Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

Lecture 3
Memory management

Johan Nordlander

Anonymous functions revisited

- Our latest expression grammar:

Calling one → x es

Creating one → f

e ::= x | x es | f | f es | lit | K

Creating one if arguments are too few → x es, f es

- Must be supported – not a functional language otherwise!
- Requires the concept of <u>closures</u>!

Closures

- <u>The</u> generic representation of functions: a function pointer with a list of free variables
- The limits of lambda-lifting:

```
g = \a  -> let f = \b -> a + b            f = \a b -> a + b
           in h f                  ⇨      g = \a -> h (f a)
h = \x -> x 7                              h = \x -> x 7
```

Too few arguments regardless!

- Closures can represent partial applications, even in the presence of free variables
- Nevertheless, lambda-lifting before closure-conversion simplifies the presentation somewhat

Closure-conversion

- Assume a lambda-lifted $f = \backslash x_1 \ldots x_n \rightarrow e$

closureConvert f = $CL\ f_0\ n$

closureConvert $(f\ e_1 \ldots e_m)$ =

  | $m < n$ = $CL\ f_m\ (n\text{-}m)\ e_1 \ldots e_m$

  ...

  where f_m is a new top-level function

    $f_m = \backslash x_{this}\ x_{m+1} \ldots x_n \rightarrow$ **case** x_{this} **of**

      $CL\ \_\ \_\ y_1 \ldots y_m \rightarrow f\ y_1 \ldots y_m\ x_{m+1} \ldots x_n$

closureConvert $(x\ e_1 \ldots e_m)$ = **case** x **of** $CL\ f_{unknown}\ n$

  | $m == n \rightarrow f_{unknown}\ x\ e_1 \ldots e_m$

  ...

Closure-conversion

- Example before and after lambda-lifting:

```
g = \a  -> let f = \b -> a + b
              in f g
h = \x -> x 7
```

⇒

```
f = \a b -> a + b
g = \a -> h (f a)
h = \x -> x 7
```

- And after closure-conversion:

```
f = \a b -> a + b
g = \a -> CL f_1 1 a
h = \x -> case x of CL f_unknown 1 -> f_unknown x 7

f_1 = \x_this x_2 -> case x_this of CL _ _ y_1 -> f y_1 x_2
```

- But we're still ignoring arity mismtaches...

Checking arities

(eval/apply)

- Assuming $f = \backslash x_1 \ldots x_n \to e$

  closureConvert f = $CL\ f_0\ n$

  closureConvert $(f\ e_1 \ldots e_m)$ =
  $\quad |\ m == n \quad = \quad f\ e_1 \ldots e_m$
  $\quad |\ m < n \quad = \quad CL\ f_m\ (n-m)\ e_1 \ldots e_m$
  $\quad |\ m > n \quad = \quad apply_{m-n}\ (f\ e_1 \ldots e_n)\ e_{n+1} \ldots e_m$

  where each $apply_k$ is a run-time system function TBD

- Note: checks are done at <u>compile-time</u>

Checking arities

(eval/apply)

- The full dynamic case (checks at run-time!):

closureConvert ($x\ e_1 \ldots e_m$) = $apply_m\ x\ e_1 \ldots e_m$

$apply_m$ = $\backslash x_{this}\ x_1 \ldots x_m$ -> **case** x_{this} **of** CL $f_{unknown}$ n
| m == n -> $f_{unknown}\ x_{this}\ x_1 \ldots x_m$
| m < n -> CL $pap_{n-m,m}$ (n-m) $x_{this}\ x_1 \ldots x_m$
| m > n -> $apply_{m-n}$ ($f_{unknown}\ x_{this}\ x_1 \ldots x_n$) $x_{n+1} \ldots x_m$

$pap_{k,m}$ = $\backslash x_{this}\ x_1 \ldots x_k$ -> **case** x_{this} **of** CL _ _ $y_{that}\ y_1 \ldots y_m$ ->
$apply_{m+k}\ y_{that}\ y_1 \ldots y_m\ x_1 \ldots x_k$

Recall: data layout

```
typedef int *Ptr;
```

Basic assumptions:

(Ptr)(int)x = x   (int)(Ptr)y = y

Construction:

$x = K_i\ e_1 \ldots e_n$

```
Ptr x = malloc((n+1)*sizeof(int));
x[0] = i;
x[1] = (int)e1; ...; x[n] = (int)en;
```

Deconstruction:

case x **of**
  ....
  $K_i\ x_1 \ldots x_n \to body_i$

```
switch (x[0]) {
  ...
  case i: { Ptr x1 = (Ptr)x[1]; ...
            Ptr xn = (Ptr)x[n];
            bodyi }
```

Nullary constructors

Could just use the generic form:

```
x = Ki                    Ptr x = malloc(sizeof(int));
                          x[0] = i;
case x of                 switch (x[0]) {
  ...                       ...
  Ki -> body                case i: { body }
```

For better memory efficiency, encode as <u>small pointer</u>:

```
Ki                        (Ptr) i
case x of                 switch ((int)x) {
  Ki -> bodyi               case i: { bodyi }
  ...                       ...
  Kj x1 ... xn -> bodyj     default: switch (x[0]) {
                                case j: { Ptr x1 = (Ptr)x[1]; ...
                                          Ptr xn = (Ptr)x[n];
                                          bodyj }
```

Single constructors

Could just use the generic form:

$x = K_0\ e_1 \ldots e_n$	`Ptr x = malloc((n+1)*sizeof(int));`
	`x[0] = 0;`
	$x[1] = (int)e_1; \ldots\ x\text{->}arg[n] = (int)e_n;$

case x **of** $K_0\ x_1 \ldots x_n \text{ -> } body_i$	**switch** (x[0]) { **case** 0: { $Ptr\ x_1 = (Ptr)x[1]; \ldots$ $Ptr\ x_n = (Ptr)x[n];$ $body_0$ }

For better efficiency, encode <u>without a tag</u>:

$x = K_0\ e_1 \ldots e_n$	`Ptr x = malloc(n*sizeof(int));` $x[0] = (int)e_1; \ldots$ $x[n-1] = (int)e_n;$
case x **of** $K_0\ x_1 \ldots x_n \text{ -> } body_0$	$Ptr\ x_1 = (Ptr)x[0]; \ldots$ $Ptr\ x_n = (Ptr)x[n-1];$ $body_0$

Global data

- Declarations on the top level:

  a) $f = \backslash x_1 \ldots x_n \to b$ ⇨ Ptr f (Ptr x_1, ..., Ptr x_n) { ... }

  b) x = K es ⇨ Ptr x = malloc(...); x[i] = e_i; ...

  c) y = e ⇨ Ptr y = e

- Case a) is straightforward, but b) and c) might require general function calls not supported by C's static initializers
- Solution:

```
x = K es        ⇨    Ptr x; Ptr y;
y = e                main() { x = malloc(...); x[i] = e_i; ...
                              y = e; }
```

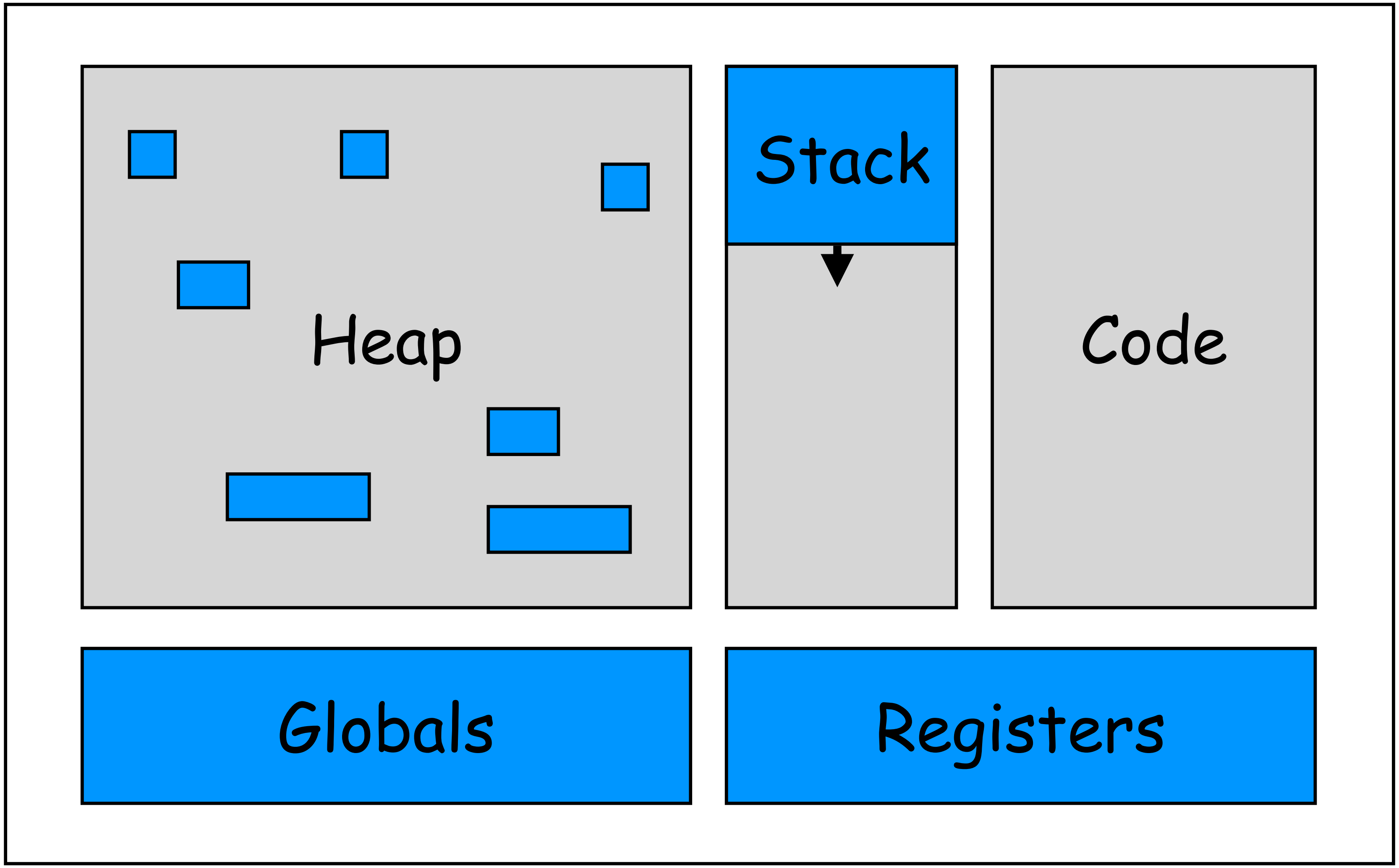
malloc

- Used as a generic name for heap allocation in C — no particular implementation implied
- Our demands:
  - Allocations are frequent, need to be fast
  - Active deallocations do not fit our model of execution (where would one put them?), automatic garbage collection is needed
  - Block sizes vary, compaction might be needed

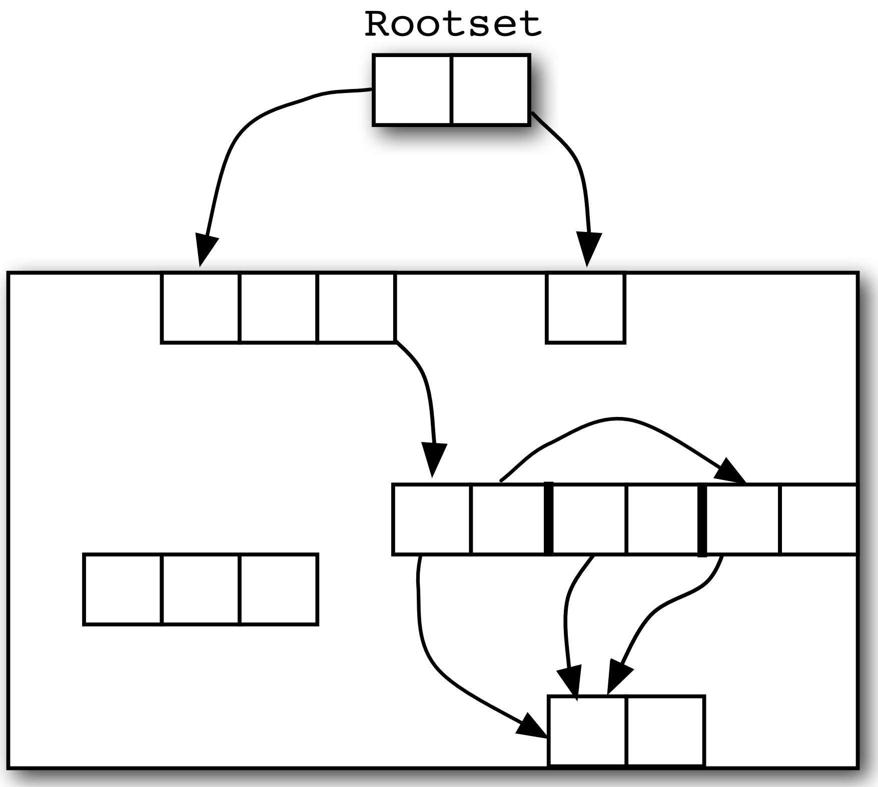
Garbage collection

- At any particular time during code execution:
  - Garbage: allocated heap blocks that are no longer live
  - Live memory: heap blocks that will be used by some subsequent machine instruction
  - Decidable approximation: blocks that are reachable from the current machine state
  - Machine state: globals, registers & stack

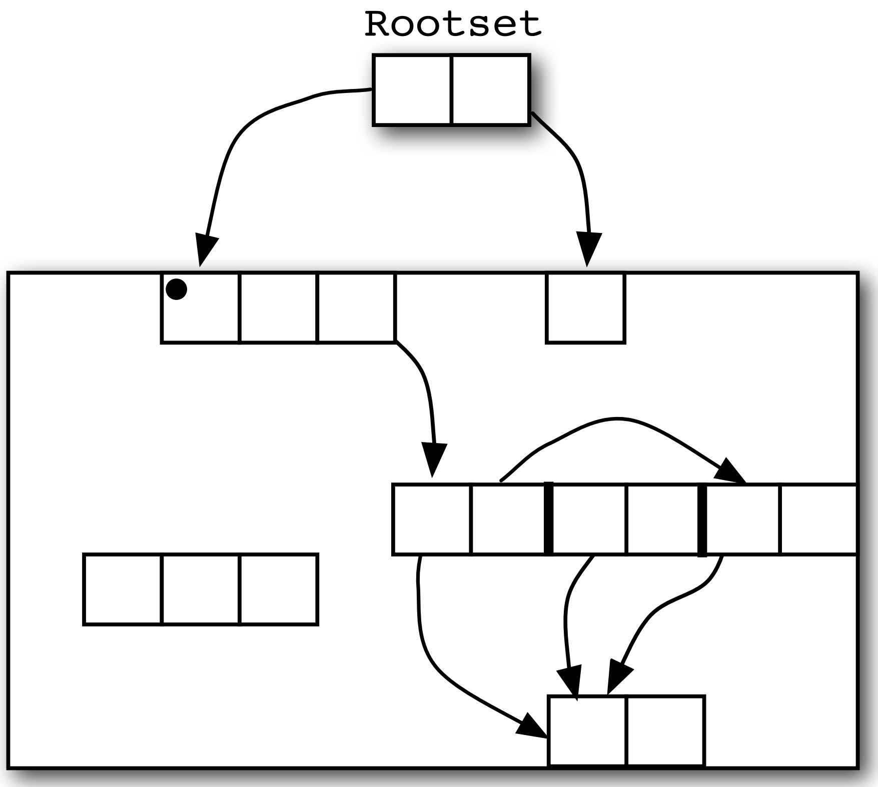
Memory layout



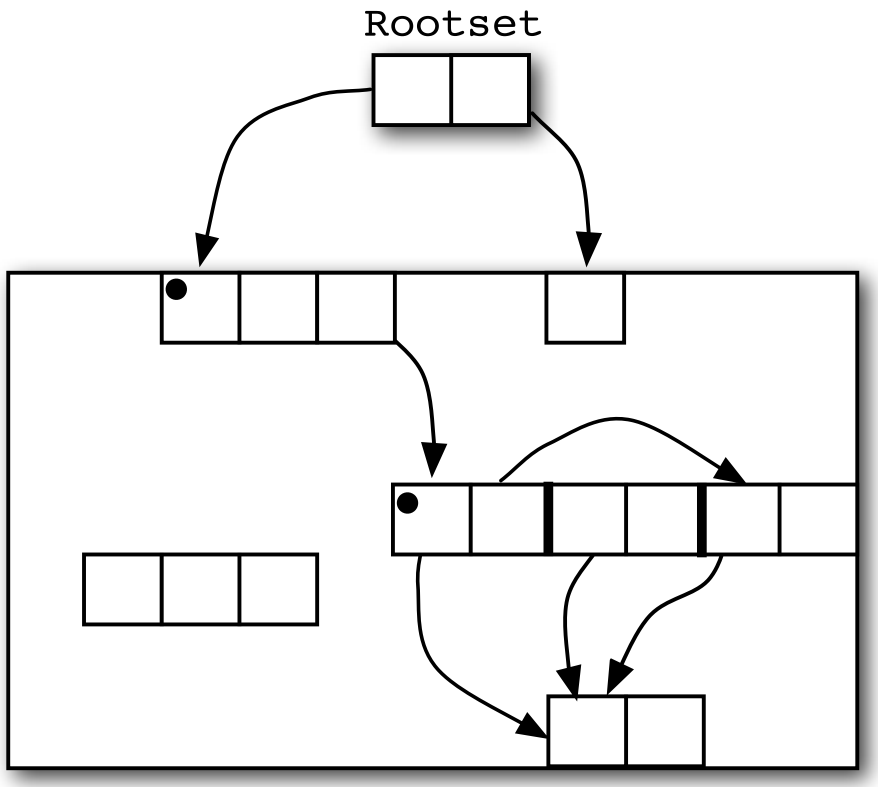
Mark-sweep collection



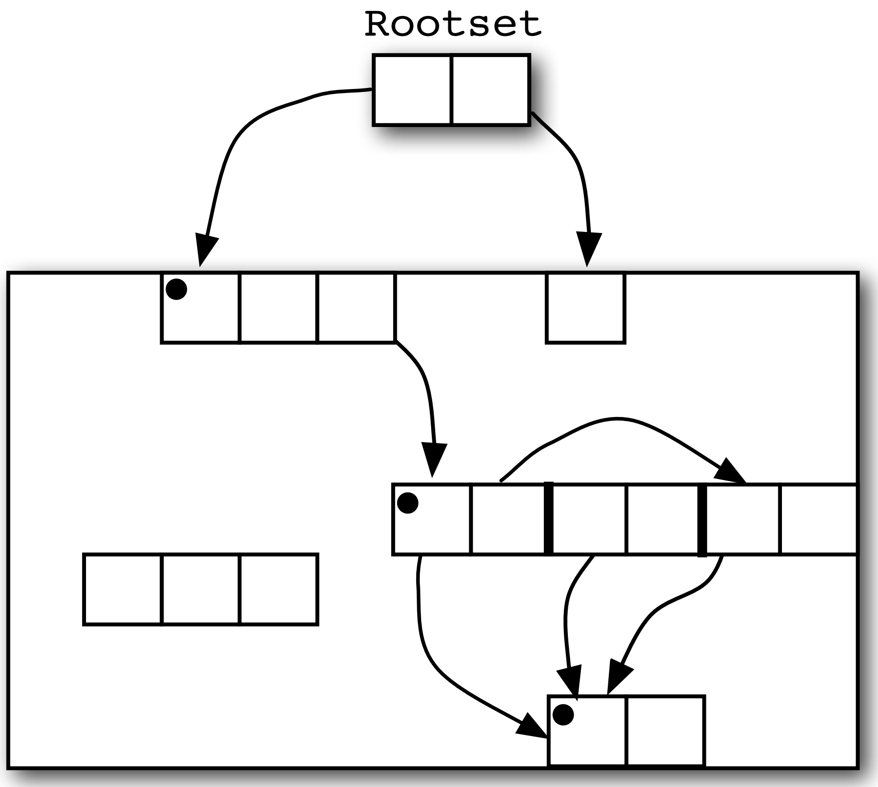
Mark-sweep collection



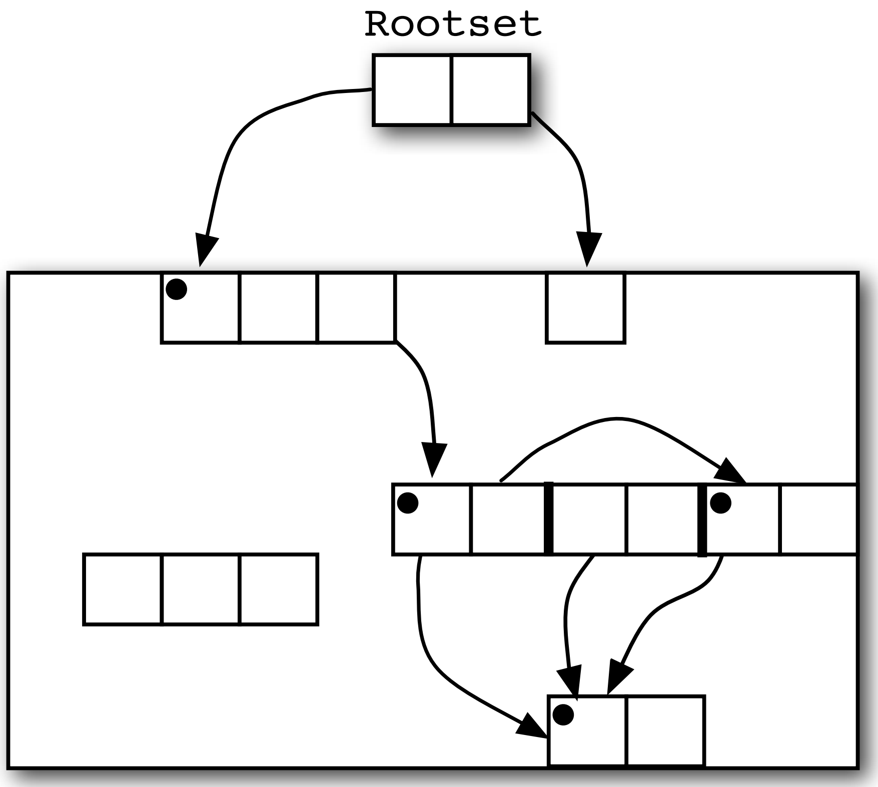
Mark-sweep collection



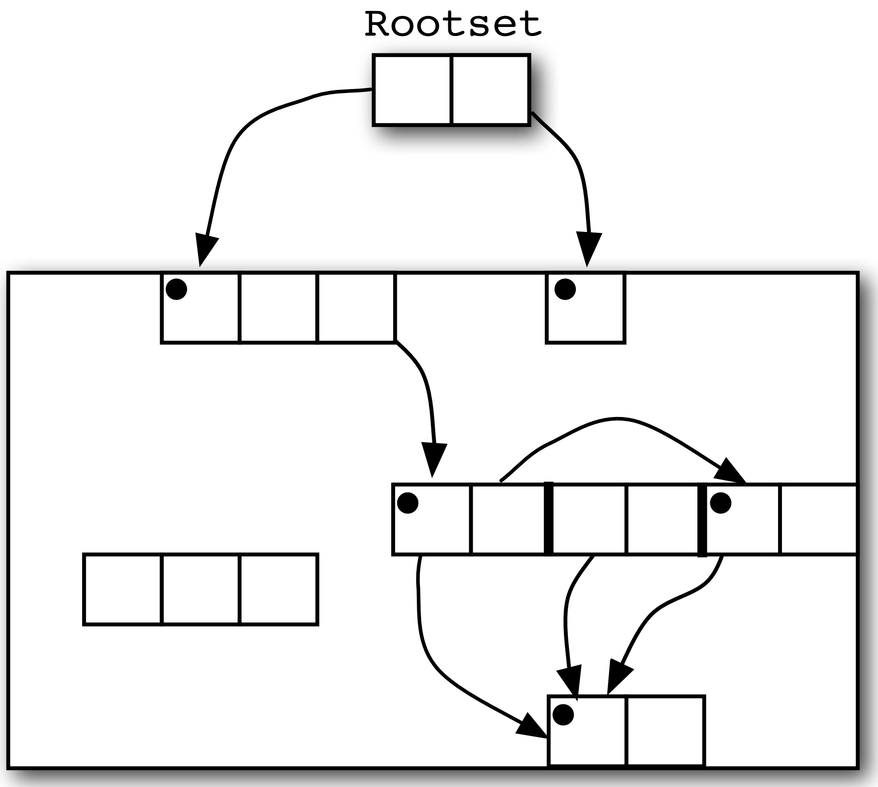
Mark-sweep collection



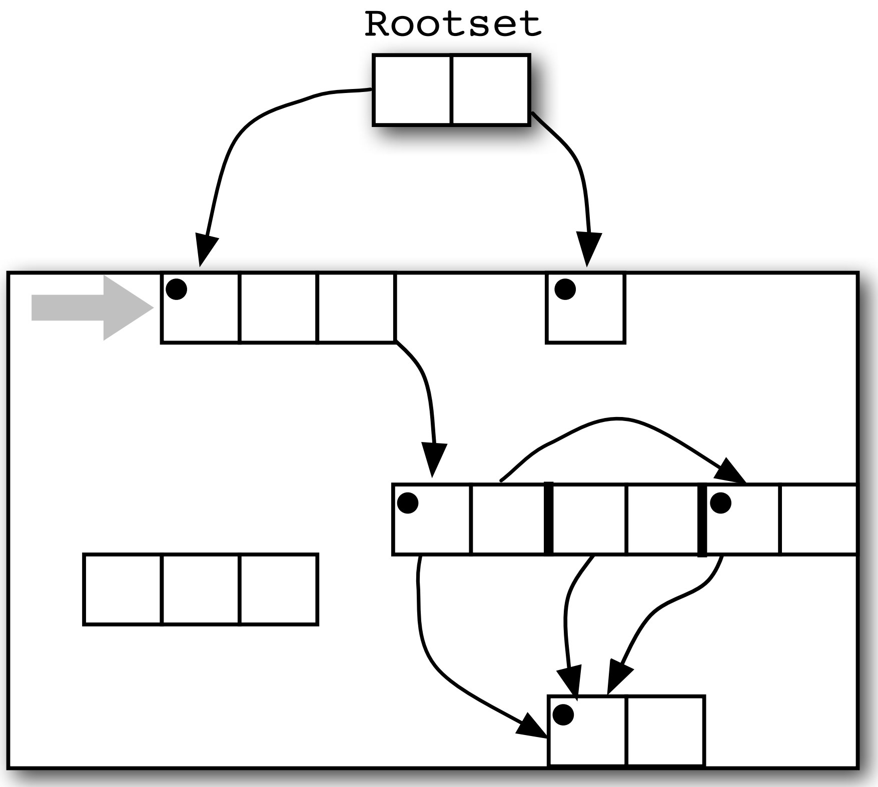
Mark-sweep collection



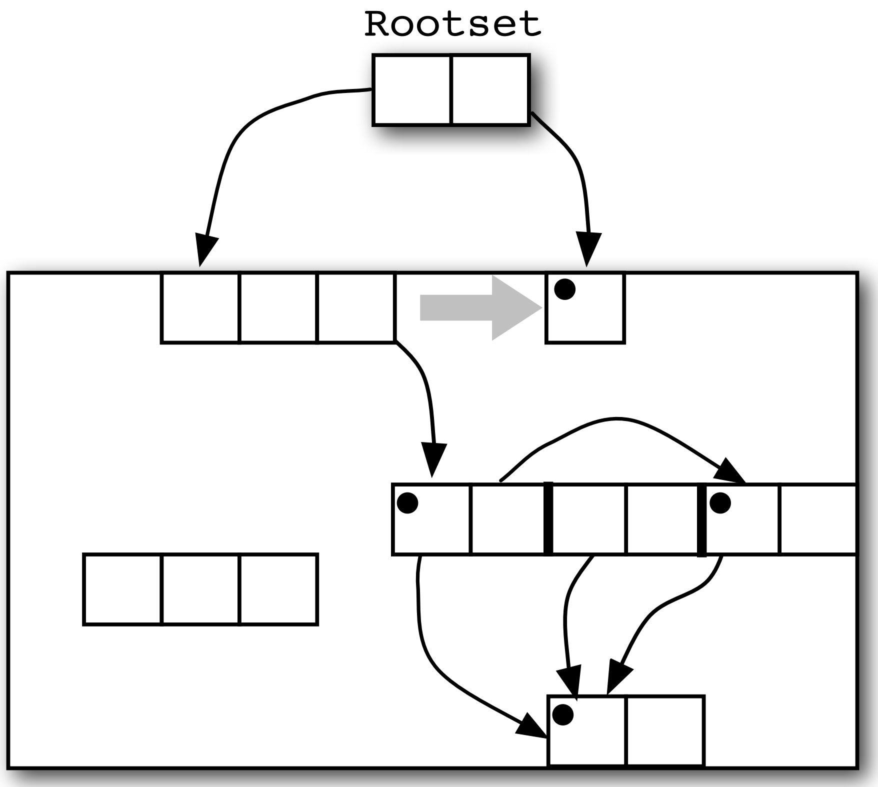
Mark-sweep collection



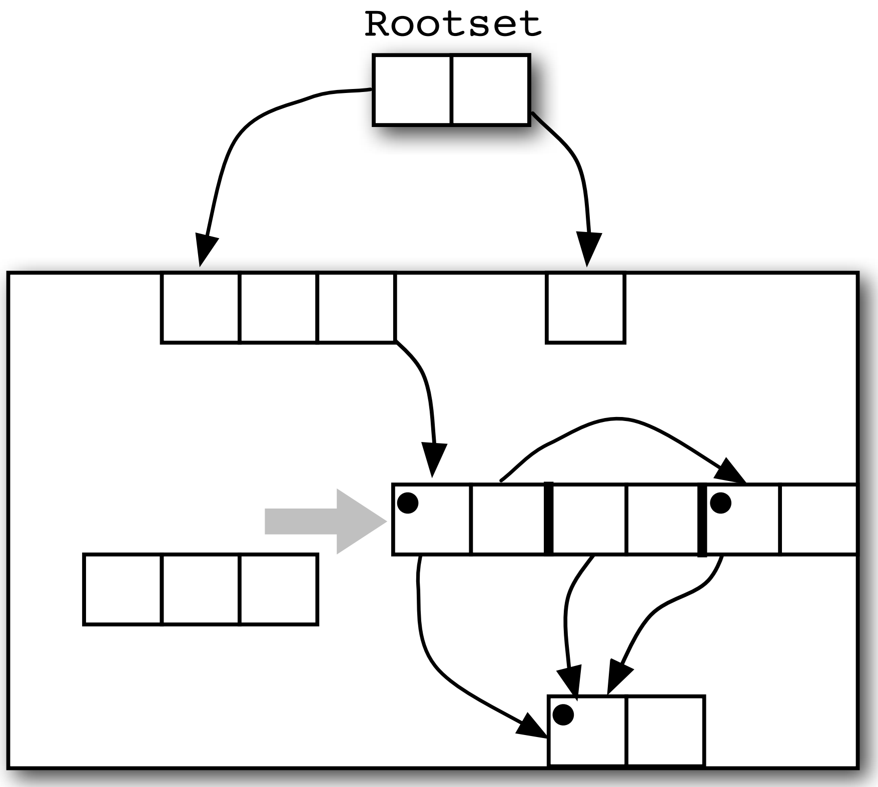
Mark-sweep collection



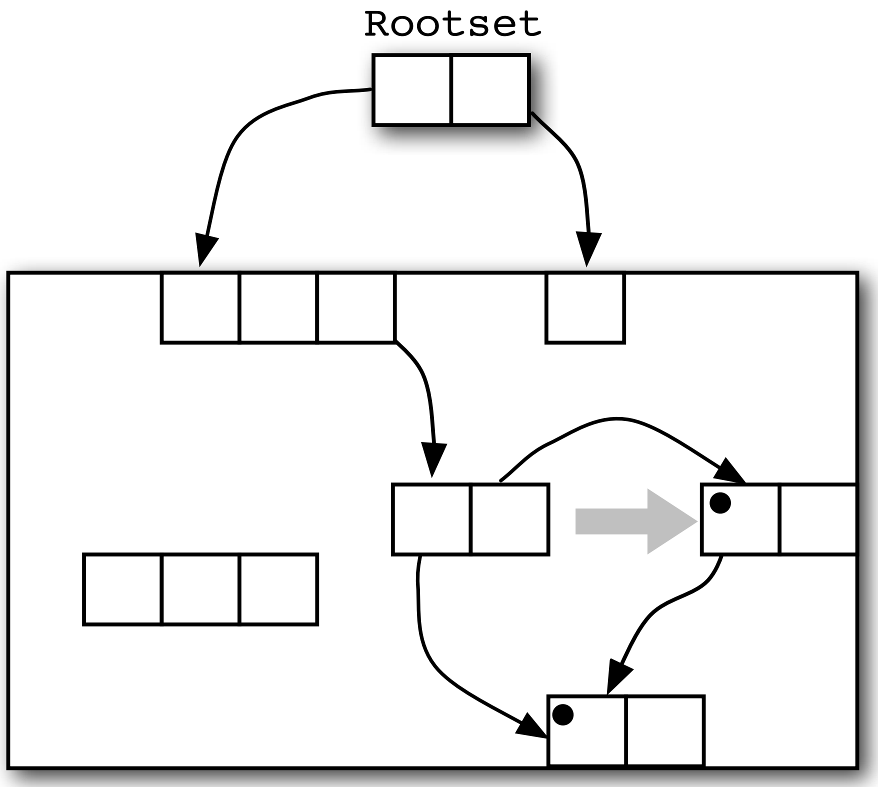
Mark-sweep collection



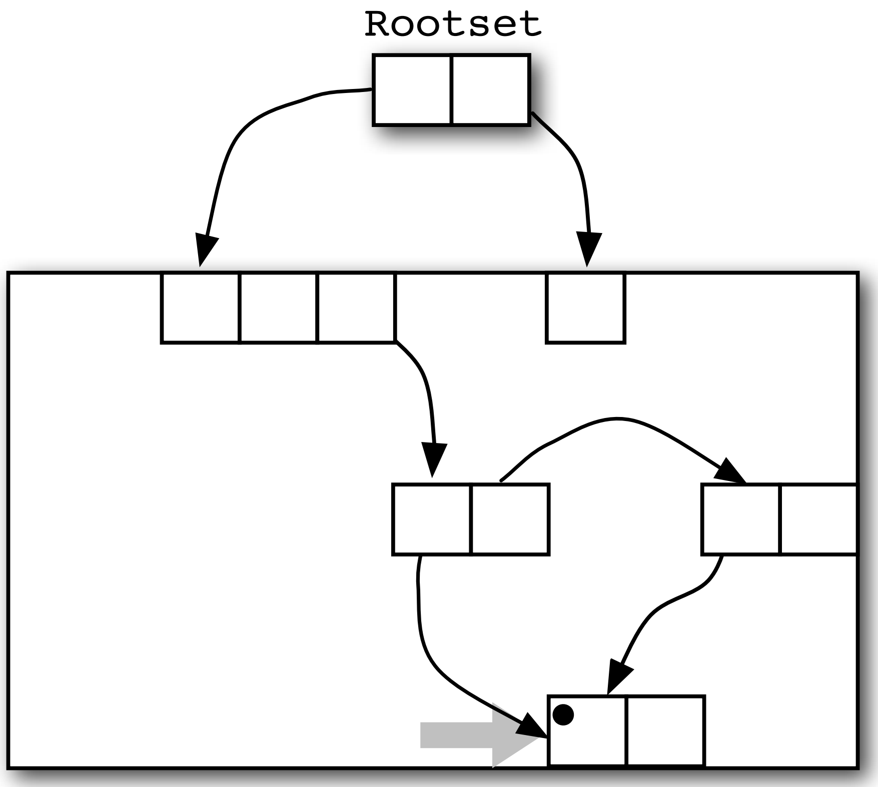
Mark-sweep collection



Mark-sweep collection



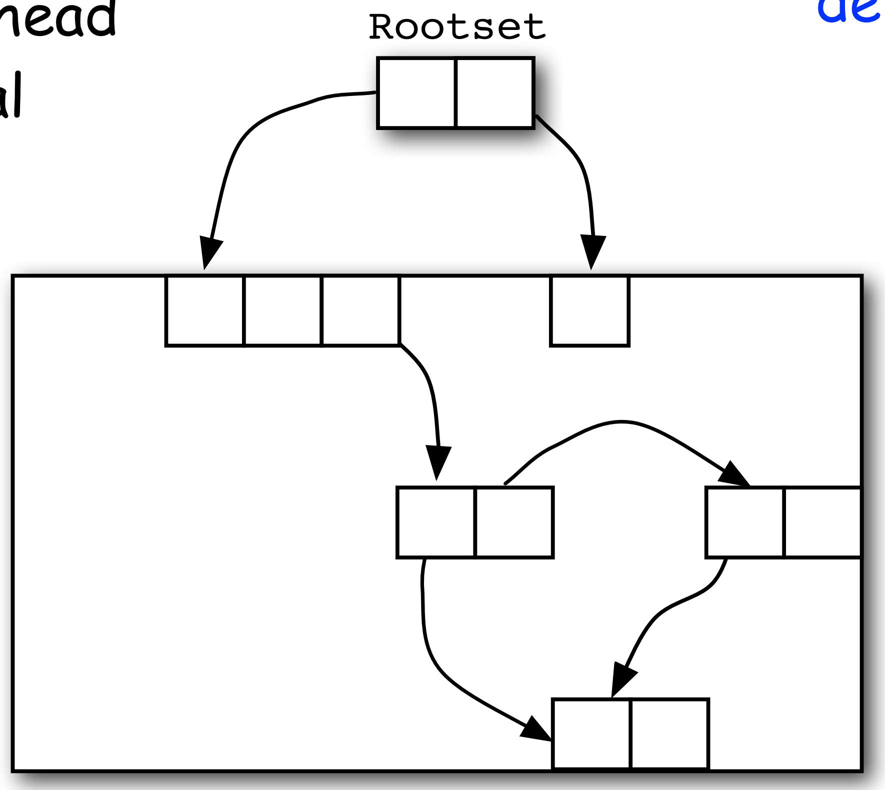
Mark-sweep collection



Mark-sweep collection

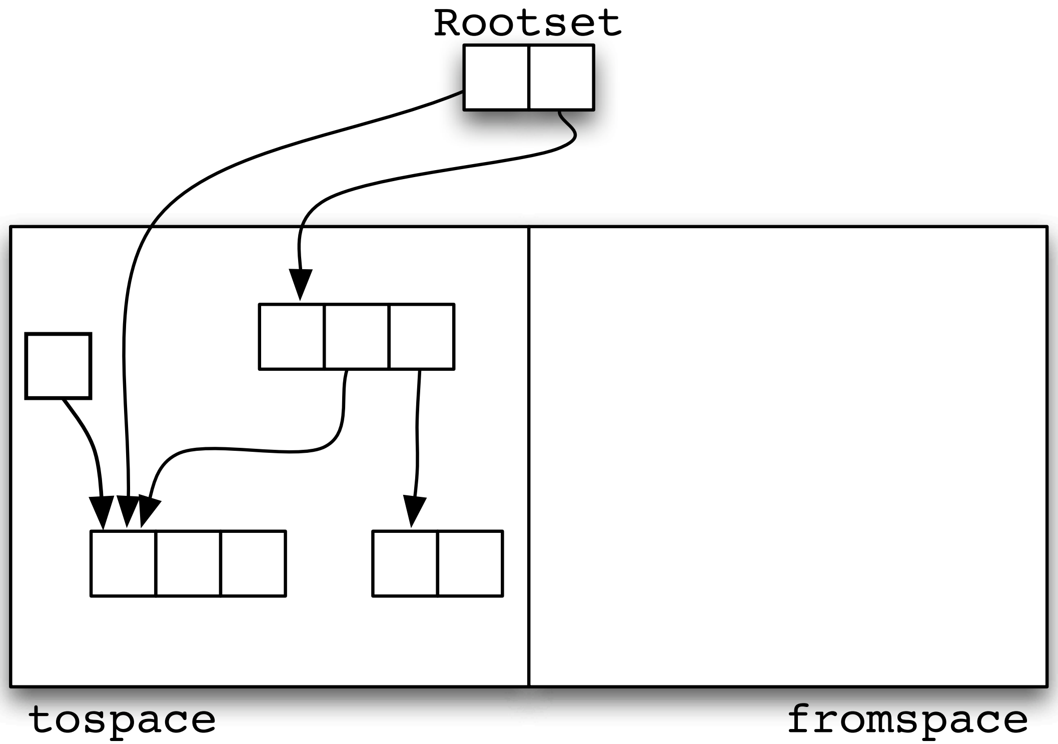
+ low memory overhead
+ easily incremental
- no compaction

depth-first GC!

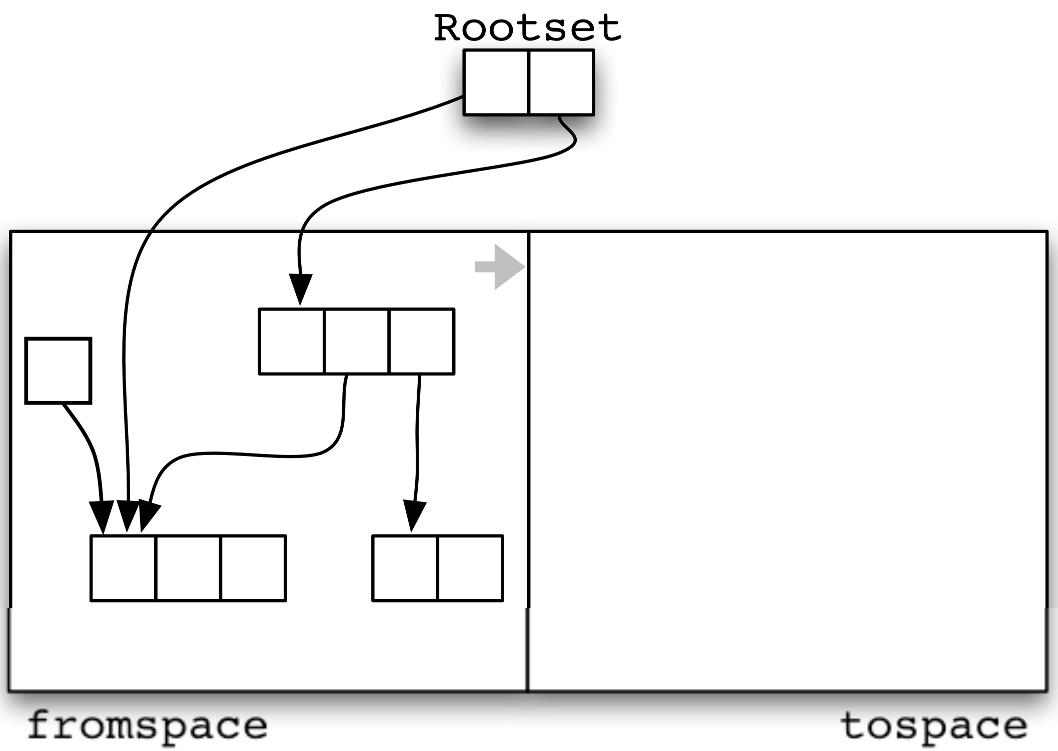


time proportional to <u>number of allocated nodes</u>

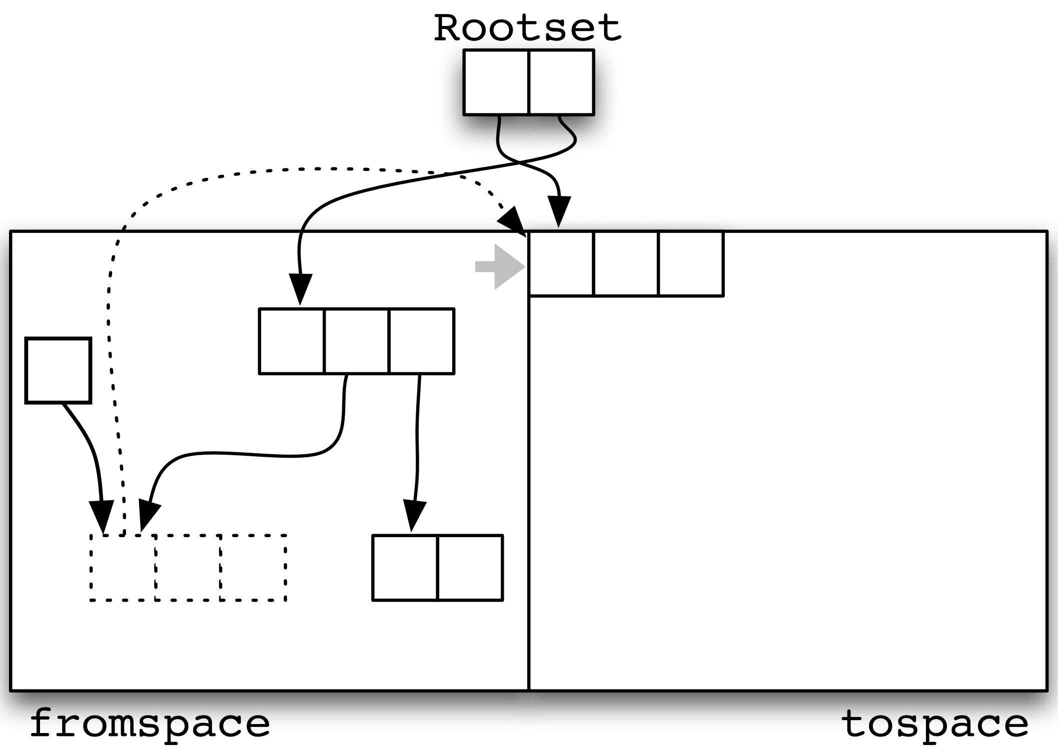
Copying collection



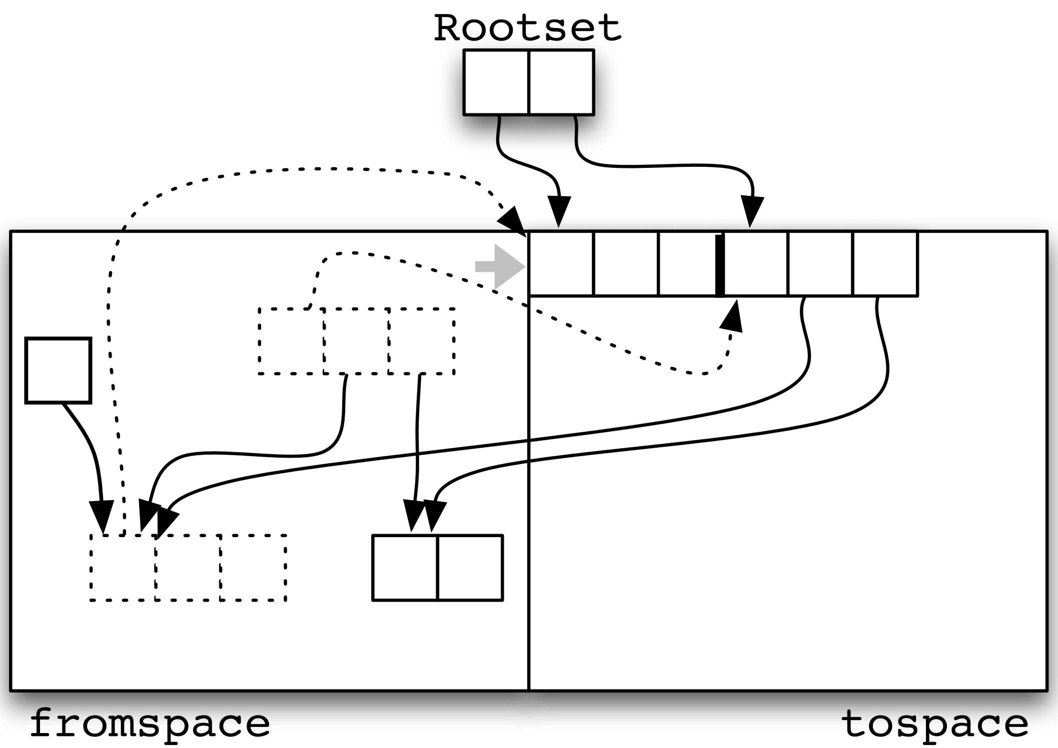
Copying collection



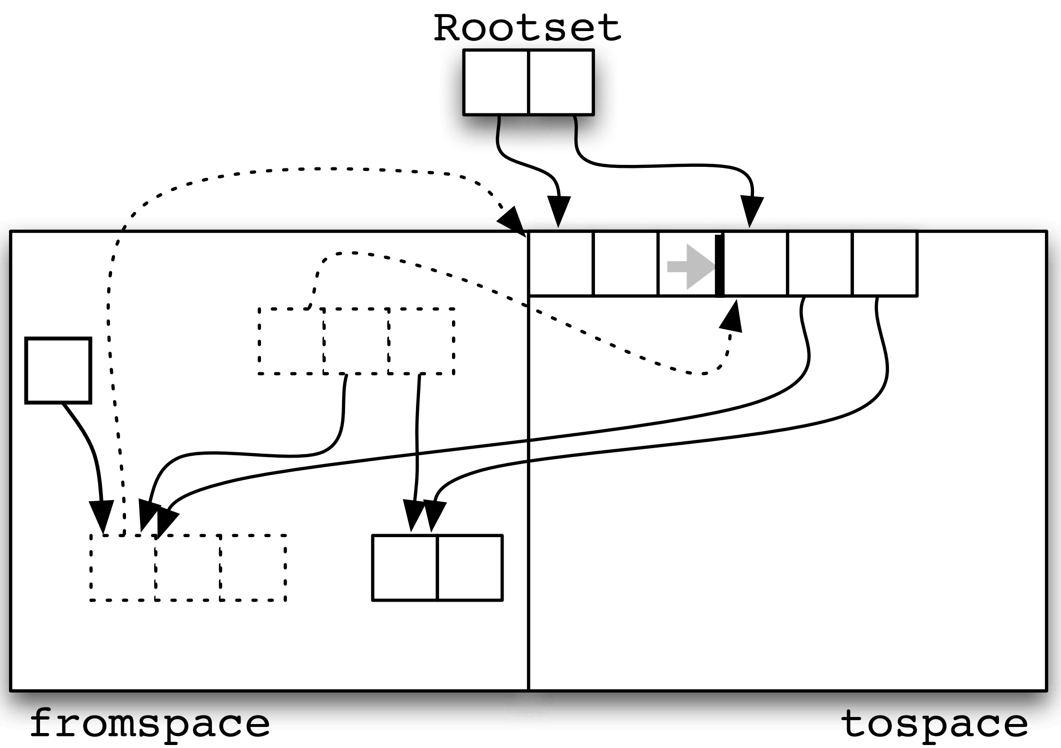
Copying collection



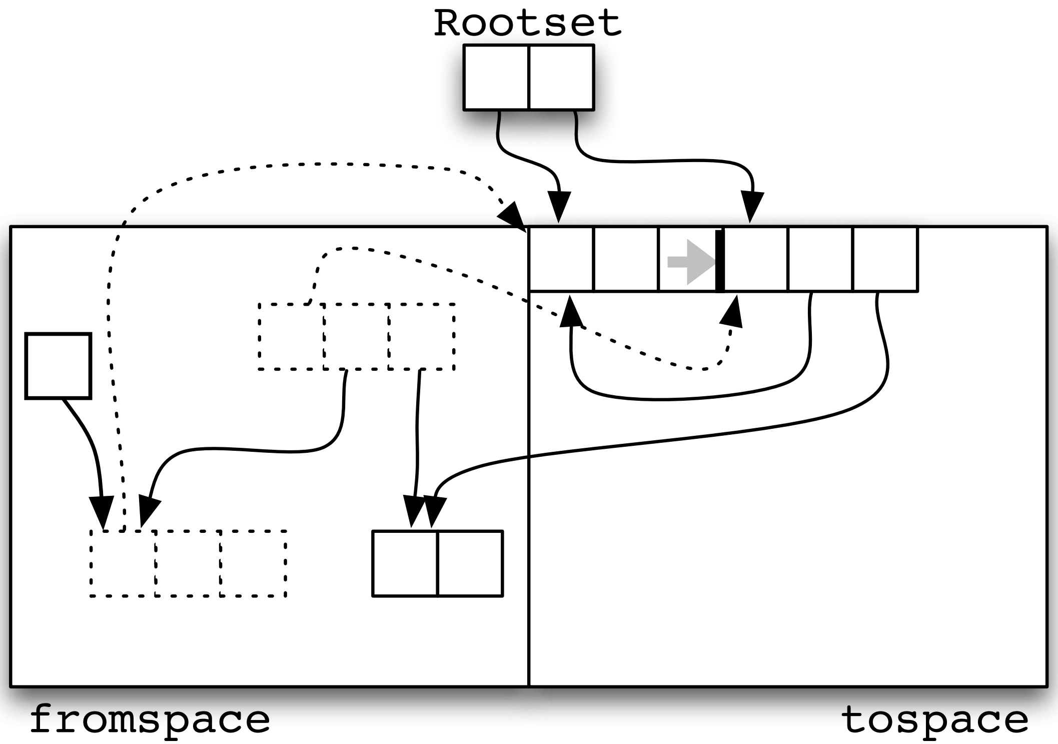
Copying collection



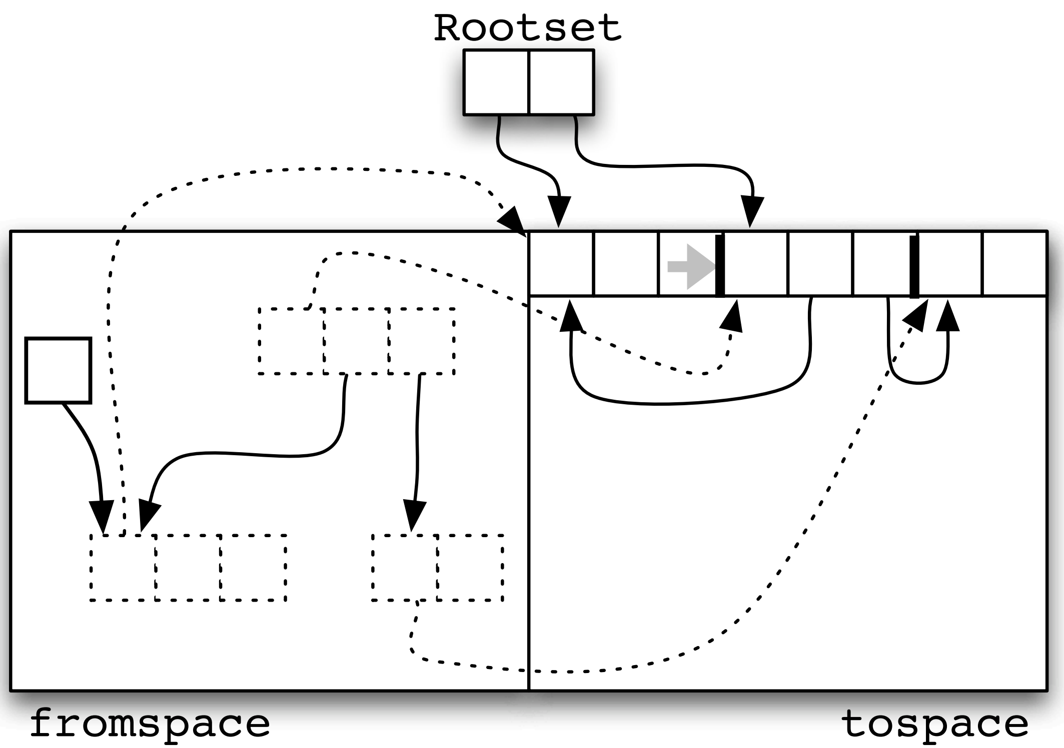
Copying collection



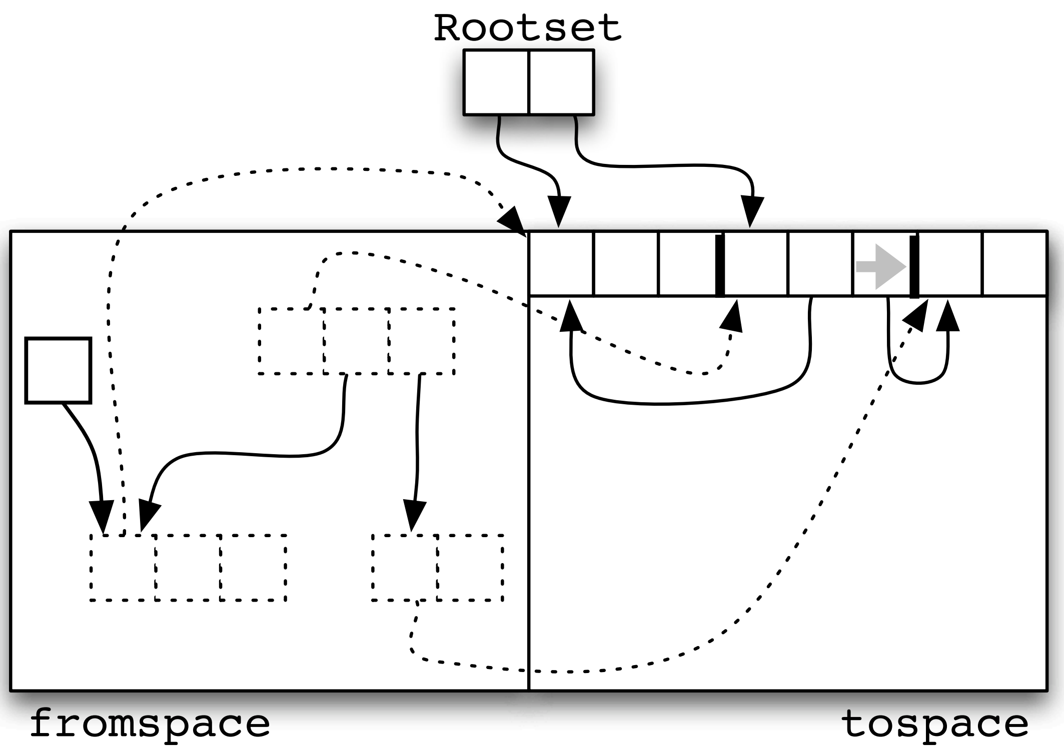
Copying collection



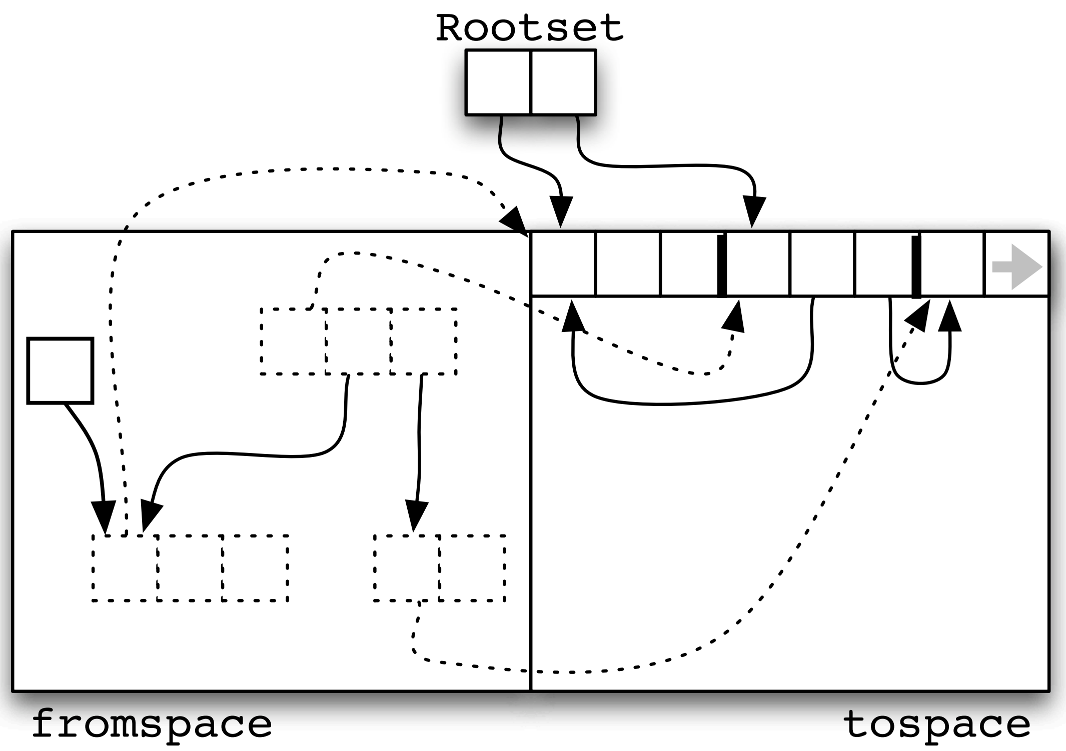
Copying collection



Copying collection



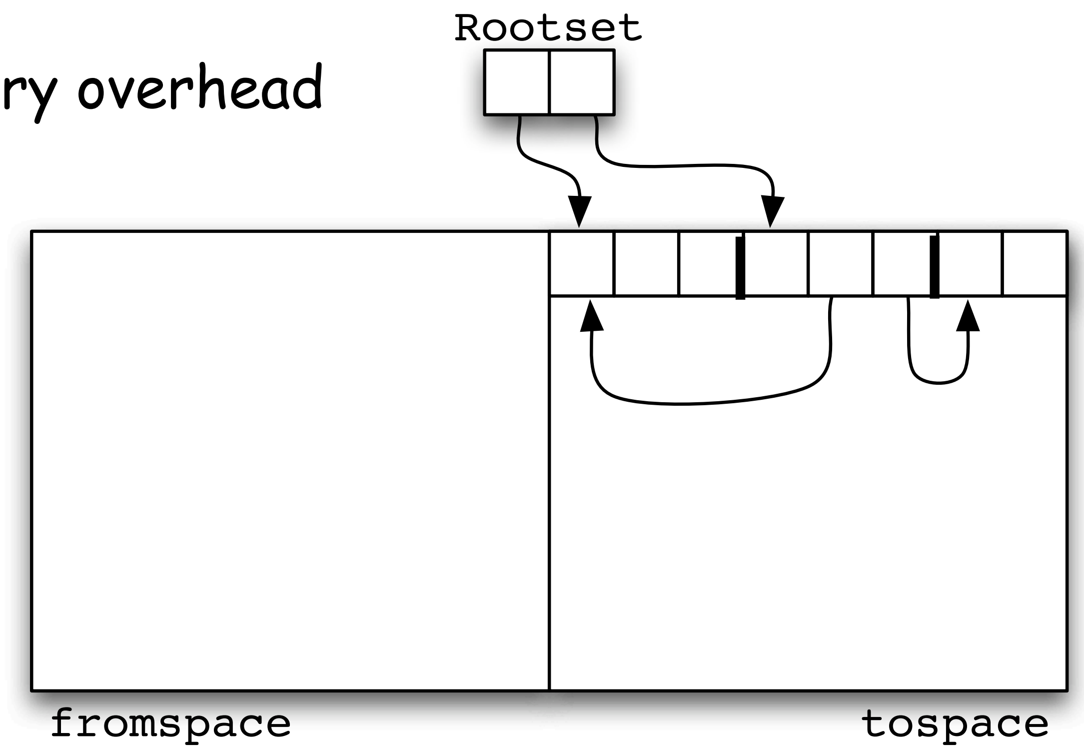
Copying collection



Copying collection

+ cheap allocation
+ compaction
- 100% memory overhead

breadth-first GC!



time proportional to <u>size of live nodes</u>

Finding roots

- A GC observes machine state on assembly level
- Two problems:
  - Finding actual variables among all stored bits (instructions, return addresses, cpu admin, ...)
  - Finding heap pointers among the variables (integers, floats, "small pointers", ...)

Finding heap pointers

- "Small" pointer: value way below start-of-heap
  - no real risk for confusion
- Type information can be used to distinguish integers and floats from pointer variables, but <u>polymorphic code</u> complicates the picture
- Polymorphism implementation strategies:
  - uniform representation (always use pointers)
  - uniform size only (all data fit a machine word)
  - code specialization for non-pointer instances

Uniform representation

- Use heap allocation for <u>all</u> ordinary types
- Integer `n` represented as heap node `Int n`
- Integer arithmetic must (1) extract values from their boxes, (2) perform operation, and (3) store result in a newly allocated `Int`
- Example: `op x (op y z)` ⇒

  ```
  case x of Int x' -> case (case y of Int y' -> case z of
  Int z' -> Int (op' y' z')) of Int v' -> Int (op' x' v')
  ```

  where `op'` is the real operation corresponding to `op`
- Optimizations clearly desirable!

Uniform representation

- GHC uses such a boxed representation for types `Int`, `Float`, etc
- In addition, GHC makes unboxed types `Int#`, `Float#`, etc, available, which <u>cannot</u> be used to instantiate type variables

```
data Int = Int Int#
```

- Literal `n#` <u>really</u> means integer `n`, while `n = Int n#`
- To improve performance, GHC goes to great length to remove repeated boxing and unboxing, even across function calls (with help of types!)

Uniform size

- Our approach so far: just cast literals to Ptr
- Literals must fit the size of pointers, true for Int and Float but not Double on 32-bit machines
- Distinguish dynamically based on

  (A) One bit stolen from every 32-bit value

  (B) Separate bitvectors that describe groupsof polymorphic variables

Uniform size (A)

- Stealing the least significant bit from
  - a pointer: ok if pointers are word-aligned (lowest bits are always 0) and 0 means "ptr"
  - an integer: represent n as $R(n) = 2n+1$ (halves the expressible range), and adjust primitives accordingly ($R(x+y) = 2x+2y+1 = R(x)+R(y)-1$)
  - a float: halves the precision (mask before use)
- Used by O'Caml to good effect

Uniform size (B)

- Adding bit-vector parameters to all polymorphic functions and constructors, which tell how they are instantiated at run-time (ptr/non-ptr flags)
  - Propagates the necessary GC information to non-local scopes
  - Note: only values of variable type need dynamic ptr/non-ptr inspection
  - Avoids the need to tag each value, but adds small overhead to function calls

Code specialization

- Create a specialized copy whenever a polymorphic function is instantiated with a non-pointer type
- Example:

```
rep 0 x = []
rep n x = x : rep (n-1) x
y = (rep 7 (1,1), rep 7 1.1)
```

```
repF 0 x = []
repF n x = x : repF (n-1) x
y = (rep 7 (1,1), repF 7 1.1)
```

- Ensures that polymorphic values are pointers, but at the price of code size increase
- Also works for types that don't fit word size

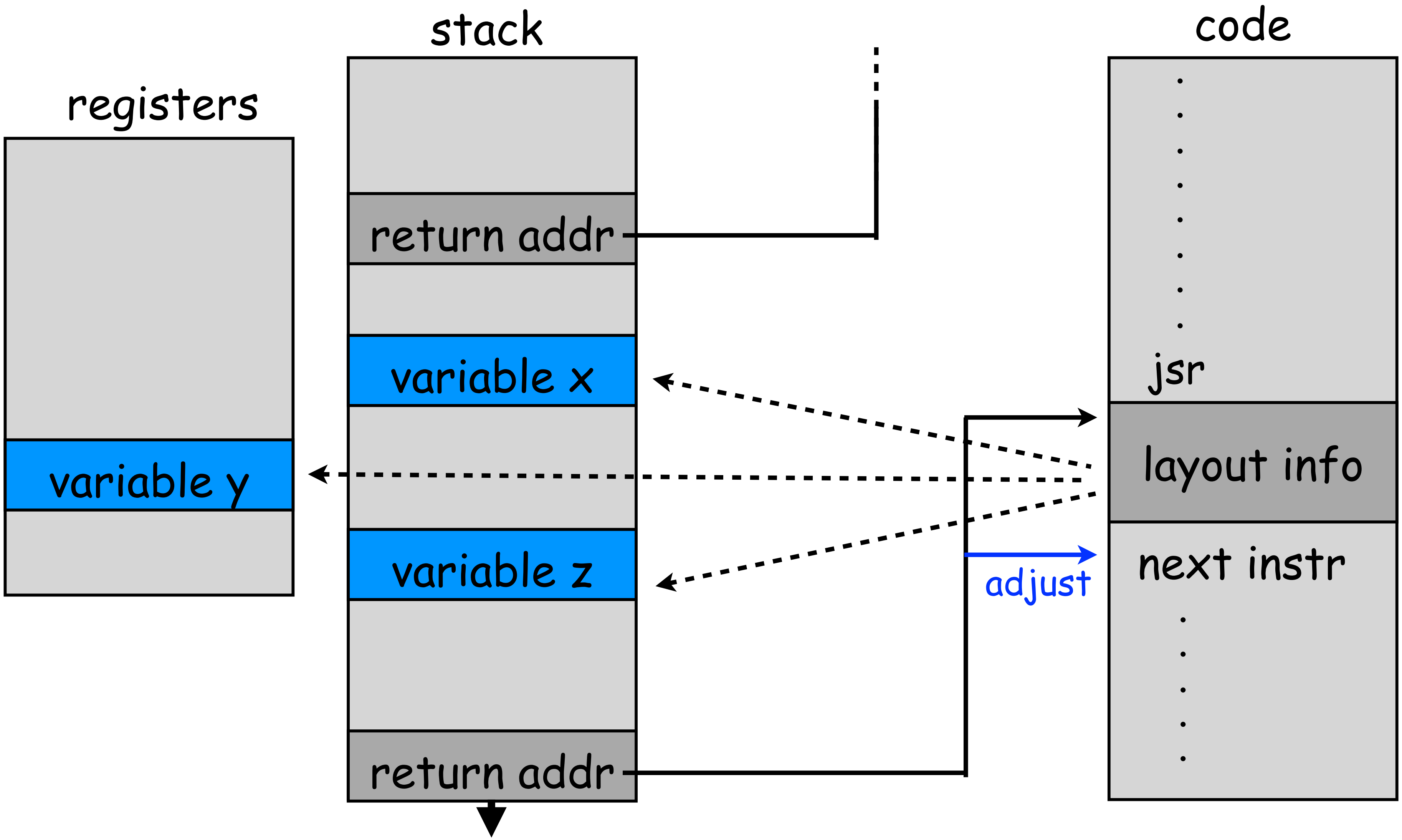
Finding all variables

- Globals are of course trivial...
- ... but layout of variables in registers and the stack is not accessible if we compile to C!
- The traditional conclusion: must generate assembly code in order to give GC full control
- But this also implies the register allocations and instruction scheduling decisions that are performance keys on modern architectures...
- A rather hefty price for the ability to just spot the data layout!

When to run the GC

- When free space drops below some treshold — a natural criterion, detected during allocations
- Memory state must thus be understandable for the GC at least at <u>every malloc call</u>
- Machine state at a malloc call also involves all suspended calls indirectly leading to the malloc — thus <u>all</u> function calls count as potential GC interruption points

Example: GHC



registers
variable y
stack
return addr
variable x
variable z
return addr
code
jsr
layout info
next instr
adjust

Example: GHC

- No cost at function call, minor cost at return
- One layout-table per function call can mean a significant size burden
- Important that static layout table is accurate no matter what path has lead to the call point
- Idea not extensible to concurrent GC (would require a layout table after every instruction!)
- (Demands of GC major motivation behind earlier work on C-- compiler target language)

Conservative GC

- Attractive alternative to writing a complete assembly-level back-end: use C with a conservative garbage collector
- Principal idea: every stack and register word is scanned, and everything that <u>looks like a pointer</u> is <u>treated like one</u>
- "Look like" = word-aligned & within heap & point at beginning of allocated block
- Precludes copying GC (can't mutate guessed root)

Conservative GC

- Leads to memory leaks if many integers, floats, etc, use bit-patterns that are also happen to be valid heap block addresses
- Has nevertheless found good use in practice
- Even eliminates the need to know the pointer-typed variables (but type info might reduce the risk for accidental misinterpretation if present)
- Ready to use in the form of a tried-and-tested implementation: the Boehm-Demers-Weiser GC library ‹http://www.hpl.hp.com/personal/Hans_Boehm/gc/›

Recommendation

For the lab project:

Use the Boehm-Demers-Wiser collector!

Stack management

- A comparably simple issue!
- Sole concern: detect stack overflow and quit instead of continuing with corrupted data
- Handled automatically by memory-management hardware on most platforms, under most operating systems
- Should such service not exist, a simple check at the beginning of each generated C function will do the job

Optimizing tail recursion

- Main reason for excessive stack usage: deeply recursive algorithms
- Unnecessarily stack-hungry code: a tail-recursive function (ends with recursive call)
- Can easily be translated into imperative loops

```
sum a [] = a
sum a (x:xs) = sum (a+x) xs
```

⇨

```
sum (a, x) {
  while (1)
    if (x==0) return a;
    else {a += x; x = x[1];}
}
```

Summary

- Garbage collection a necessity for FP
- Collection techniques: copying vs. mark-sweep
- Relies on ability to find all program variables, and to distinguish pointers from other values
- Challenge: devise a means to locate variables without having to build a full low-level back-end
- Conservative collectors can work without knowing where the variables are, at some higher risks for space leaks