Memory abstraction, verification of a garbage collector

Lecture 5

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

Memory

Machine code:

- load, store instructions access memory
- memory is large flat array

23	12	3	6	0	0	0	33	...	12

Memory

Machine code:

- load, store instructions access memory
- memory is large flat array

23	12	3	6	0	0	0	33	...	12

In machine-code Hoare triples:

- assertion about a single cell of memory (from prev. lecture):

```
M a x = (Mem a) ↦ (Word x)
```

Memory

Machine code:

- load, store instructions access memory
- memory is large flat array

23	12	3	6	0	0	0	33	...	12

In machine-code Hoare triples:

- assertion about a single cell of memory (from prev. lecture):

```
M a x = (Mem a) ↦ (Word x)
```

- assertion about a region of memory:

```
MEM m = λs. domain s = { Mem a | a ∈ domain m} ∧
            ∀a ∈ domain m. s (Mem a) = Word (m a)
```

single-word memcopy

Using MEM we can decompile

```
“ load r0,[r2]
  store r0,[r1] ”
```

The extracted function:

```
memcopy (r1,r2,m) =
  let r0 = m r2 in
  let m = m[r1 ↦ r0] in
    (r0,r1,r2,m)
```

single-word memcopy

Using MEM we can decompile

```
“ load r0,[r2]
  store r0,[r1] ”
```

The extracted function:

```
memcopy (r1,r2,m) =
  let r0 = m r2 in
  let m = m[r1 ↦ r0] in
    (r0,r1,r2,m)
```

```
memcopy_pre (r1,r2,m) =
  let cond1 = r2 ∈ domain m in
  let r0 = m r2 in
  let cond2 = r1 ∈ domain m in
  let m = m[r1 ↦ r0] in
    cond1 ∧ cond2
```

single-word memcopy

Using MEM we can decompile

```
“ load r0,[r2]
  store r0,[r1] ”
```

The extracted function:

```
memcopy (r1,r2,m) =
  let r0 = m r2 in
  let m = m[r1 ↦ r0] in
    (r0,r1,r2,m)
```

```
memcopy_pre (r1,r2,m) =
  let cond1 = r2 ∈ domain m in
  let r0 = m r2 in
  let cond2 = r1 ∈ domain m in
  let m = m[r1 ↦ r0] in
    cond1 ∧ cond2
```

The certificate theorem:

```
memcopy_pre (r1,r2,m) ⟹
{ PC pc * R 0 r0 * R 1 r1 * R 2 r2 * MEM m }
  “ load r0,[r2]
    store r0,[r1] ”
{ let (r0,r1,r2,m) = memcopy (r1,r2,m) in
    PC (pc + 8) * R 0 r0 * R 1 r1 * R 2 r2 * MEM m }
```

FP's memory abstraction

FP requires memory abstraction

In Lisp, all data is s-expressions, e.g.

```
1  nil  (1 . 2)  (a b c d)
```

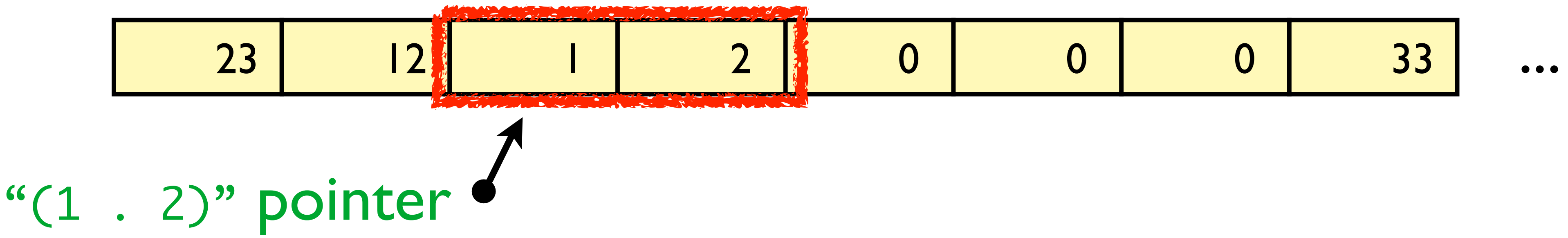
FP's memory abstraction

FP requires memory abstraction

In Lisp, all data is s-expressions, e.g.

```
1  nil  (1 . 2)  (a b c d)
```

Representation in memory:



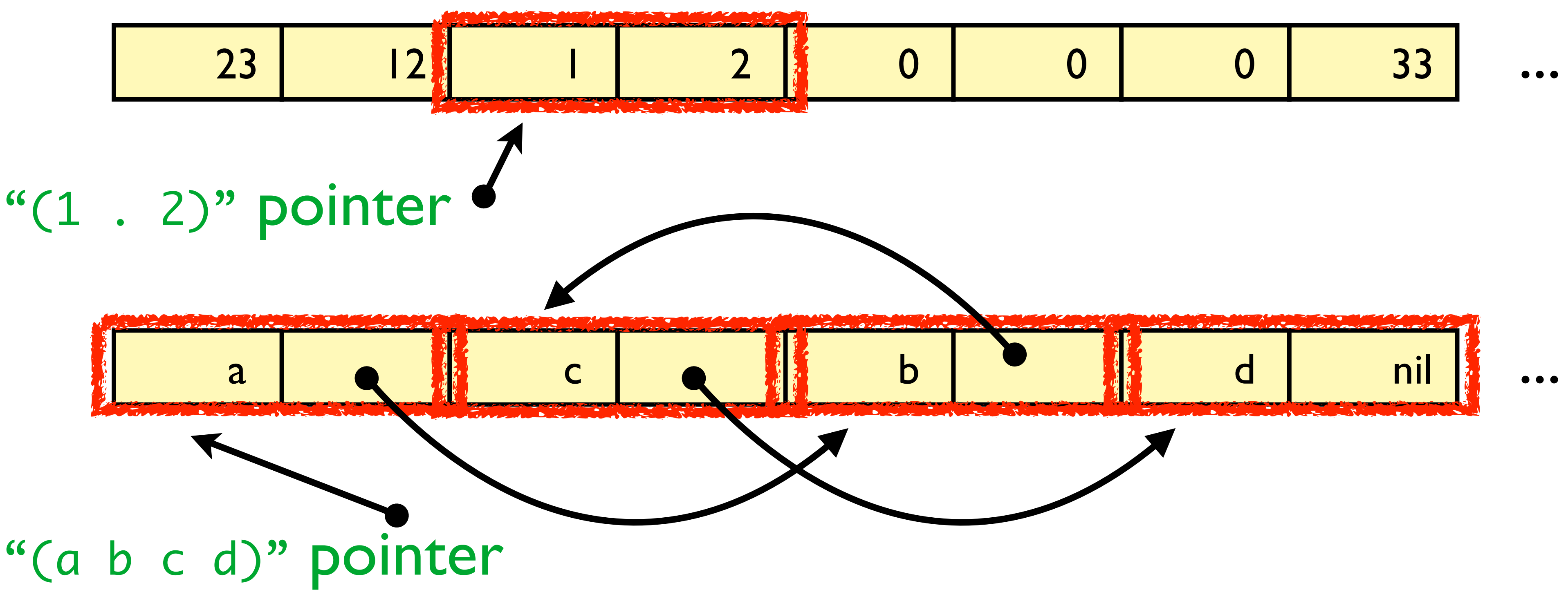
FP's memory abstraction

FP requires memory abstraction

In Lisp, all data is s-expressions, e.g.

`1  nil  (1 . 2)  (a b c d)`

Representation in memory:



Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

allocates new

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

allocates new

makes some cons-data unreachable, i.e. garbage

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

allocates new

makes some cons-data unreachable, i.e. garbage

Eventually, the entire memory is full of unusable old data...

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

allocates new

makes some cons-data unreachable, i.e. garbage

Eventually, the entire memory is full of unusable old data...

Garbage collection (GC) finds and deletes unused data.

Garbage collection

Evaluation of Lisp expression:

```
> (car (cons 1 (cons 2 nil)))
1
```

allocates new

allocates new

makes some cons-data unreachable, i.e. garbage

Eventually, the entire memory is full of unusable old data...

Garbage collection (GC) finds and deletes unused data.

GC is invisible to user (part of memory abstraction).

Garbage collection (cont.)

Jargon:

- moving or non-moving?
- copying or mark-and-sweep or mark-and-don't-sweep?
- generational or not?
- stop-the-world or incremental or concurrent?
- precise or conservative?

Garbage collection (cont.)

Jargon:

moving or non-moving?
copying or mark-and-sweep or mark-and-don't-sweep?
generational or not?
stop-the-world or incremental or concurrent?
precise or conservative?

This lecture: verification of a simple copying GC.

Organising a verification proof

Task: construction of verified code for GC routine

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Step 2: write abstract implementation
(small-step relation), prove correct w.r.t spec

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Step 2: write abstract implementation
(small-step relation), prove correct w.r.t spec

Step 3: introduce a more concrete notion of memory,
prove connection with small-step relation

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Step 2: write abstract implementation
(small-step relation), prove correct w.r.t spec

Step 3: introduce a more concrete notion of memory,
prove connection with small-step relation

Step 4: write assembly, use decompilation to produce
functions with concrete types

Organising a verification proof

Task: construction of verified code for GC routine

Plan: stepwise refinement from high-level specification

Step 1: specify what GC is to achieve

Step 2: write abstract implementation
(small-step relation), prove correct w.r.t spec

Step 3: introduce a more concrete notion of memory,
prove connection with small-step relation

Step 4: write assembly, use decompilation to produce
functions with concrete types

Step 5: prove connection between impl. of Step 3 and 4.

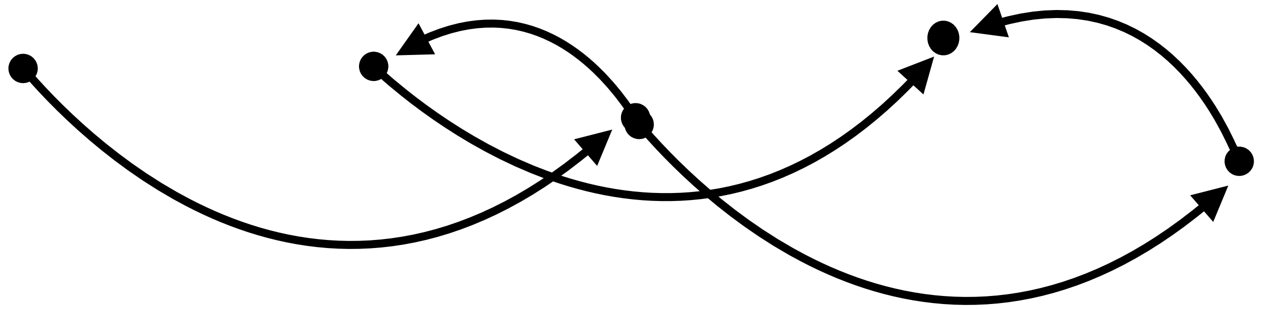
Specification of copying GC

How to model the ‘heap’ (i.e. memory) abstractly?

Specification of copying GC

How to model the ‘heap’ (i.e. memory) abstractly?

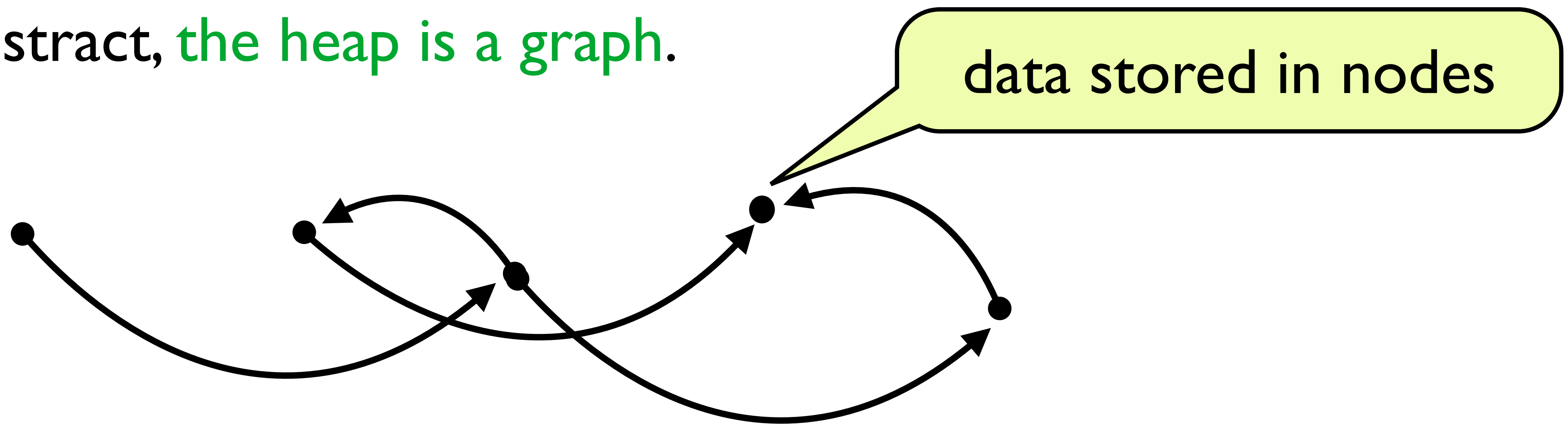
In the abstract, the heap is a graph.



Specification of copying GC

How to model the ‘heap’ (i.e. memory) abstractly?

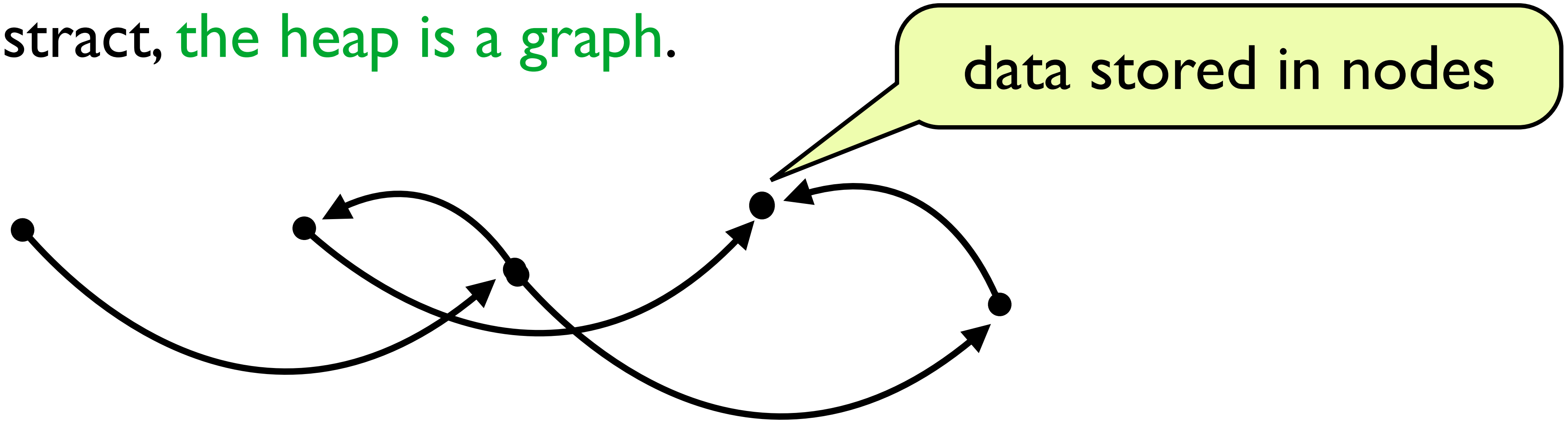
In the abstract, the heap is a graph.



Specification of copying GC

How to model the 'heap' (i.e. memory) abstractly?

In the abstract, the heap is a graph.



We model the graph as a finite partial map from `num` to `heap_node`.

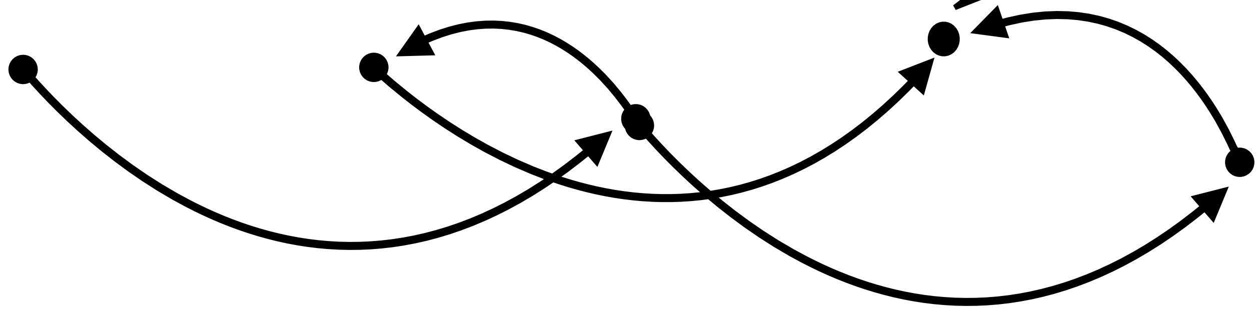
```
heap_addr ::= LHS num | RHS 'ptr_data
heap_node ::= (heap_addr list, 'data)
```

Specification of copying GC

How to model the 'heap' (i.e. memory) abstractly?

In the abstract, the heap is a graph.

data stored in nodes



We model the graph as a finite partial map from `num` to `heap_node`.

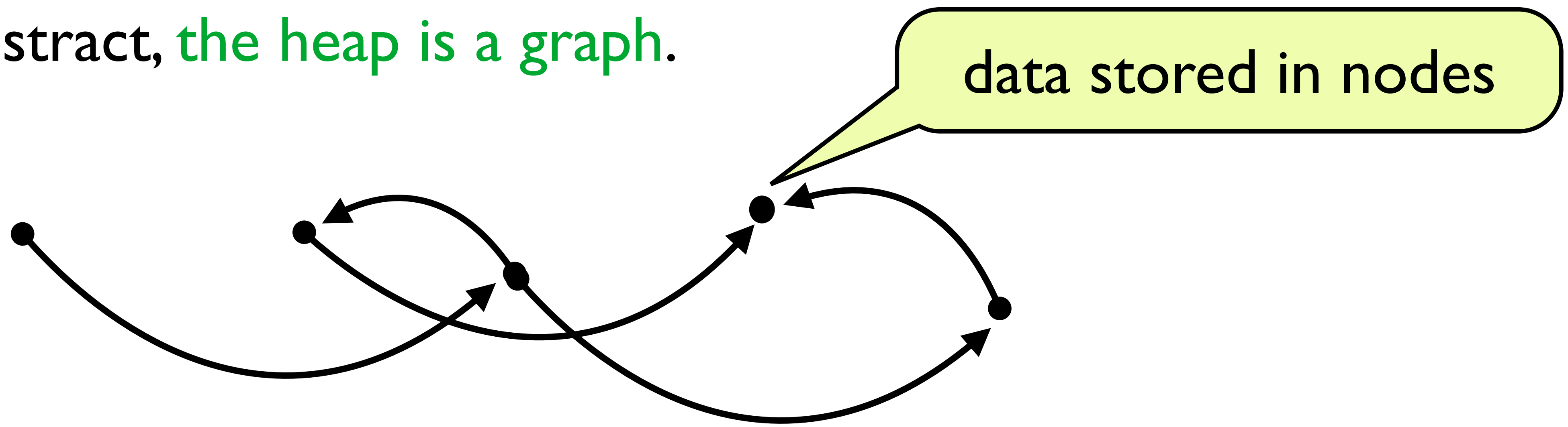
```
heap_addr ::= LHS num | RHS 'ptr_data
heap_node ::= (heap_addr list, 'data)
```

misaligned ptrs are data

Specification of copying GC

How to model the ‘heap’ (i.e. memory) abstractly?

In the abstract, the heap is a graph.



We model the graph as a finite partial map from `num` to `heap_node`.

```
heap_addr ::= LHS num | RHS 'ptr_data
heap_node ::= (heap_addr list, 'data)
```

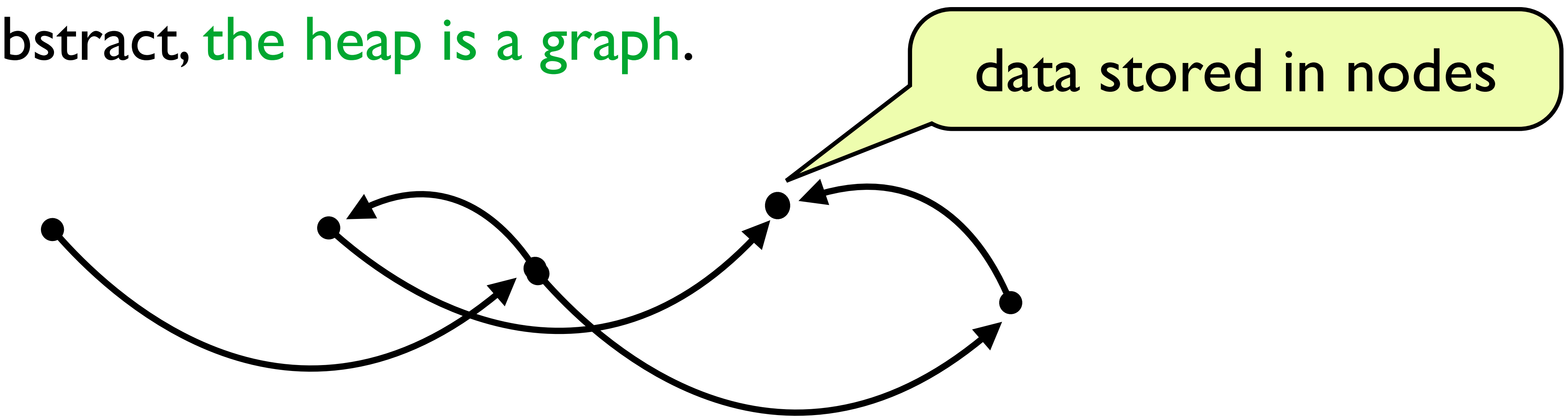
misaligned ptrs are data

State = heap graph + root pointers (active pointers in program).

Specification of copying GC

How to model the 'heap' (i.e. memory) abstractly?

In the abstract, the heap is a graph.



We model the graph as a finite partial map from `num` to `heap_node`.

```
heap_addr ::= LHS num | RHS 'ptr_data
heap_node ::= (heap_addr list, 'data)
```

misaligned ptrs are data

at this level, type variable

State = heap graph + root pointers (active pointers in program).

Reachability

GC must not delete reachable nodes. Reachable:

$$\frac{a \in \mathsf{set}\ roots}{a \in \mathsf{reach}\ (h, roots)}$$

$$\frac{a \in \mathsf{set}\ as \ \wedge\ h(b) = (as, data)\ \wedge\ b \in \mathsf{reach}\ (h, roots)}{a \in \mathsf{reach}\ (h, roots)}$$

A full GC ought to only keep reachable nodes:

$$\mathsf{filter}\ (h, roots)\ =\ (h{\restriction}(\mathsf{reach}\ (h, roots)), roots)$$

Reachability

GC must not delete reachable nodes. Reachable:

$$\frac{a \in \mathsf{set}\ roots}{a \in \mathsf{reach}\ (h, roots)}$$

$$\frac{a \in \mathsf{set}\ as\ \wedge\ h(b) = (as, data)\ \wedge\ b \in \mathsf{reach}\ (h, roots)}{a \in \mathsf{reach}\ (h, roots)}$$

A full GC ought to only keep reachable nodes:

$$\mathsf{filter}\ (h, roots)\ =\ (h\lfloor(\mathsf{reach}\ (h, roots)), roots)$$

restricts domain of h function to reach set

Moving

A moving GC is allowed to rename addresses:

Moving

A moving GC is allowed to rename addresses:

We are allowed to apply a renaming function.

$$\textsf{domain}\ (\textsf{rename}\ f\ \ h) = \textsf{image}\ f\ \ (\textsf{domain}\ h)$$
$$(\textsf{rename}\ f\ \ h)(f(x)) = (\textsf{map}\ f\ \ as, d) \qquad \text{whenever } h(x) = (as, d)$$

Define:

$$\frac{f \circ f = \textsf{id}}{(h, roots) \xrightarrow{\textsf{translate}} (\textsf{rename}\ f\ \ h, \textsf{map}\ f\ \ roots)}$$

Moving

A moving GC is allowed to rename addresses:

We are allowed to apply a renaming function.

$$\begin{aligned} \mathsf{domain}\ (\mathsf{rename}\ f\ \ h) &= \mathsf{image}\ f\ \ (\mathsf{domain}\ h) \\ (\mathsf{rename}\ f\ \ h)(f(x)) &= (\mathsf{map}\ f\ \ as, d) \qquad \text{whenever } h(x) = (as, d) \end{aligned}$$

Define:

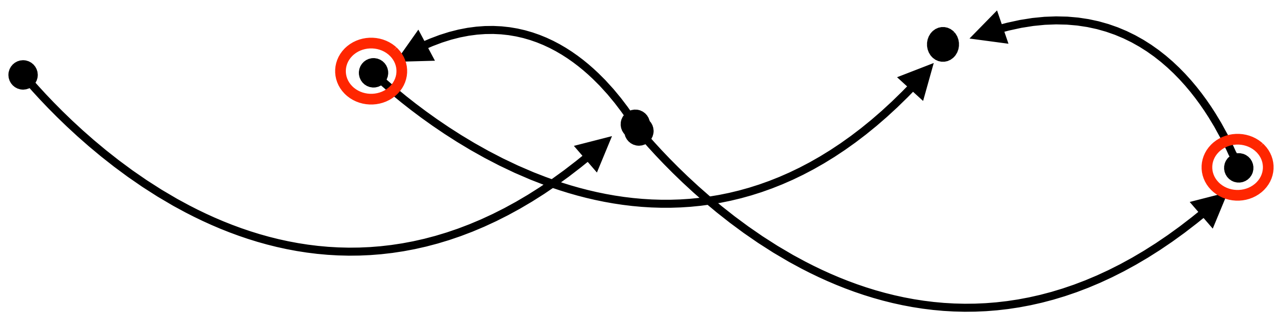
$$\frac{f \circ f = \mathsf{id}}{(h, roots) \xrightarrow{\mathsf{translate}} (\mathsf{rename}\ f\ \ h, \mathsf{map}\ f\ \ roots)}$$

The specification of a full moving GC:

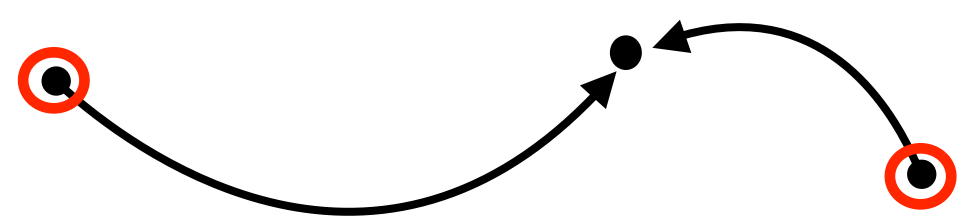
$$x \xrightarrow{\mathsf{gc}} y \quad = \quad (\mathsf{filter}\ x) \xrightarrow{\mathsf{translate}} y$$

Example

Initial heap graph (with roots marked red):



After GC:



Abstract implementation

Next: small-step relation $\xrightarrow{\text{step}}$ for tri-colour moving GC algorithm

Abstract implementation

Next: small-step relation $\xrightarrow{\text{step}}$ for tri-colour moving GC algorithm

State consists of components:

h — the heap, a finite partial mapping,
x — address set: completely processed heap elements,
y — address set: moved elements with pointers to not-yet-moved elements,
z — address set: elements that are still to be moved,
f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

Next: small-step relation $\xrightarrow{\textsf{step}}$ for tri-colour moving GC algorithm

$$\frac{a \in z \ \wedge\ b \notin \textsf{domain}\ h \ \wedge\ f(a) = a \ \wedge\ f(b) = b \ \wedge\ h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h[b \mapsto (as, d)]\lfloor\{a\}^c, x, y \cup \{b\}, z \cup \textsf{set}\ as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \ \wedge\ f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h, x, y, z - \{a\}, f)}$$

$$\frac{b \in y \ \wedge\ h(b) = (as, d) \ \wedge\ \textsf{set}\ as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h[b \mapsto (\textsf{map}\ f\ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

h — the heap, a finite partial mapping,
x — address set: completely processed heap elements,
y — address set: moved elements with pointers to not-yet-moved elements,
z — address set: elements that are still to be moved,
f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

Next: small-step relation $\xrightarrow{\textsf{step}}$ for tri-colour moving GC algorithm

$$\frac{a \in z \ \wedge\ b \notin \textsf{domain}\ h \ \wedge\ f(a) = a \ \wedge\ f(b) = b \ \wedge\ h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h[b \mapsto (as, d)] \lfloor \{a\}^c, x, y \cup \{b\}, z \cup \textsf{set}\ as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \ \wedge\ f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h, x, y, z - \{a\}, f)}$$

$$\frac{b \in y \ \wedge\ h(b) = (as, d) \ \wedge\ \textsf{set}\ as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\textsf{step}} (h[b \mapsto (\textsf{map}\ f\ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

h — the heap, a finite partial mapping,
x — address set: completely processed heap elements,
y — address set: moved elements with pointers to not-yet-moved elements,
z — address set: elements that are still to be moved,
f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

b is unused location

Next: smal … i-colour moving GC algorithm

$$\frac{a \in z \;\wedge\; b \notin \mathsf{domain}\ h \;\wedge\; f(a) = a \;\wedge\; f(b) = b \;\wedge\; h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (as, d)] \restriction \{a\}^c, x, y \cup \{b\}, z \cup \mathsf{set}\ as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \;\wedge\; f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h, x, y, z - \{a\}, f)}$$

$$\frac{b \in y \;\wedge\; h(b) = (as, d) \;\wedge\; \mathsf{set}\ as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (\mathsf{map}\ f\ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

h — the heap, a finite partial mapping,
x — address set: completely processed heap elements,
y — address set: moved elements with pointers to not-yet-moved elements,
z — address set: elements that are still to be moved,
f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

b is unused location

content at *a*

Next: smal … i-colour moving GC algorithm

$$\frac{a \in z \;\wedge\; b \notin \mathsf{domain}\; h \;\wedge\; f(a) = a \;\wedge\; f(b) = b \;\wedge\; h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (as, d)]\lfloor\{a\}^c, x, y \cup \{b\}, z \cup \mathsf{set}\; as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \;\wedge\; f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h, x, y, z - \{a\}, f)}$$

$$\frac{b \in y \;\wedge\; h(b) = (as, d) \;\wedge\; \mathsf{set}\; as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (\mathsf{map}\; f\; as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

- h — the heap, a finite partial mapping,
- x — address set: completely processed heap elements,
- y — address set: moved elements with pointers to not-yet-moved elements,
- z — address set: elements that are still to be moved,
- f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

b is unused location

content at *a*

swap

Next: smal … i-colour moving … rithm

$$\frac{a \in z \;\wedge\; b \notin \mathsf{domain}\; h \;\wedge\; f(a) = a \;\wedge\; f(b) = b \;\wedge\; h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (as, d)]\restriction\{a\}^c, x, y \cup \{b\}, z \cup \mathsf{set}\; as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \;\wedge\; f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h, x, y, z - \{a\}, f)}$$

$$\frac{b \in y \;\wedge\; h(b) = (as, d) \;\wedge\; \mathsf{set}\; as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (\mathsf{map}\; f\; as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

- h — the heap, a finite partial mapping,
- x — address set: completely processed heap elements,
- y — address set: moved elements with pointers to not-yet-moved elements,
- z — address set: elements that are still to be moved,
- f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

b is unused location

content at *a*

swap

Next: smal … i-colour moving … rithm

$$\frac{a \in z \ \wedge\ b \notin \mathsf{domain}\ h \ \wedge\ f(a) = a \ \wedge\ f(b) = b \ \wedge\ h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (as, d)]\lfloor\{a\}^c, x, y \cup \{b\}, z \cup \mathsf{set}\ as, f[a \mapsto b][b \mapsto a])}$$

$$\frac{a \in z \ \wedge\ f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h, x, y, z - \{a\}, f)}$$

already processed

$$\frac{b \in y \ \wedge\ h(b) = (as, d) \ \wedge\ \mathsf{set}\ as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (\mathsf{map}\ f\ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

- h — the heap, a finite partial mapping,
- x — address set: completely processed heap elements,
- y — address set: moved elements with pointers to not-yet-moved elements,
- z — address set: elements that are still to be moved,
- f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Abstract implementation

a needs to be moved

b is unused location

content at *a*

swap

Next: smal… …i-colour moving … …rithm

$$\frac{a \in z \;\wedge\; b \notin \mathsf{domain}\ h \;\wedge\; f(a) = a \;\wedge\; f(b) = b \;\wedge\; h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (as, d)]\lfloor\{a\}^c, x, y \cup \{b\}, z \cup \mathsf{set}\ as, f[a \mapsto b][b \mapsto a])}$$

already processed

$$\frac{a \in z \;\wedge\; f(a) \neq a}{(h, x, \ldots}$$

finalise heap cell

$$\frac{b \in y \;\wedge\; h(b) = (as, d) \;\wedge\; \mathsf{set}\ as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\mathsf{step}} (h[b \mapsto (\mathsf{map}\ f\ as, d)], x \cup \{b\}, y - \{b\}, z, f)}$$

State consists of components:

h — the heap, a finite partial mapping,
x — address set: completely processed heap elements,
y — address set: moved elements with pointers to not-yet-moved elements,
z — address set: elements that are still to be moved,
f — a function which records where elements have been moved: $\mathbb{N} \to \mathbb{N}$

Correctness

Correctness theorem:

$$\forall h\ h_2\ roots\ x\ f.$$
$$(h, \{\}, \{\}, \mathsf{set}\ roots, \mathsf{id}) \xrightarrow{\mathsf{step}}{}^* (h_2, x, \{\}, \{\}, f) \ \wedge\ \mathsf{ok\_heap}\ (h, roots) \implies$$
$$(h, roots) \xrightarrow{\mathsf{gc}} (h_2 | x, \mathsf{map}\ f\ roots)$$

$$\text{where}\quad \mathsf{ok\_heap}\ (h, roots) \ = \ \mathsf{pointers}\ h \cup \mathsf{set}\ roots \subseteq \mathsf{domain}\ h$$
$$\mathsf{pointers}\ h \ = \ \{\ x \mid \exists a\ as\ d.\ x \in \mathsf{set}\ as \wedge h(a) = (as, d)\ \}$$

Proof: we prove that an invariant is maintained

$$\forall x\ s\ t.\ \ \mathsf{inv}\ x\ s\ \wedge\ s \xrightarrow{\mathsf{step}} t \implies \mathsf{inv}\ x\ t$$

and sufficient. Invariant on next slide...

Correctness

starts off with roots to-be moved

Correctness theorem:

$$\forall h\ h_2\ roots\ x\ f.$$
$$(h, \{\}, \{\}, \mathsf{set}\ roots, \mathsf{id}) \xrightarrow{\mathsf{step}}{}^{*} (h_2, x, \{\}, \{\}, f) \ \wedge\ \mathsf{ok\_heap}\ (h, roots) \implies$$
$$(h, roots) \xrightarrow{\mathsf{gc}} (h_2 \lfloor x, \mathsf{map}\ f\ roots)$$

$$\text{where}\quad \mathsf{ok\_heap}\ (h, roots) \ = \ \mathsf{pointers}\ h \cup \mathsf{set}\ roots \subseteq \mathsf{domain}\ h$$
$$\mathsf{pointers}\ h \ = \ \{\ x \mid \exists a\ as\ d.\ x \in \mathsf{set}\ as \wedge h(a) = (as, d)\ \}$$

Proof: we prove that an invariant is maintained

$$\forall x\ s\ t.\ \ \mathsf{inv}\ x\ s\ \wedge\ s \xrightarrow{\mathsf{step}} t \implies \mathsf{inv}\ x\ t$$

and sufficient. Invariant on next slide...

Correctness

starts off with roots to-be moved

Correctness theorem:

terminates when nothing left to-do

$$\forall h\ h_2\ roots\ x\ f.$$
$$(h, \{\}, \{\}, \mathsf{set}\ roots, \mathsf{id}) \xrightarrow{\mathsf{step}}{}^{*}\ (h_2, x, \{\}, \{\}, f)\ \wedge\ \mathsf{ok\_heap}\ (h, roots)\ \Longrightarrow$$
$$(h, roots) \xrightarrow{\mathsf{gc}} (h_2 \lfloor x, \mathsf{map}\ f\ roots)$$

$$\text{where}\quad \mathsf{ok\_heap}\ (h, roots)\ =\ \mathsf{pointers}\ h \cup \mathsf{set}\ roots \subseteq \mathsf{domain}\ h$$
$$\mathsf{pointers}\ h\ =\ \{\ x \mid \exists a\ as\ d.\ x \in \mathsf{set}\ as \wedge h(a) = (as, d)\ \}$$

Proof: we prove that an invariant is maintained

$$\forall x\ s\ t.\ \ \mathsf{inv}\ x\ s\ \wedge\ s \xrightarrow{\mathsf{step}} t\ \Longrightarrow\ \mathsf{inv}\ x\ t$$

and sufficient. Invariant on next slide...

Correctness

starts off with roots to-be moved

Correctness theorem:

terminates when nothing left to-do

$$\forall h\ h_2\ roots\ x\ f.$$
$$(h, \{\}, \{\}, \textsf{set}\ roots, \textsf{id}) \xrightarrow{\textsf{step}}{}^{*} (h_2, x, \{\}, \{\}, f)\ \wedge\ \textsf{ok\_heap}\ (h, roots) \implies$$
$$(h, roots) \xrightarrow{\textsf{gc}} (h_2{\downharpoonright}x, \textsf{map}\ f\ roots)$$

$$\begin{aligned} \text{where}\quad \textsf{ok\_heap}\ (h, roots) &= \textsf{pointers}\ h \cup \textsf{set}\ roots \subseteq \textsf{domain}\ h \\ \textsf{pointers}\ h &= \{\ x \mid \exists a\ as\ d.\ x \in \textsf{set}\ as \wedge h(a) = (as, d)\ \} \end{aligned}$$

any such terminating execution is a valid GC execution

Proof: we prove that an invariant is maintained

$$\forall x\ s\ t.\ \ \textsf{inv}\ x\ s\ \wedge\ s \xrightarrow{\textsf{step}} t \implies \textsf{inv}\ x\ t$$

and sufficient. Invariant on next slide...

Invariant

The lengthy invariant:

$$
\begin{array}{rl}
 & \textsf{inv}\ (h_0, roots)\ (h, x, y, z, f) = \\
0 & \quad \textsf{let}\ old = (\textsf{domain}\ h \cup \{\ a \mid f(a) \neq a\ \}) - (x \cup y)\ \textsf{in} \\
1 & \qquad (x \cap y = \{\}) \wedge (f \circ f = \textsf{id}) \wedge \\
2 & \qquad \textsf{pointers}\ (h\lfloor x) \subseteq x \cup y\ \wedge \\
3 & \qquad \textsf{pointers}\ (h\lfloor x^c) \subseteq old\ \wedge \\
4 & \qquad \textsf{pointers}\ (h\lfloor y) \cup \textsf{set}\ roots \subseteq \textsf{image}\ f\ (x \cup y) \cup z \subseteq \textsf{reach}\ (h_0, roots)\ \wedge \\
5 & \qquad (\forall a.\ a \in z \implies \textsf{if}\ f(a) = a\ \textsf{then}\ a \in old\ \textsf{else}\ f(a) \in x \cup y) \wedge \\
6 & \qquad (\forall a.\ f(a) \neq a \implies \neg(a \in x \cup y \iff f(a) \in x \cup y)) \wedge \\
7 & \qquad (\forall a.\ a \in x \cup y \iff f(a) \neq a \wedge a \in \textsf{domain}\ h) \wedge \\
8 & \qquad \textsf{domain}\ h = \textsf{image}\ f\ (\textsf{domain}\ h_0) \wedge \\
9 & \qquad (\forall a\ as\ d.\ f(a) \in \textsf{domain}\ h \wedge h(f(a)) = (as, d) \implies \\
 & \qquad\qquad\qquad h_0(a) = \textsf{if}\ f(a) \in x\ \textsf{then}\ (\textsf{map}\ f\ as, d)\ \textsf{else}\ (as, d))
\end{array}
$$

Most effort is spent finding the invariant.

First-order prover can automate much of the proof.

Implementation with memory

Next refinement introduces an abstract memory.

Memory consists of

$$
\begin{array}{lll}
\text{Block } (as, l, d) & \text{---} & \text{block of data, e.g. a cons-cell} \\
\text{Ref } a & \text{---} & \text{record of where data has moved} \\
\text{Emp} & \text{---} & \text{empty or `don't care'}
\end{array}
$$

Relation to small-step relation's state:

$$
\begin{array}{ll}
m(a) = \text{Block } (h(a)) & \text{if } a \in \text{domain } h \\
m(a) = \text{Ref } (f(a)) & \text{if } a \notin \text{domain } h \text{ and } f(a) \neq a \\
m(a) = \text{Emp} & \text{if } a \notin \text{domain } h \text{ and } f(a) = a
\end{array}
$$

Implementation with memory

move (RHS n, j, m) $=$ (RHS n, j, m)
move (LHS a, j, m) $=$ case $m(a)$ of
  Ref $i \rightarrow$ (LHS i, j, m)
  $|$ Block $(as, l, d) \rightarrow$
    let $m = m[a \mapsto$ Ref $j]$ in
    let $m = m[j \mapsto$ Block $(as, l, d)]$ in
      (LHS $j, j + l + 1, m$)

move_list $([], j, m) = ([], j, m)$
move_list $(r{::}rs, j, m) =$
  let $(r, j, m) =$ move (r, j, m) in
  let $(rs, j, m) =$ move_list (rs, j, m) in
    $(r{::}rs, j, m)$

readBlock (Block x) $= x$
cut $(i, j)\ m\ =\ \lambda k.$ if $i \leq k \wedge k < j$ then $m\,k$ else Emp

loop $(i, j, m)\ =$
  if $i = j$ then (i, m) else
    let $(as, l, d) =$ readBlock $(m\ i)$ in
    let $(as, j, m) =$ move_list (as, j, m) in
    let $m = m[i \mapsto$ Block $(as, l, d)]$ in
      loop $(i + l + 1, j, m)$

collector $(roots, b, i, e, b_2, e_2, m)\ =$
  let $(b_2, e_2, b, e) = (b, e, b_2, e_2)$ in
  let $(roots, j, m) =$ move_list $(roots, b, m)$ in
  let $(i, m) =$ loop (b, j, m) in
  let $m =$ cut $(b, i)\ m$ in
    $(roots, b, i, e, b_2, e_2, m)$

Correctness

Correctness theorem:

$$
\begin{array}{l}
\forall h\ roots\ roots_2\ x\ y. \\
\quad \textsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \textsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow \\
\quad \exists h_2.\ \textsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\textsf{gc}} (h_2, roots_2)
\end{array}
$$

Correctness

if abstract state is correctly represented, then ...

Correctness theorem:

$$
\begin{aligned}
&\forall h\ roots\ roots_2\ x\ y.\\
&\quad \mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow\\
&\quad \exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)
\end{aligned}
$$

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$
\begin{array}{l}
\forall h\ roots\ roots_2\ x\ y.\\
\quad \mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow\\
\quad \exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)
\end{array}
$$

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$\forall h\ roots\ roots_2\ x\ y.$$
$$\mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow$$
$$\exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)$$

some new abstract heap exists such that ...

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$\begin{array}{l}\forall h\ roots\ roots_2\ x\ y.\\ \quad \mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow\\ \quad \exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)\end{array}$$

some new abstract heap exists such that ...

specification is met

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$
\begin{array}{l}
\forall h\ roots\ roots_2\ x\ y.\\
\quad \textsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \textsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow\\
\quad \exists h_2.\ \textsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\textsf{gc}} (h_2, roots_2)
\end{array}
$$

some new abstract heap exists such that ...

specification is met

Proof: again uses a lengthy invariant

$$
\begin{array}{l}
\textsf{mem\_inv}\ (h_0, roots_0, h, f)\ (b, i, j, e, b_2, e_2, m, z) =\\
\quad b \le i \le j \le e \wedge (e < b_2 \vee e_2 < b)\ \wedge\\
\quad (\forall a.\ a \notin b_2 ... e_2 \cup b ... j\ \Longrightarrow\ m(a) = \textsf{Emp})\ \wedge\\
\quad \textsf{part\_heap}\ (b, i)\ m\ (i - b) \wedge \textsf{part\_heap}\ (i, j)\ m\ (j - i)\ \wedge\\
\quad (\exists k.\ \textsf{part\_heap}\ (b_2, e_2)\ m\ k \wedge k \le e - j)\ \wedge\\
\quad \textsf{ref\_mem}\ (h, f)\ m \wedge \textsf{ok\_heap}\ (h_0, roots_0)\ \wedge\\
\quad (h_0, \{\}, \{\}, \textsf{set}\ roots_0, \textsf{id}) \xrightarrow{\textsf{step}}{*}\ (h, \textsf{domain}\ h \cap (b ... i), \textsf{domain}\ h \cap (i ... j), z, f)
\end{array}
$$

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$\forall h\ roots\ roots_2\ x\ y.$

$\quad \mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow$

$\quad \exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)$

some new abstract heap exists such that ...

specification is met

Proof: again uses a lengthy invariant

$\mathsf{mem\_inv}\ (h_0, roots_0, h, f)\ (b, i, j, e, b_2, e_2, m, z) =$

$\quad b \le i \le j \le e \wedge (e < b_2 \vee e_2 < b)\ \wedge$

$\quad (\forall a.\ a \notin b_2 ... e_2 \cup b ... j\ \Longrightarrow\ m(a) = \mathsf{Emp})\ \wedge$

$\quad \mathsf{part\_heap}\ (b, i)\ m\ (i - b) \wedge \mathsf{part\_heap}\ (i, j)\ m$

$\quad (\exists k.\ \mathsf{part\_heap}\ (b_2, e_2)\ m\ k \wedge k \le e - j)\ \wedge$

$\quad \mathsf{ref\_mem}\ (h, f)\ m \wedge \mathsf{ok\_heap}\ (h_0, roots_0)\ \wedge$

$\quad (h_0, \{\}, \{\}, \mathsf{set}\ roots_0, \mathsf{id}) \xrightarrow{\mathsf{step}} {*}\ (h, \mathsf{domain}\ h \cap (b...i), \mathsf{domain}\ h \cap (i...j), z, f)$

most is trivial book-keeping
(where/how state is repr.)

Correctness

if abstract state is correctly represented, then ...

Correctness theorem

for any execution of implementation...

$$\forall h\ roots\ roots_2\ x\ y.$$
$$\mathsf{ok\_mem\_heap}\ (h, roots)\ x\ \wedge\ \mathsf{collector}\ (roots, x) = (roots_2, y)\ \Longrightarrow$$
$$\exists h_2.\ \mathsf{ok\_mem\_heap}\ (h_2, roots_2)\ y\ \wedge\ (h, roots) \xrightarrow{\mathsf{gc}} (h_2, roots_2)$$

some new abstract heap exists such that ...

specification is met

Proof: again uses a lengthy invariant

$$\mathsf{mem\_inv}\ (h_0, roots_0, h, f)\ (b, i, j, e, b_2, e_2, m, z) =$$
$$b \le i \le j \le e \wedge (e < b_2 \vee e_2 < b)\ \wedge$$
$$(\forall a.\ a \notin b_2 ... e_2 \cup b ... j\ \Longrightarrow\ m(a) = \mathsf{Emp})\ \wedge$$
$$\mathsf{part\_heap}\ (b, i)\ m\ (i - b) \wedge \mathsf{part\_heap}\ (i, j)\ m$$
$$(\exists k.\ \mathsf{part\_heap}\ (b_2, e_2)\ m\ k \wedge k \le e - j)\ \wedge$$
$$\mathsf{ref\_mem}\ (h, f)\ m \wedge \mathsf{ok\_heap}\ (h_0, roots_0)\ \wedge$$
$$(h_0, \{\}, \{\}, \mathsf{set}\ roots_0, \mathsf{id}) \xrightarrow{\mathsf{step}}{*}\ (h, \mathsf{domain}\ h \cap (b...i), \mathsf{domain}\ h \cap (i...j), z, f)$$

most is trivial book-keeping
(where/how state is repr.)

reason for correctness inherited

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Decompilation produces functions, e.g.

$$
\begin{array}{l}
\text{mc\_move\_loop}\ (r_2, r_3, r_4, g) = \\
\quad \text{if } r_4 = 0 \text{ then } (r_2, r_3, r_4, g) \text{ else} \\
\quad\quad \text{let } r_5 = g(r_2) \text{ in} \\
\quad\quad \text{let } r_4 = r_4 - 1 \text{ in} \\
\quad\quad \text{let } r_2 = r_2 + 4 \text{ in} \\
\quad\quad \text{let } g = g[r_3 \mapsto r_5] \text{ in} \\
\quad\quad \text{let } r_3 = r_3 + 4 \text{ in} \\
\quad\quad\quad \text{mc\_move\_loop}\ (r_2, r_3, r_4, g)
\end{array}
$$

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Decompilation produces functions, e.g.

mc_move_loop $(r_2, r_3, r_4, g) =$
  if $r_4 = 0$ then (r_2, r_3, r_4, g) else
    let $r_5 = g(r_2)$ in
    let $r_4 = r_4 - 1$ in
    let $r_2 = r_2 + 4$ in
    let $g = g[r_3 \mapsto r_5]$ in
    let $r_3 = r_3 + 4$ in
      mc_move_loop (r_2, r_3, r_4, g)

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Carefully written code for other architectures (x86, PowerPC etc.) decompiles to the same function in logic.

Decompilation produces functions, e.g.

$$
\begin{array}{l}
\text{mc\_move\_loop}\ (r_2, r_3, r_4, g) = \\
\quad \text{if } r_4 = 0 \text{ then } (r_2, r_3, r_4, g) \text{ else} \\
\quad\quad \text{let } r_5 = g(r_2) \text{ in} \\
\quad\quad \text{let } r_4 = r_4 - 1 \text{ in} \\
\quad\quad \text{let } r_2 = r_2 + 4 \text{ in} \\
\quad\quad \text{let } g = g[r_3 \mapsto r_5] \text{ in} \\
\quad\quad \text{let } r_3 = r_3 + 4 \text{ in} \\
\quad\quad\quad \text{mc\_move\_loop}\ (r_2, r_3, r_4, g)
\end{array}
$$

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

Carefully written code for other architectures (x86, PowerPC etc.) decompiles to the same function in logic. Proof reuse!

Decompilation produces functions, e.g.

$$
\begin{array}{l}
\text{mc\_move\_loop}\ (r_2, r_3, r_4, g) = \\
\quad \text{if } r_4 = 0 \text{ then } (r_2, r_3, r_4, g) \text{ else} \\
\qquad \text{let } r_5 = g(r_2) \text{ in} \\
\qquad \text{let } r_4 = r_4 - 1 \text{ in} \\
\qquad \text{let } r_2 = r_2 + 4 \text{ in} \\
\qquad \text{let } g = g[r_3 \mapsto r_5] \text{ in} \\
\qquad \text{let } r_3 = r_3 + 4 \text{ in} \\
\qquad\quad \text{mc\_move\_loop}\ (r_2, r_3, r_4, g)
\end{array}
$$

Some assembly code

ARM

```
    tst r2, #3
    bne L0
    ldr r4, [r2]
    tst r4, #3
    streq r4, [r1]
    beq L0
    str r3, [r1]
    str r4, [r3]
    str r3, [r2], #4
    mov r4, r4, LSR #10
    add r3, r3, #4
L1: cmp r4, #0
    beq L0
    ldr r5, [r2]
    sub r4, r4, #1
    add r2, r2, #4
    str r5, [r3]
    add r3, r3, #4
    b L1
L0:
```

x86

```
    test ecx, 3
    jne L0
    mov ebx, [ecx]
    test ebx, 3
    jne L2
    mov [eax], ebx
    jmp L0
L2: mov [eax], edx
    mov [edx], ebx
    mov [ecx], edx
    shr ebx, 10
    add edx, 4
    add ecx, 4
L1: cmp ebx, 0
    je L0
    mov edi, [ecx]
    dec ebx
    add ecx, 4
    mov [edx], edi
    add edx, 4
    jmp L1
L0:
```

PowerPC

```
    andi. 0, 2, 3
    bne L0
    lwz 4, 0(2)
    andi. 0, 4, 3
    bne L2
    stw 4, 0(1)
    b L0
L2: stw 3, 0(1)
    stw 4, 0(3)
    stw 3, 0(2)
    srawi 4, 4, 10
    addi 3, 3, 4
    addi 2, 2, 4
L1: cmplwi 4,0
    beq L0
    lwz 5, 0(2)
    addi 4, 4, -1
    addi 2, 2, 4
    stw 5, 0(3)
    addi 3, 3, 4
    b L1
L0:
```

Proving final connection

Correctness theorem:

$$\forall x\ y\ z.\ \ \mathsf{ok\_mc\_heap}\ x\ y\ z \implies \mathsf{ok\_mc\_heap}\ x\ (\mathsf{collector}\ y)\ (\mathsf{mc\_collector}\ z)$$

Proving final connection

Correctness theorem:

$$\forall x\ y\ z.\ \ \mathsf{ok\_mc\_heap}\ x\ y\ z \implies \mathsf{ok\_mc\_heap}\ x\ (\mathsf{collector}\ y)\ (\mathsf{mc\_collector}\ z)$$

relation between abstract memory and concrete memory

Proving final connection

abstract memory impl.

Correctness theorem:

$$\forall x\ y\ z.\ \ \text{ok\_mc\_heap}\ x\ y\ z \implies \text{ok\_mc\_heap}\ x\ (\text{collector}\ y)\ (\text{mc\_collector}\ z)$$

relation between abstract memory and concrete memory

Proving final connection

abstract memory impl.

decompiler extracted functions

Correctness theorem:

$$\forall x\ y\ z.\ \ \text{ok\_mc\_heap}\ x\ y\ z \implies \text{ok\_mc\_heap}\ x\ (\text{collector}\ y)\ (\text{mc\_collector}\ z)$$

relation between abstract memory and concrete memory

Proving final connection

abstract memory impl.

decompiler extracted functions

Correctness theorem:

$$\forall x\ y\ z.\ \ \text{ok\_mc\_heap}\ x\ y\ z \implies \text{ok\_mc\_heap}\ x\ (\text{collector}\ y)\ (\text{mc\_collector}\ z)$$

relation between abstract memory and concrete memory

relation maintained

Proving final connection

abstract memory impl.

Correctness theorem:

decompiler extracted functions

$$\forall x\ y\ z.\ \ \mathsf{ok\_mc\_heap}\ x\ y\ z \implies \mathsf{ok\_mc\_heap}\ x\ (\mathsf{collector}\ y)\ (\mathsf{mc\_collector}\ z)$$

relation between abstract memory and concrete memory

relation maintained

The ok_mc_heap relation:

- specifies the exact layout in machine memory
- separation logic notation used for brevity
- lengthy definition omitted

Result: memory abstraction

A high-level theorem about the machine code for GC:

```
{ HEAP abs_state * PC pc }
  “ entire GC implementation with entry point pc ”
{ HEAP abs_state * PC (pc + length_of_gc_impl) }
```

where `HEAP abs_state = ∃m regs. MEM m * ... *`
`pure (high_low_rel abs_state m regs)`

GC implementation:

- always terminates
- maintains the memory abstraction
- is transparent: no visible change in high-level view of state
- even though all addresses renamed

Summary

Garbage collection

- reclaims unused memory
- automatic memory management
- part of memory abstraction
- copying collection moves data

Summary

Garbage collection

- reclaims unused memory
- automatic memory management
- part of memory abstraction
- copying collection moves data

Non-trivial verification

Summary

Garbage collection

- reclaims unused memory
- automatic memory management
- part of memory abstraction
- copying collection moves data

Non-trivial verification

Common pitfall:

Attempt to verify complex implementation at low level of abstraction.

Summary

Garbage collection

- reclaims unused memory
- automatic memory management
- part of memory abstraction
- copying collection moves data

Common pitfall:

Attempt to verify complex implementation at low level of abstraction.

Non-trivial verification

- best split into separate layers of abstraction
- proof by (data-)refinement separates
- high-level algorithm proof from
- low-level implementation proof

Summary

Garbage collection

- reclaims unused memory
- automatic memory management
- part of memory abstraction
- copying collection moves data

Non-trivial verification

Common pitfall:
Attempt to verify complex implementation at low level of abstraction.

- best split into separate layers of abstraction
- proof by (data-)refinement separates
- high-level algorithm proof from
- low-level implementation proof

Tip: powerful proof automation can be used
if problem is phrased suitably