 1, Artifact No. 10, pp. 10:1–10:3
Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Specializing a Meta-Interpreter

JIT Compilation of DynSem Specifications on the Graal VM

Vlad Vergu
TU Delft
The Netherlands
v.a.vergu@tudelft.nl

Eelco Visser
TU Delft
The Netherlands
visser@acm.org

ABSTRACT

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. DynSem specifications can be executed to interpret programs in the language under development. To enable fast turnaround during language development, we have developed a meta-interpreter for DynSem specifications, which requires minimal processing of the specification. In addition to fast development time, we also aim to achieve fast run times for interpreted programs.

In this paper we present the design of a meta-interpreter for DynSem and report on experiments with JIT compiling the application of the meta-interpreter on the Graal VM. By interpreting specifications directly, we have minimal compilation overhead. By specializing pattern matches, maintaining call-site dispatch chains and using native control-flow constructs we gain significant run-time performance. We evaluate the performance of the meta-interpreter when applied to the Tiger language specification running a set of common benchmark programs. Specialization enables the Graal VM to JIT compile the meta-interpreter giving speedups of up to factor 15 over running on the standard Oracle Java VM.

CCS CONCEPTS

• **Software and its engineering → Interpreters**; **Domain specific languages**; *Semantics*;

KEYWORDS

dynamic semantics, interpretation, JIT, run-time optimization

ACM Reference Format:
Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3237009.3237018

1 INTRODUCTION

The dynamic semantics of a programming language defines the run time execution behavior of programs in the language. Ideally, the design of a programming language starts with the specification of its dynamic semantics to provide a high-level readable and unambiguous definition. However, understanding the design of a programming language also requires experimentation by actually running programs. Therefore, this ideal route is rarely taken, but language designs are embodied in the implementation of interpreters or compilers instead.

We have previously designed DynSem [33], a high-level meta-DSL for dynamic semantics specifications of programming languages, with the aim of supporting readable *and* executable specification. It supports the definition of modular and concise semantics by means of reduction rules with implicit propagation of contextual information. DynSem's executable semantics entails that specifications can be used to interpret object language programs.

In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating a Java implementation of an interpreter and compiling that generated code caused long turnaround times during language prototyping. In order to support rapid prototyping with short turnaround times, we turned to interpreting specifications directly instead of compiling them. A DynSem interpreter is a *meta-interpreter* since the programs it interprets are themselves interpreters. Figure 1 depicts the high-level architecture of the DynSem meta-interpreter. First, a DynSem specification is desugared (explicated) to make implicit passing of semantic components explicit. The resulting specification in DynSem Core is then loaded into the meta-interpreter together with the AST of the interpreted object program. The interpreter consumes the program as input enacting the specification. This produces the desired result of a short turnaround time for experimenting with dynamic semantics specifications.

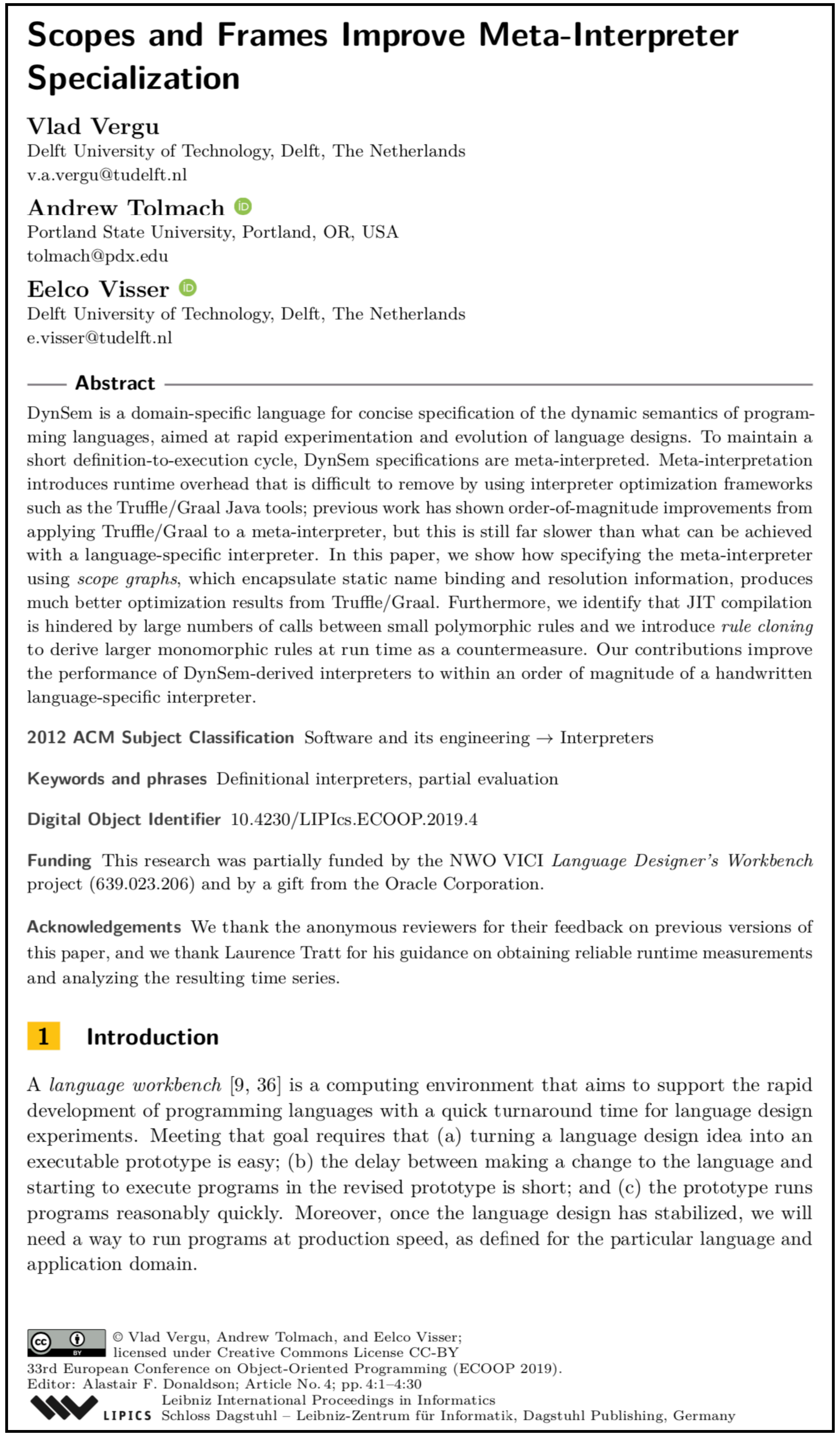
Meta-interpretation reduces the turnaround time at the expense of execution performance. At run time there are two interpreter layers operating (the meta-language interpreter and the object-language interpreter) which introduces substantial overhead. While we envision DynSem as a convenient way to prototype the dynamic semantics of programming languages, ultimately we also envision it as a convenient way to bridge the gap between the prototyping and production phases of a programming language's lifecycle. Thus, we not only want an interpreter fast, but we also want a fast interpreter, which raises the question: Can we achieve fast object-language interpreters by optimizing the meta-interpretation of dynamic semantics specifications?

Direct vanilla interpreters are in general slow to begin with, even when they are implemented in a host language that is JIT-ed. This is because the host JIT is unable to see patterns in the object language and to meaningfully optimize the interpreter. The task of optimizing an interpreter has traditionally been long and

---

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237018



Scopes and Frames Improve Meta-Interpreter Specialization

Vlad Vergu
Delft University of Technology, Delft, The Netherlands
v.a.vergu@tudelft.nl

Andrew Tolmach
Portland State University, Portland, OR, USA
tolmach@pdx.edu

Eelco Visser
Delft University of Technology, Delft, The Netherlands
e.visser@tudelft.nl

Abstract

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. To maintain a short definition-to-execution cycle, DynSem specifications are meta-interpreted. Meta-interpretation introduces runtime overhead that is difficult to remove by using interpreter optimization frameworks such as the Truffle/Graal Java tools; previous work has shown order-of-magnitude improvements from applying Truffle/Graal to a meta-interpreter, but this is still far slower than what can be achieved with a language-specific interpreter. In this paper, we show how specifying the meta-interpreter using *scope graphs*, which encapsulate static name binding and resolution information, produces much better optimization results from Truffle/Graal. Furthermore, we identify that JIT compilation is hindered by large numbers of calls between small polymorphic rules and we introduce *rule cloning* to derive larger monomorphic rules at run time as a countermeasure. Our contributions improve the performance of DynSem-derived interpreters to within an order of magnitude of a handwritten language-specific interpreter.

2012 ACM Subject Classification Software and its engineering → Interpreters

Keywords and phrases Definitional interpreters, partial evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.4

Funding This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206) and by a gift from the Oracle Corporation.

Acknowledgements We thank the anonymous reviewers for their feedback on previous versions of this paper, and we thank Laurence Tratt for his guidance on obtaining reliable runtime measurements and analyzing the resulting time series.

1 Introduction

A *language workbench* [9, 36] is a computing environment that aims to support the rapid development of programming languages with a quick turnaround time for language design experiments. Meeting that goal requires that (a) turning a language design idea into an executable prototype is easy; (b) the delay between making a change to the language and starting to execute programs in the revised prototype is short; and (c) the prototype runs programs reasonably quickly. Moreover, once the language design has stabilized, we will need a way to run programs at production speed, as defined for the particular language and application domain.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 4; pp. 4:1–4:30
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A desirable property for programming languages is type safety: well-typed programs don’t go wrong.

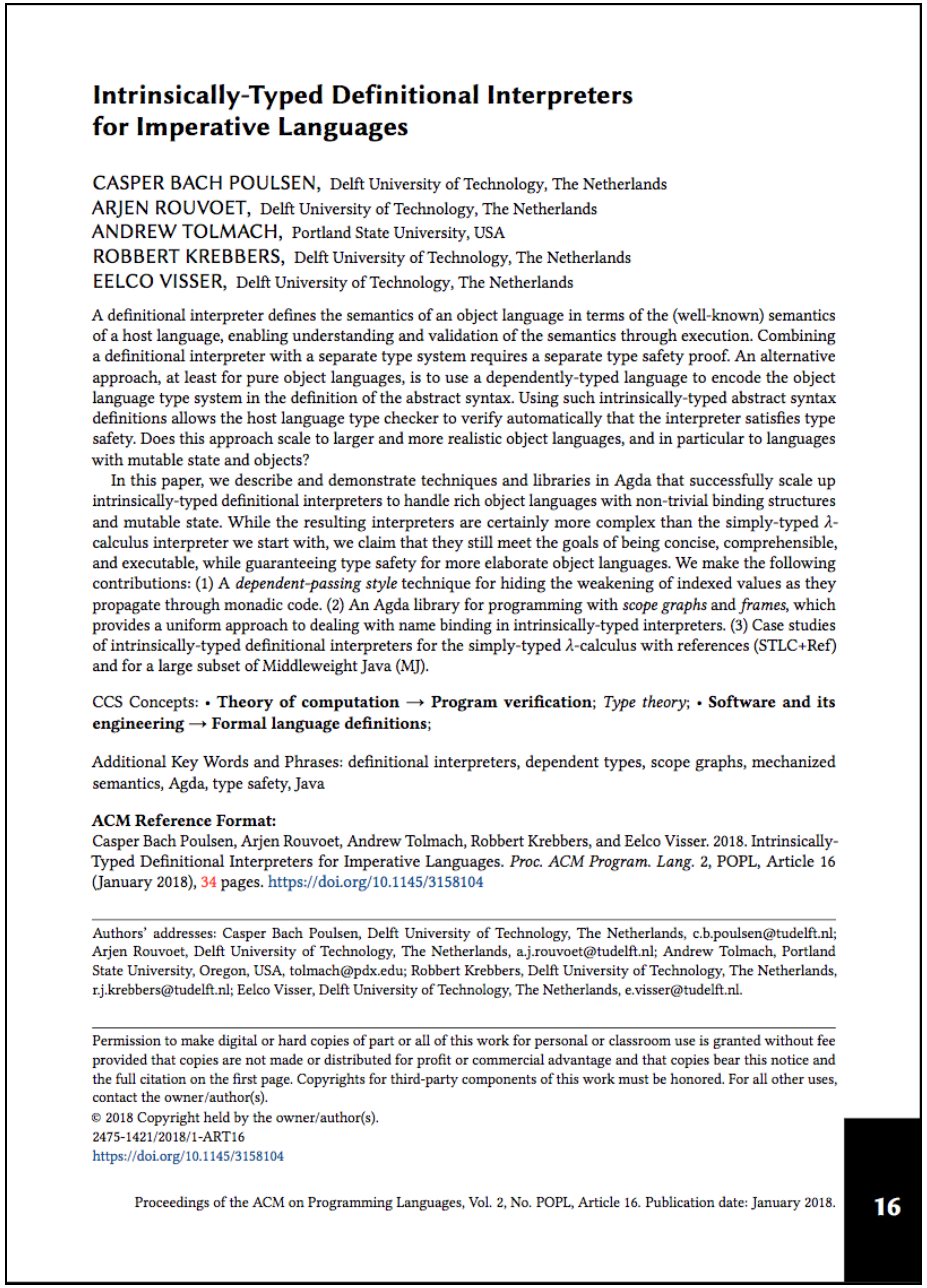
Demonstrating type safety for language implementations requires a proof. Such a proof is hard (at least tedious) for language models, and rarely done for language implementations.

Can we automatically check type safety for language implementations?

This paper shows how to do that at least for definitional interpreters for non-trivial languages. (By using scopes and frames to represent bindings.)

POPL 2018

https://doi.org/10.1145/3158104



Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
ANDREW TOLMACH, Portland State University, USA
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed λ-calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed λ-calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation → Program verification**; *Type theory*; • **Software and its engineering → Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

ACM Reference Format:
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. https://doi.org/10.1145/3158104

Authors’ addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Andrew Tolmach, Portland State University, Oregon, USA, tolmach@pdx.edu; Robbert Krebbers, Delft University of Technology, The Netherlands, r.j.krebbers@tudelft.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

2475-1421/2018/1-ART16
https://doi.org/10.1145/3158104