Verifying a Concurrent Garbage Collector using a Rely-Guarantee Methodology

Yannick Zakowski
David Cachera
Delphine Demange
Gustavo Petri
David Pichardie
Suresh Jagannathan
Jan Vitek

ITP'17

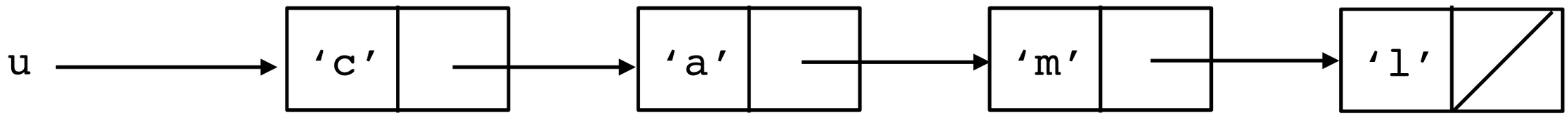
Automatic memory management

Dynamic allocation of memory (in the heap)

In C: `int * array = malloc(10 * sizeof(int))`

In Java: `Point originOne = new Point(23, 94)`

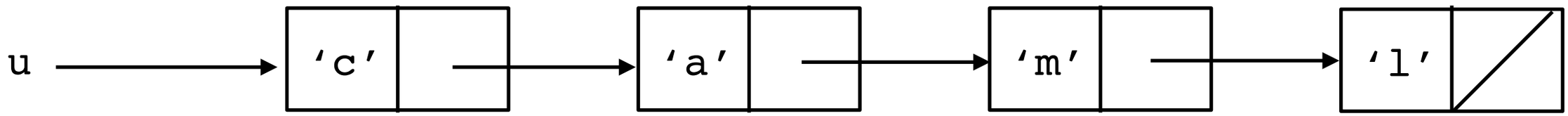
In OCaml: `let u = 'c'::'a'::'m'::'l'::[]`



Automatic memory management

Dynamic allocation of memory (in the heap)

In C: `int * array = malloc(10 * sizeof(int))`
In Java: `Point originOne = new Point(23, 94)`
In OCaml: `let u = 'c'::'a'::'m'::'l'::[]`



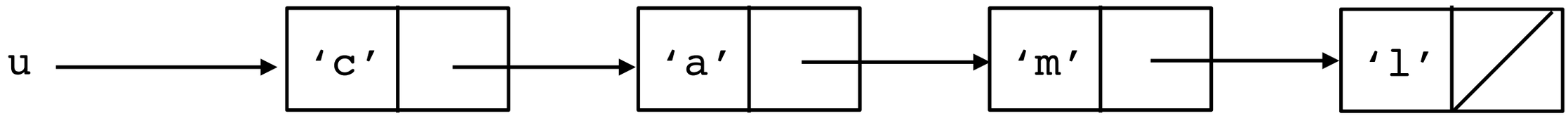
Manual memory management

Programmer responsible for deallocation (C, C++…)
Risks: premature/double free, memory leak

Automatic memory management

Dynamic allocation of memory (in the heap)

In C: `int * array = malloc(10 * sizeof(int))`
In Java: `Point originOne = new Point(23, 94)`
In OCaml: `let u = 'c'::'a'::'m'::'l'::[]`



Manual memory management

Programmer responsible for deallocation (C, C++…)
Risks: premature/double free, memory leak

Automatic memory management

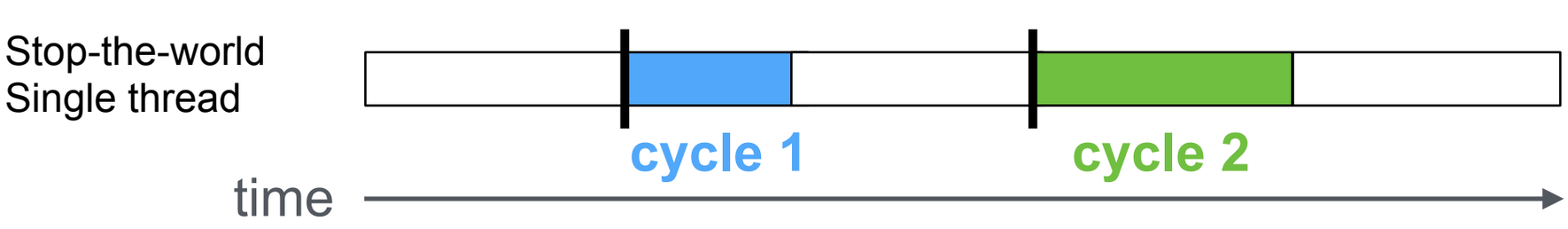
Memory reclaimed automatically: Garbage Collector (Lisp, OCaml, Java…)

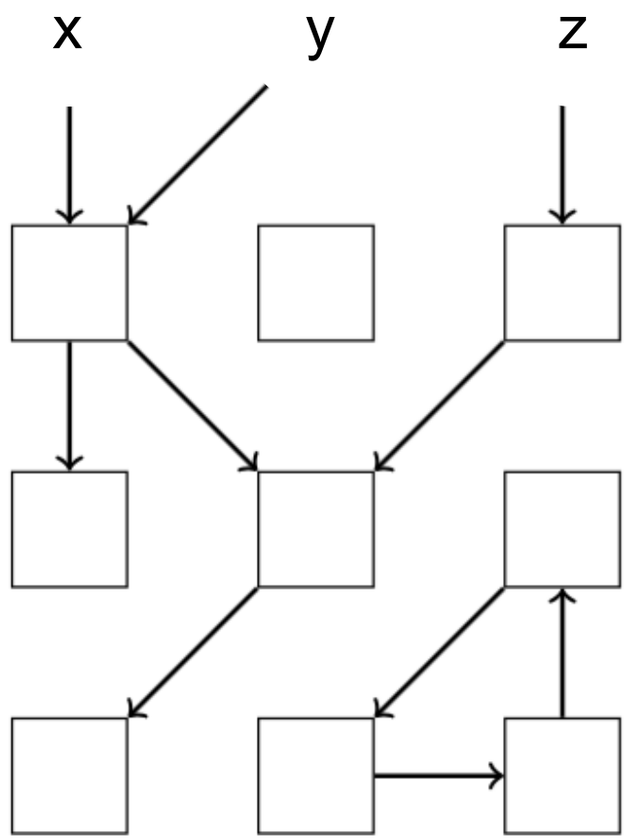
Garbage Collection

Sequential Mark-Sweep Collectors (McCarthy, 1960)

On allocation, if few memory available:

1. Stop the user program
2. Perform full cycle: unreachable memory is reclaimed



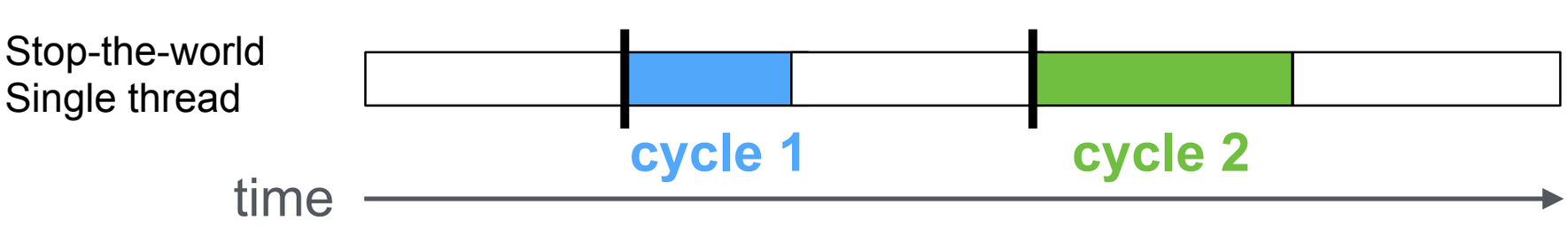


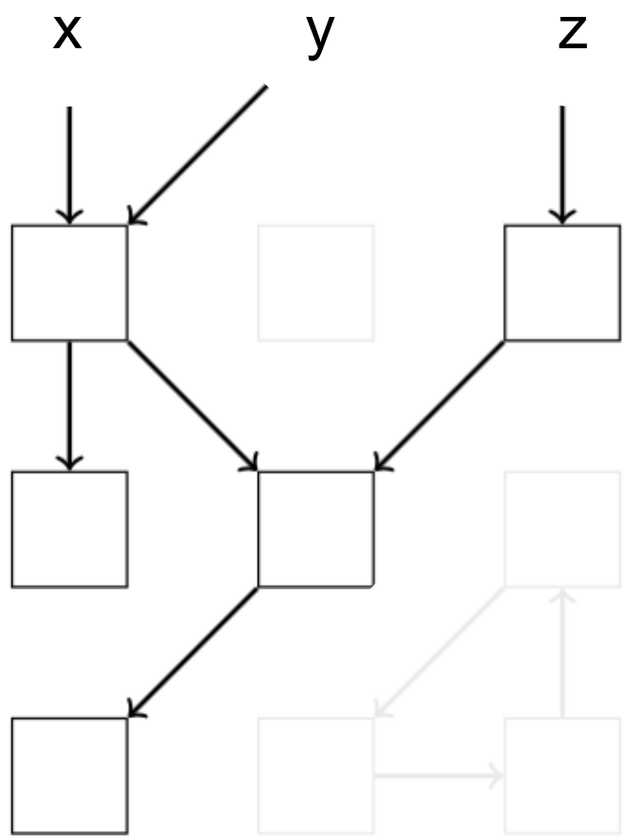
Garbage Collection

Sequential Mark-Sweep Collectors (McCarthy, 1960)

On allocation, if few memory available:

1. Stop the user program
2. Perform full cycle: unreachable memory is reclaimed



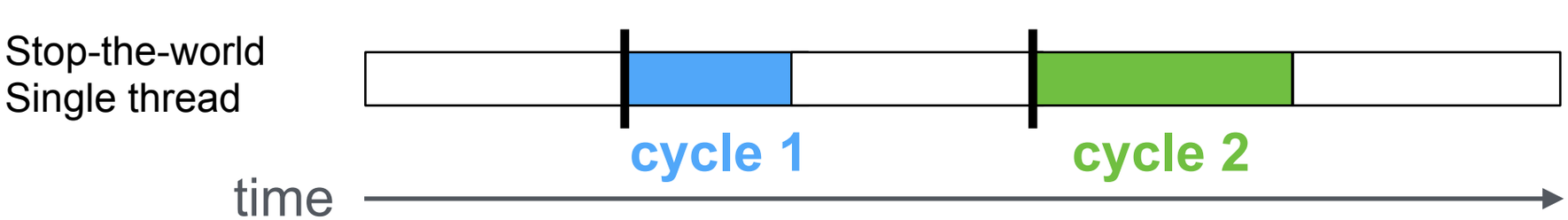


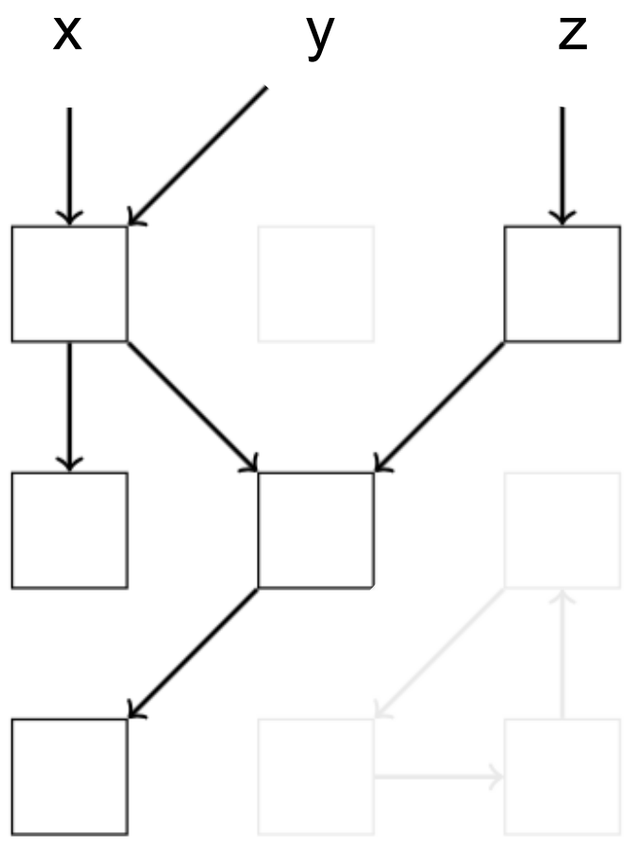
Garbage Collection

Sequential Mark-Sweep Collectors (McCarthy, 1960)

On allocation, if few memory available:

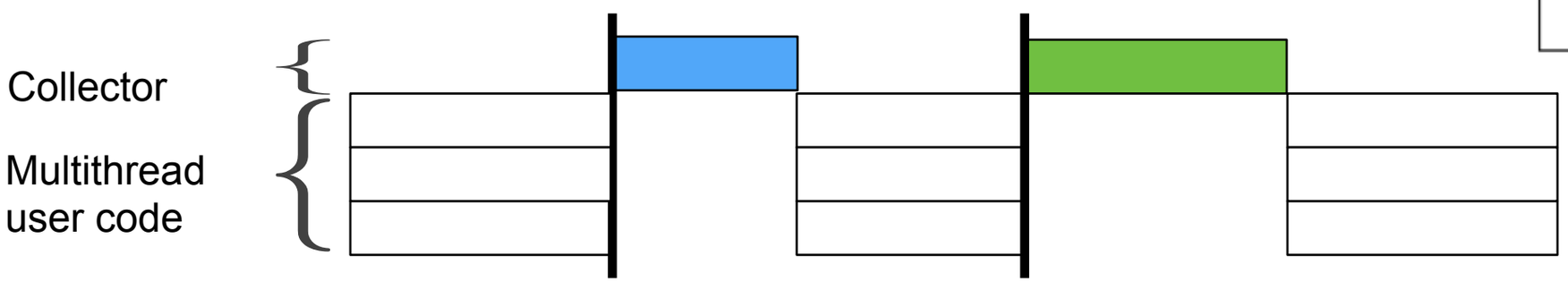
1. Stop the user program
2. Perform full cycle: unreachable memory is reclaimed





With concurrency

Stop-the-world?

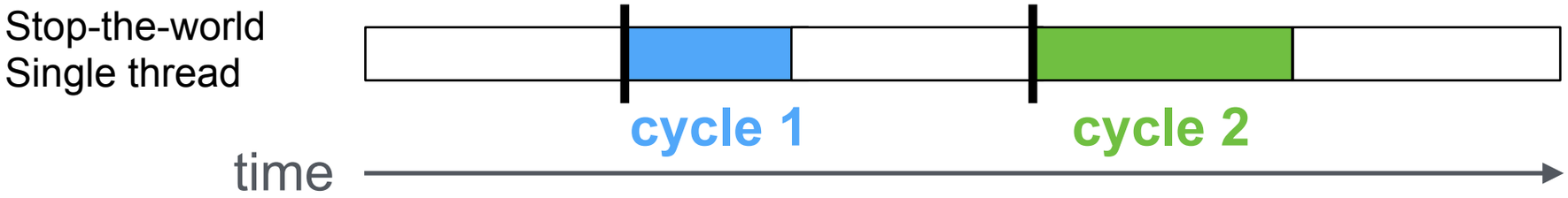


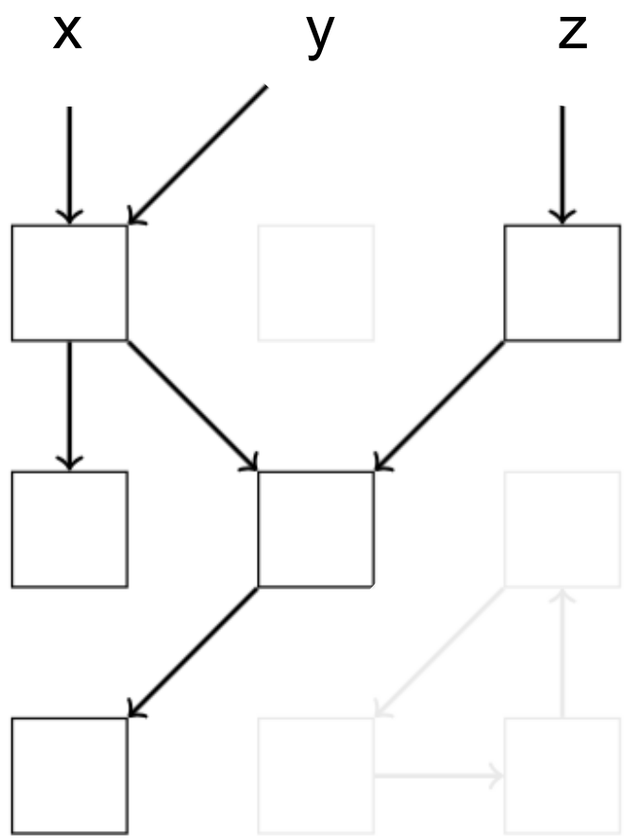
Garbage Collection

Sequential Mark-Sweep Collectors (McCarthy, 1960)

On allocation, if few memory available:

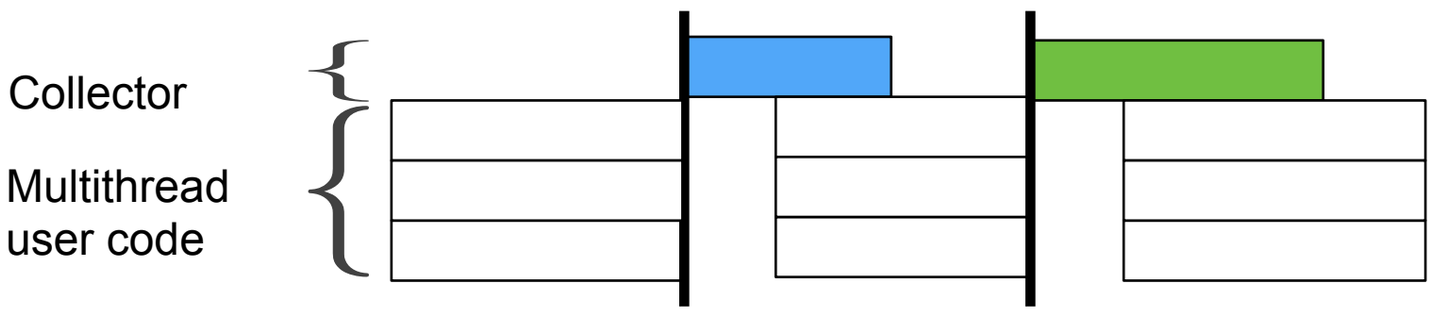
1. Stop the user program
2. Perform full cycle: unreachable memory is reclaimed





With concurrency

Mostly-concurrent?

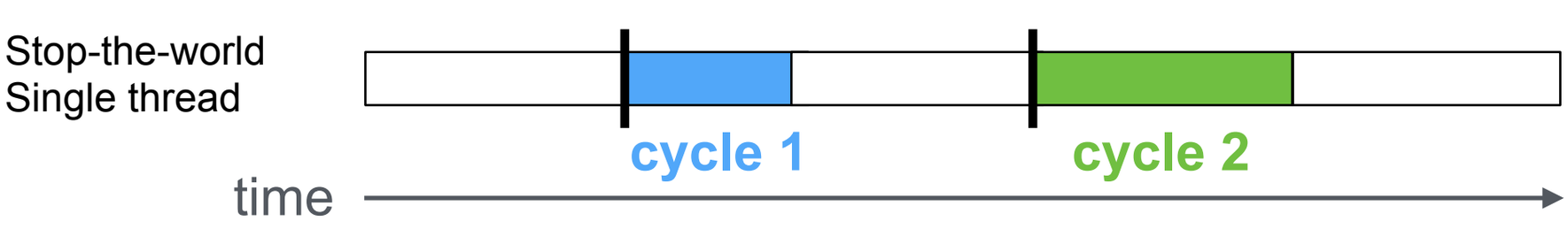


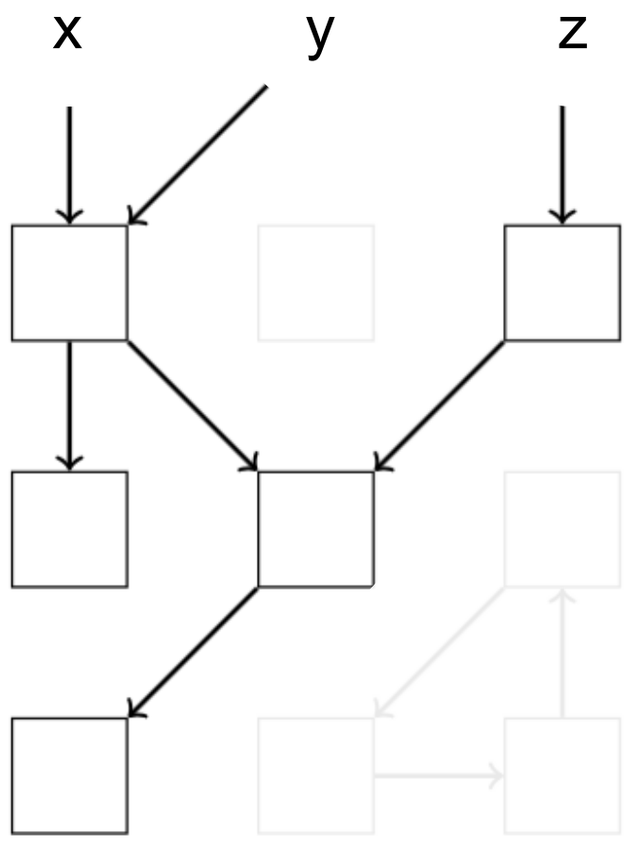
Garbage Collection

Sequential Mark-Sweep Collectors (McCarthy, 1960)

On allocation, if few memory available:

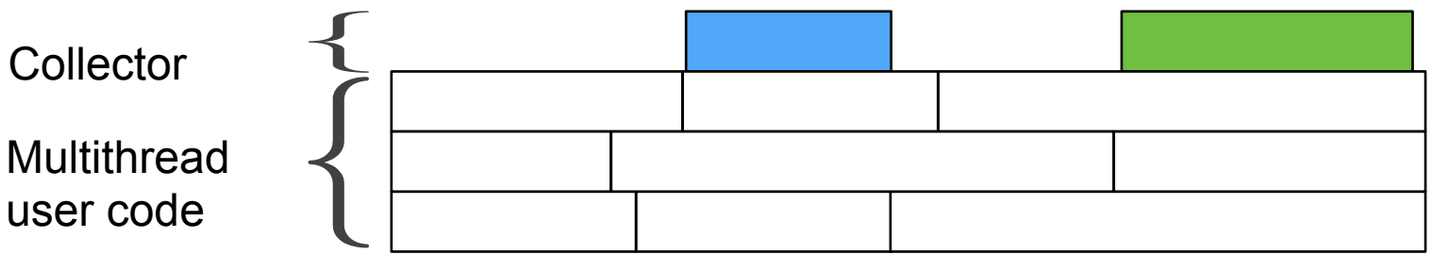
1. Stop the user program
2. Perform full cycle: unreachable memory is reclaimed





With concurrency

On-the-fly!



On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

A.J. Martin, C.S. Scholten, and
E.F.M. Steffens
Philips Research Laboratories

As an example of cooperation between sequential processes with very little mutual interference despite

On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

POPL ‘93 ‘94

A concurrent, generational garbage collector for a multithreaded implementation of ML

Damien Doligez Xavier Leroy

Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

Damien Doligez
École Normale Supérieure
INRIA Rocquencourt
École Polytechnique
Damien.Doligez@inria.fr

Georges Gonthier*
INRIA Rocquencourt
78153 LE CHESNAY CEDEX
FRANCE
Georges.Gonthier@inria.fr

On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

POPL ‘93 ‘94

A concurrent, generational garbage collector for a multithreaded implementation of ML

Damien Doligez Xavier Leroy

Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

Damien Doligez Georges Gonthier*

the basic algorithm of [9], and expose a series of counterexamples to explain why a straightforward adaptation of this algorithm to multiple mutators would not work; we also address some efficiency issues. In section 4 we describe the

On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

POPL ‘93 ‘94

A concurrent, generational garbage collector for a multithreaded implementation of ML

Damien Doligez Xavier Leroy

Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

Damien Doligez Georges Gonthier*

the basic algorithm of [9], and expose a series of counterexamples to explain why a straightforward adaptation of this algorithm to multiple mutators would not work; we also address some efficiency issues. In section 4 we describe the

PLDI ‘00

Implementing an On-the-fly Garbage Collector for Java

Tamar Domani Elliot K. Kolodner* Ethan Lewis Eliot E. Salant
Katherine Barabash Itai Lahan Erez Petrank Igor Yanover
Yossi Levanoni

Abstract

introduced for LISP in 1960. Since then, garbage collection has been adapted for many

On-the-fly garbage collection

CACM ‘78

On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsger W. Dijkstra
Burroughs Corporation

Leslie Lamport
SRI International

technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.

POPL ‘93 ‘94

A concurrent, generational garbage collector for a multithreaded implementation of ML

Damien Doligez Xavier Leroy

Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

Damien Doligez Georges Gonthier*

the basic algorithm of [9], and expose a series of counterexamples to explain why a straightforward adaptation of this algorithm to multiple mutators would not work; we also address some efficiency issues. In section 4 we describe the

PLDI ‘00

Implementing an On-the-fly Garbage Collector for Java

Tamar Domani Elliot K. Kolodner* Ethan Lewis Eliot E. Salant
Katherine Barabash Itai Lahan Erez Petrank Igor Yanover
Yossi Levanoni

Abstract

introduce mark buffers, adapt the algorithm to Java

An on-the-fly garbage collector

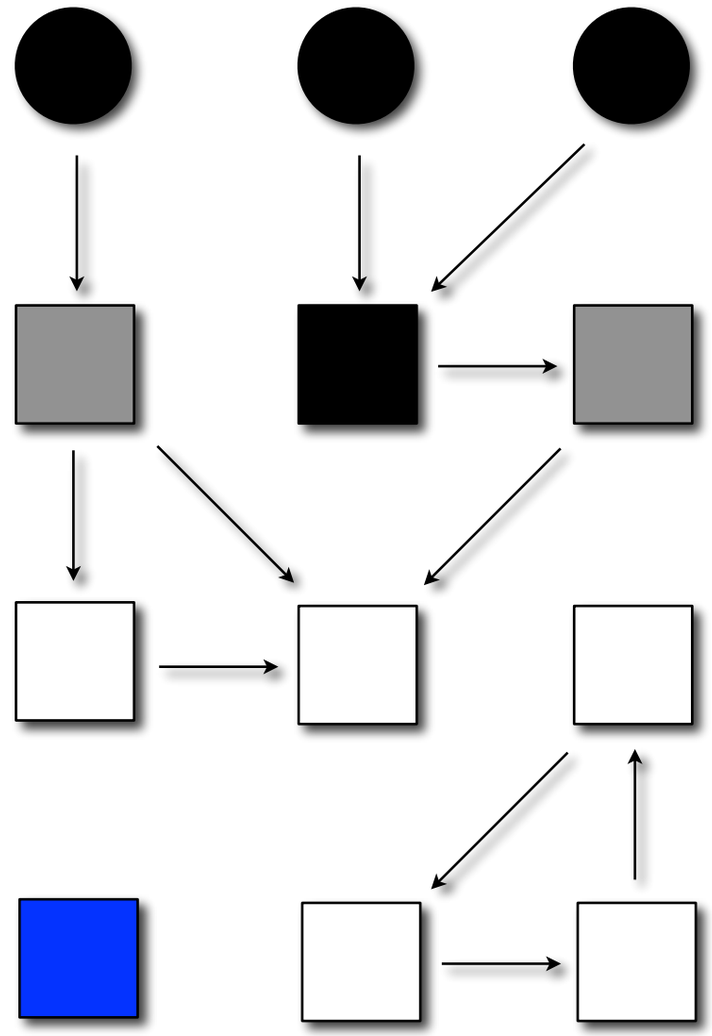
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



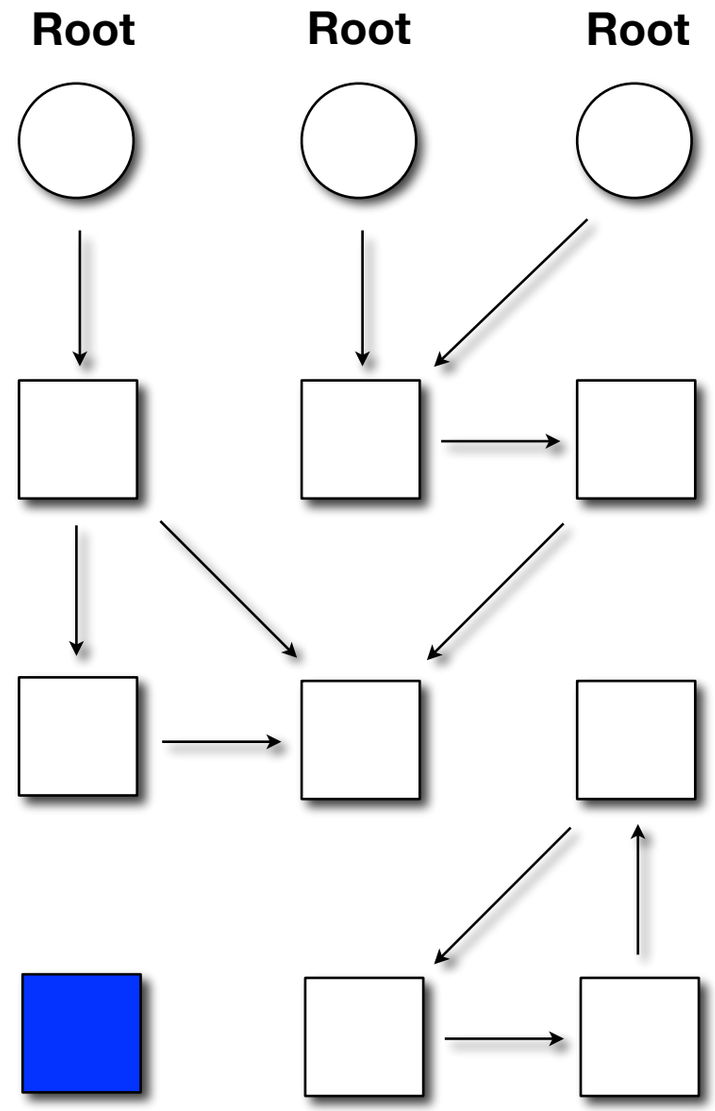
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



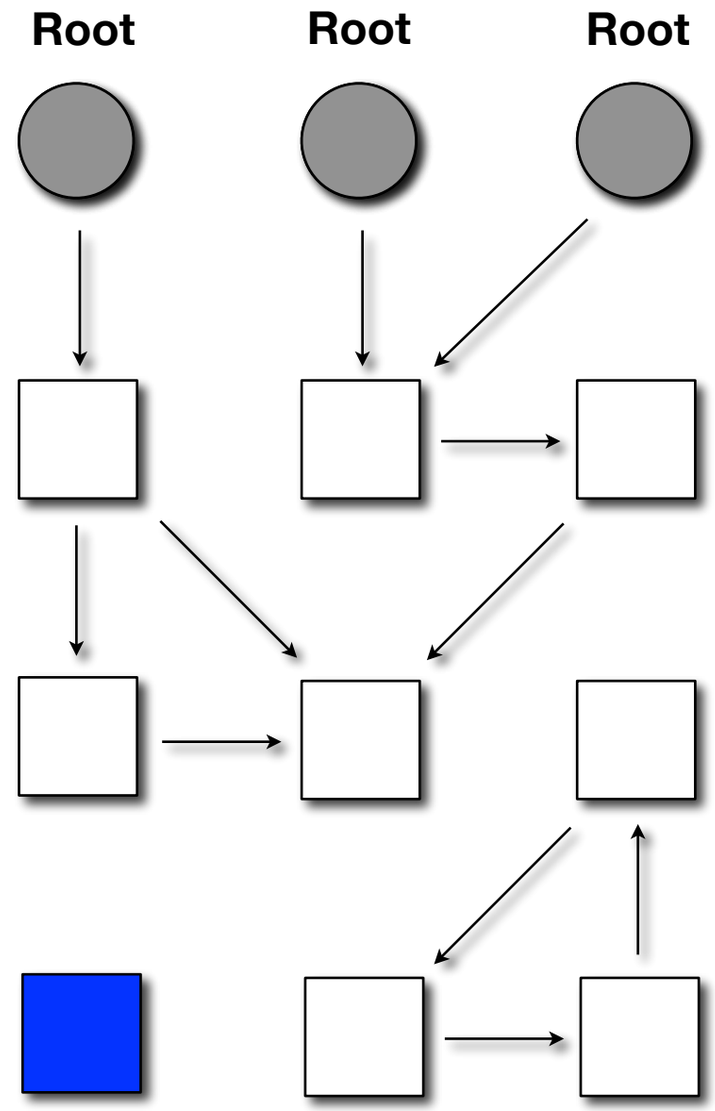
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



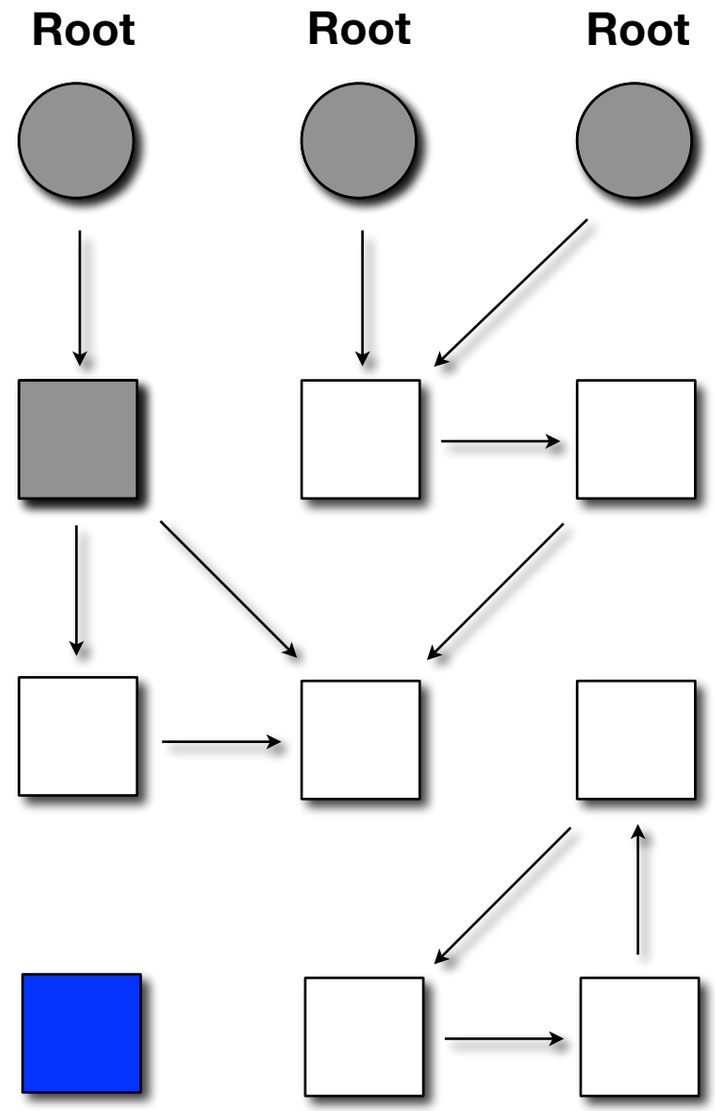
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



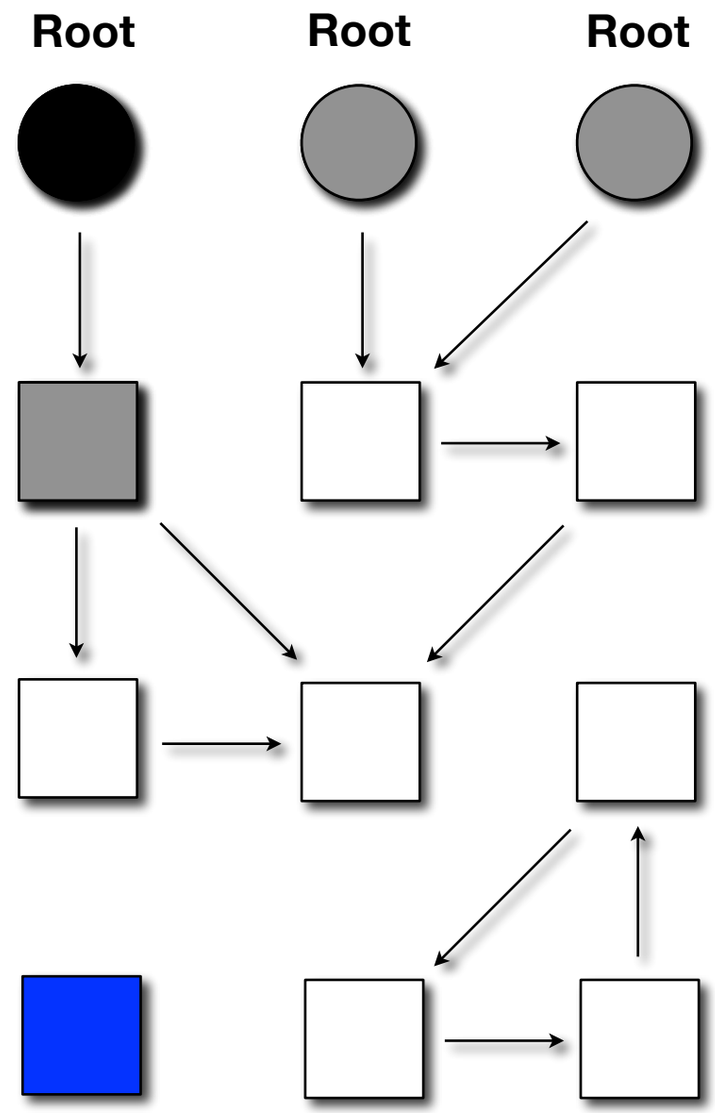
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



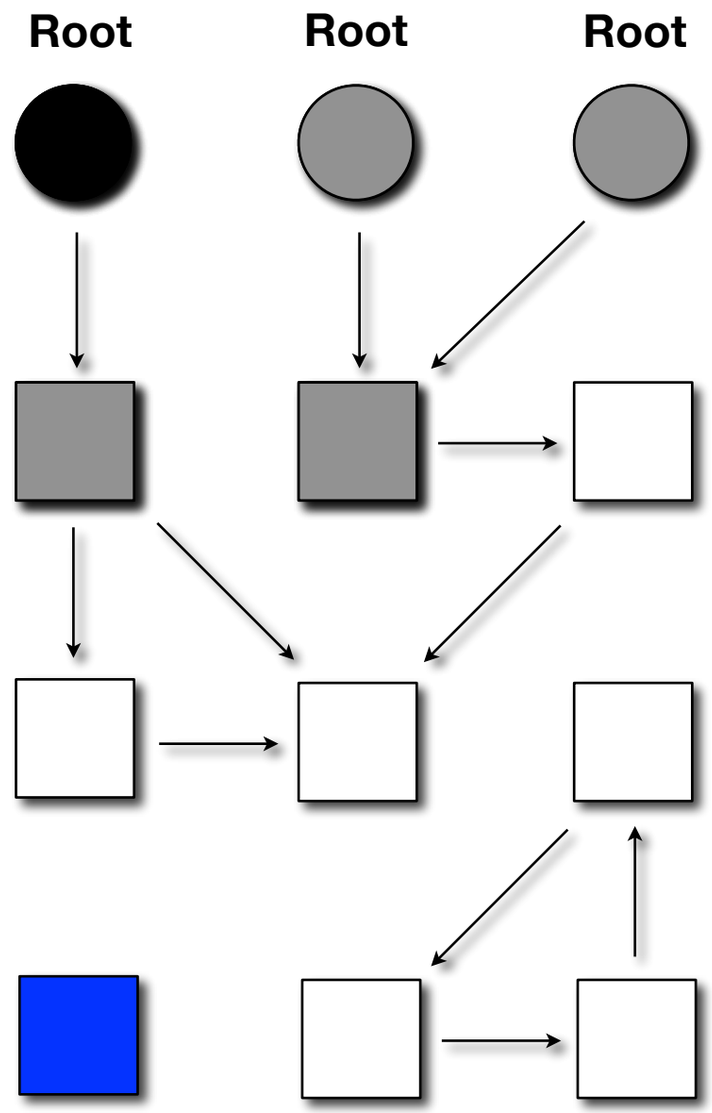
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



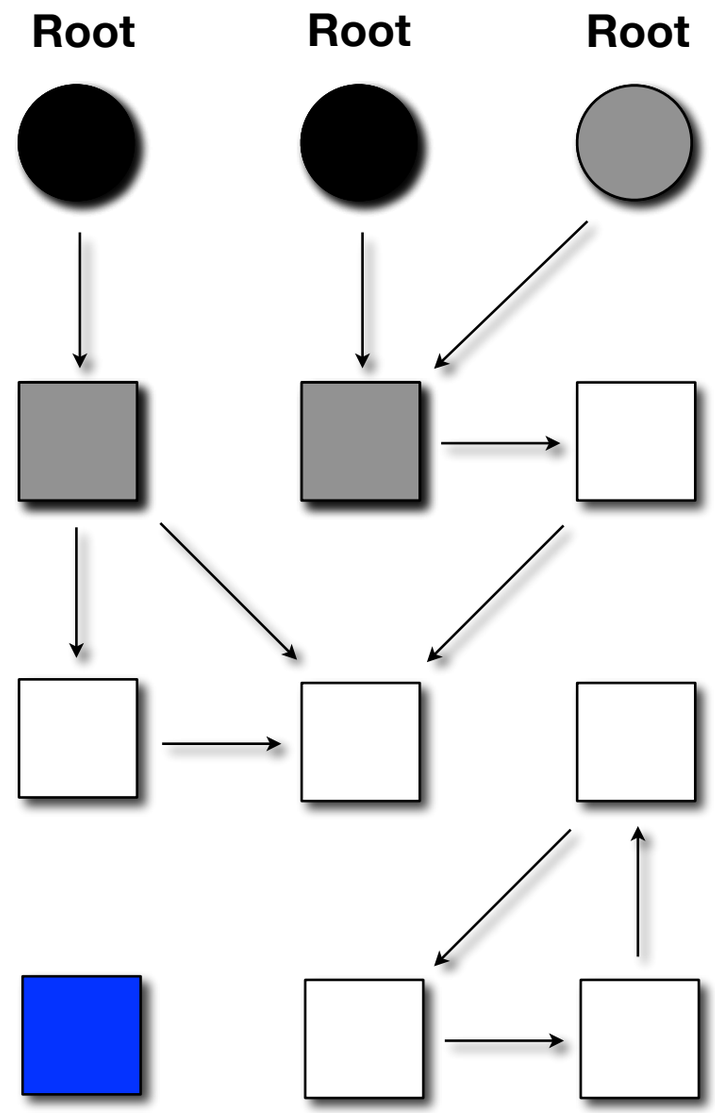
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



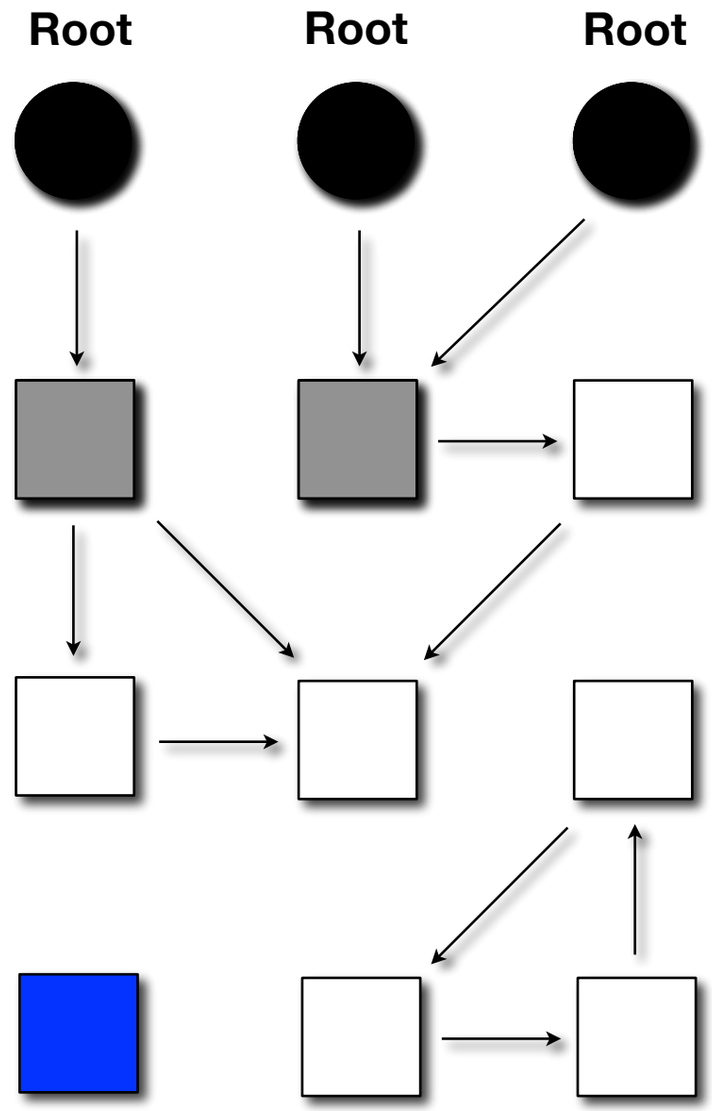
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



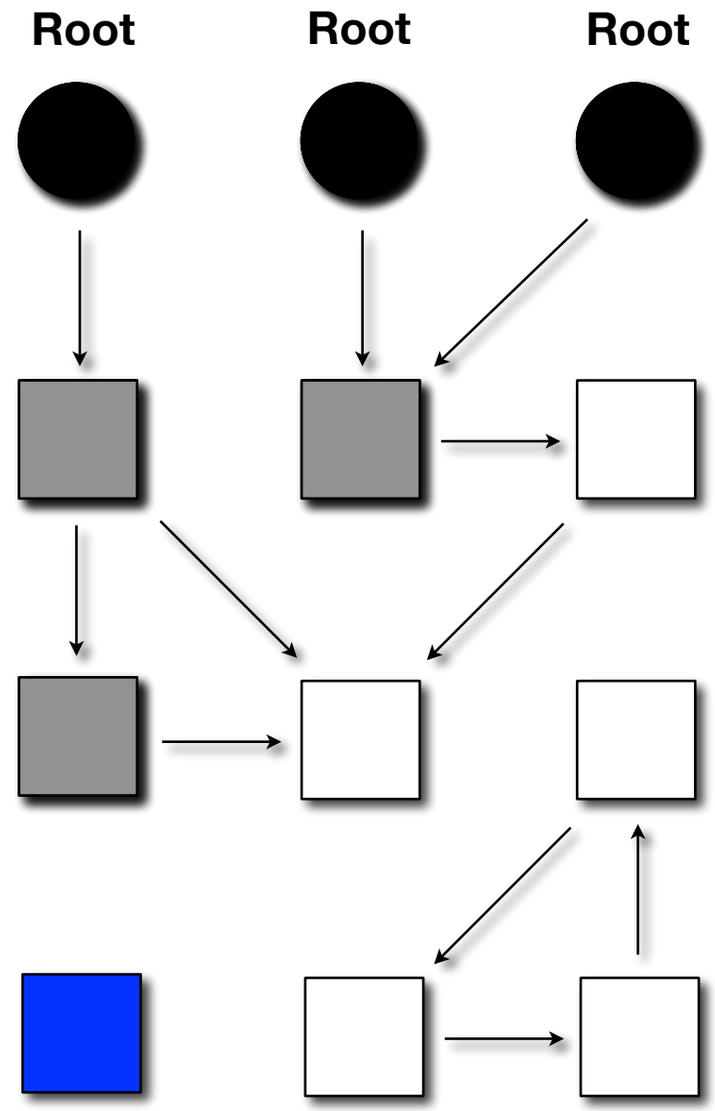
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



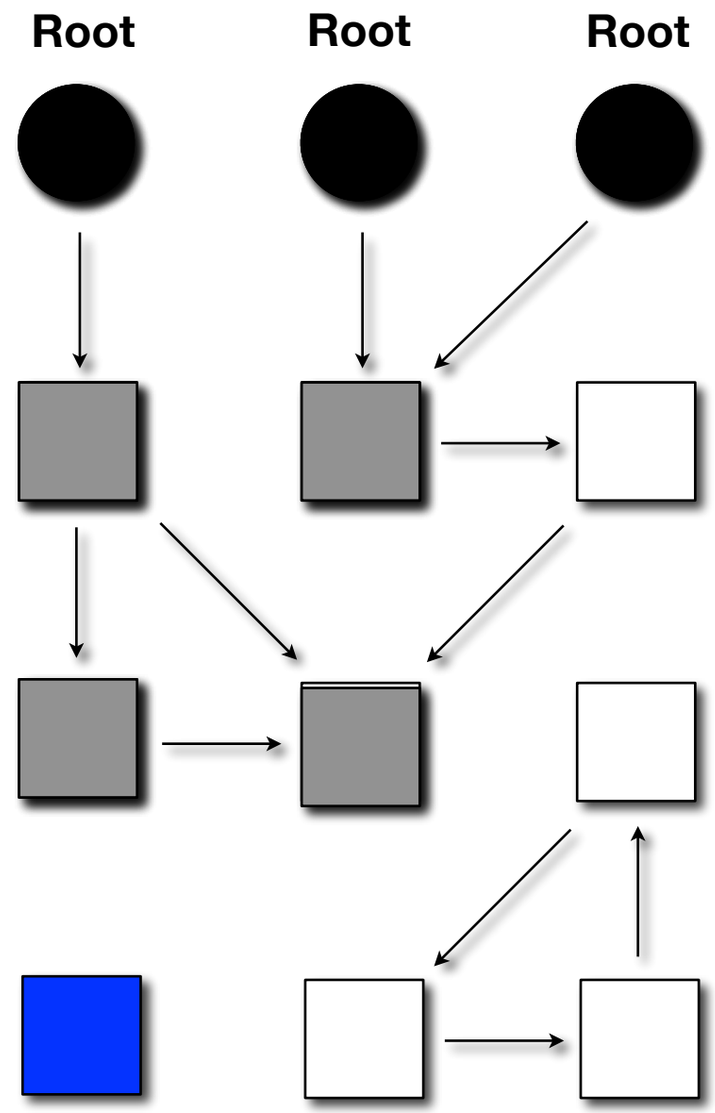
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



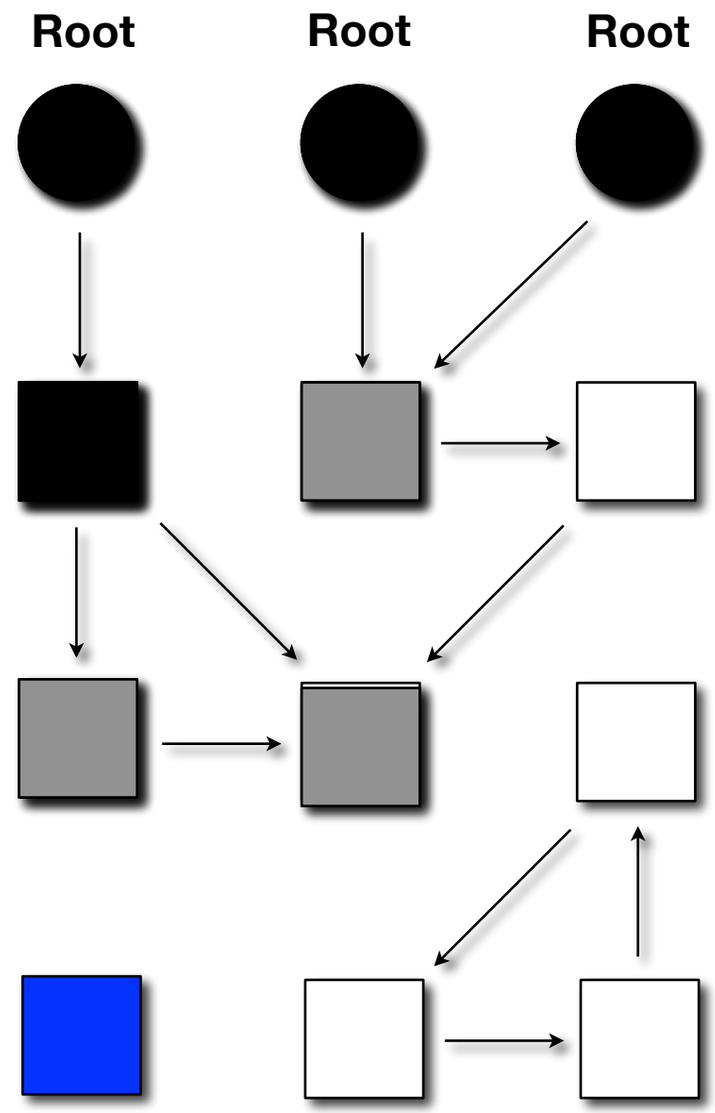
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



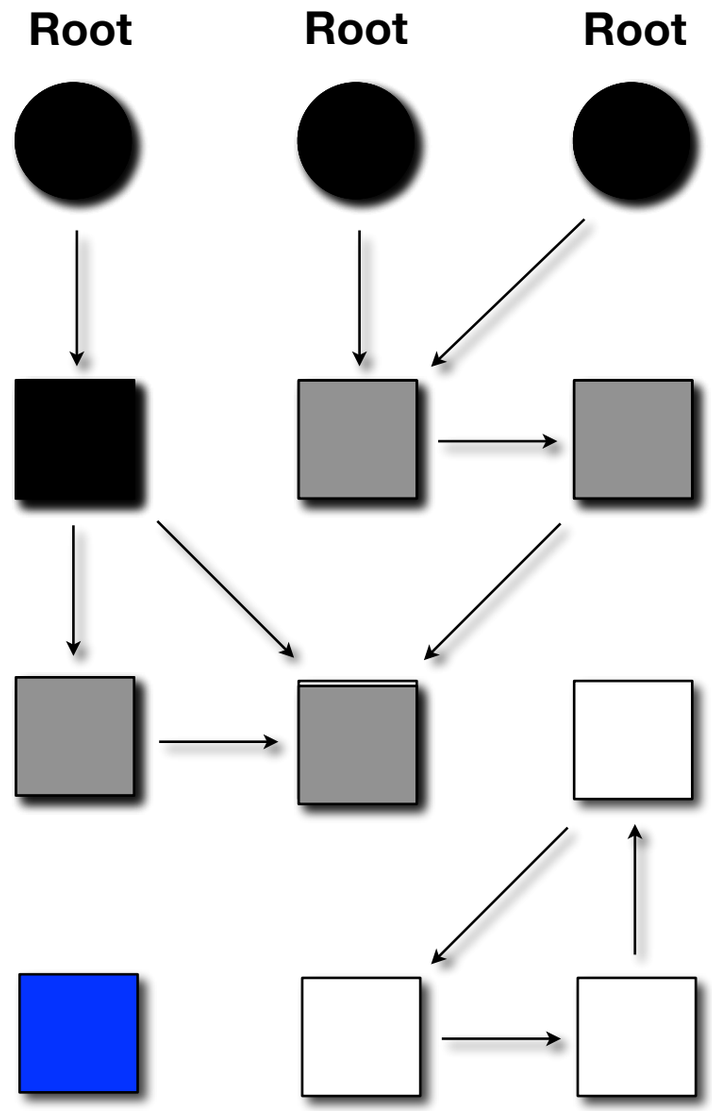
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



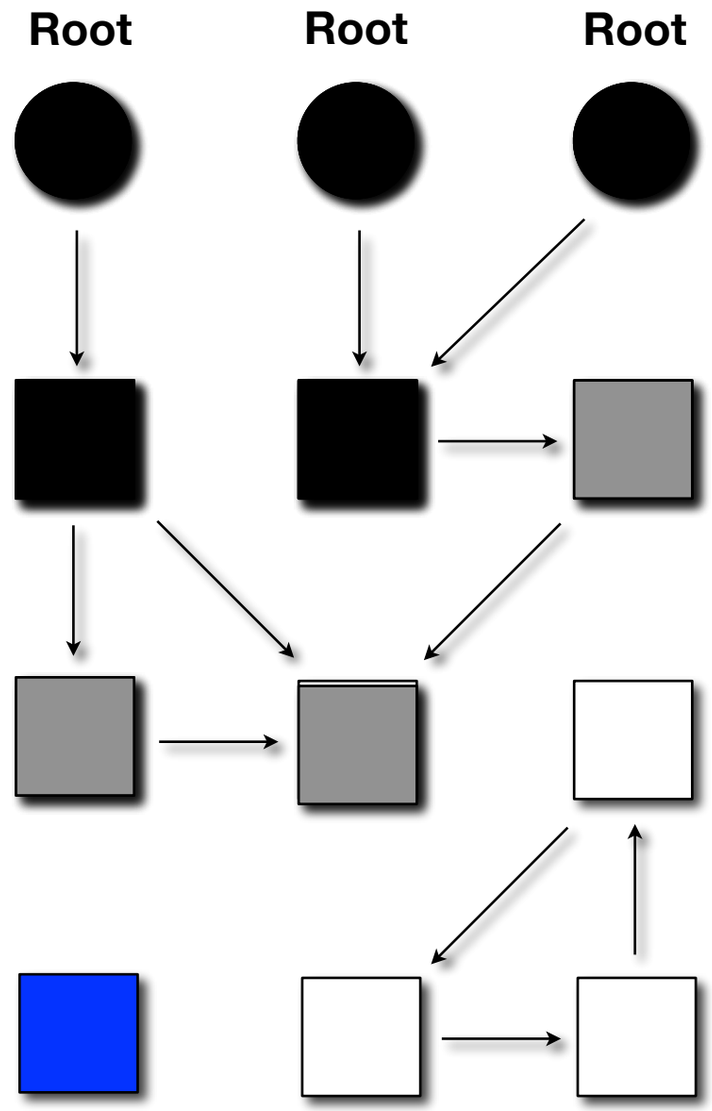
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



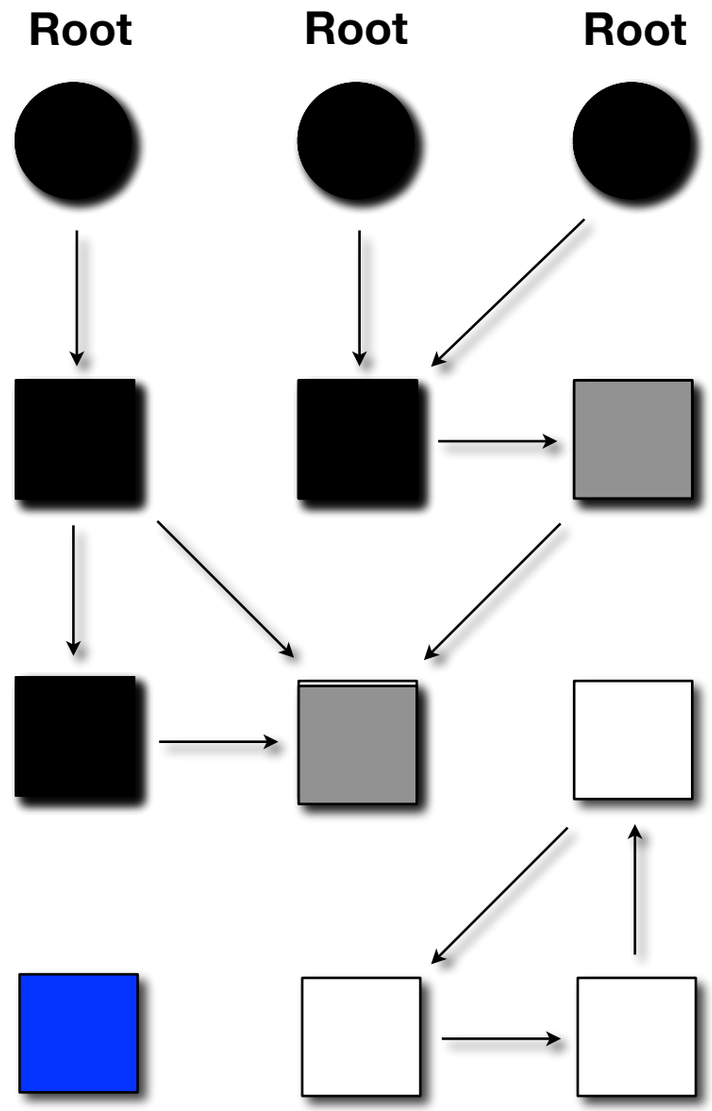
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



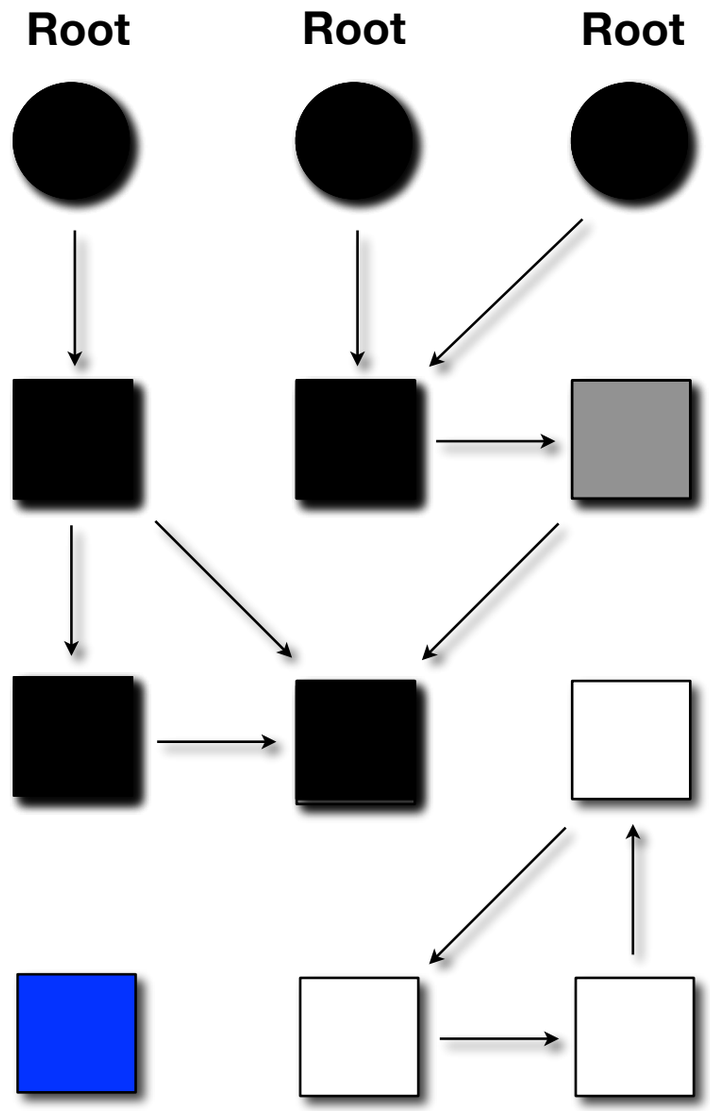
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



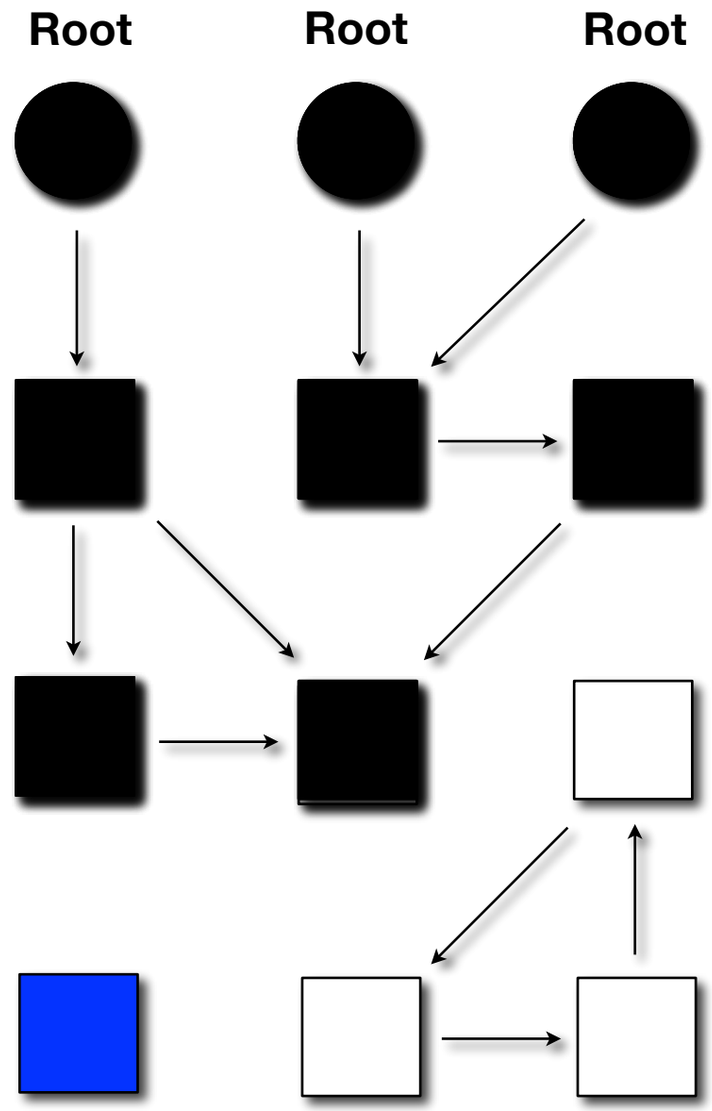
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked



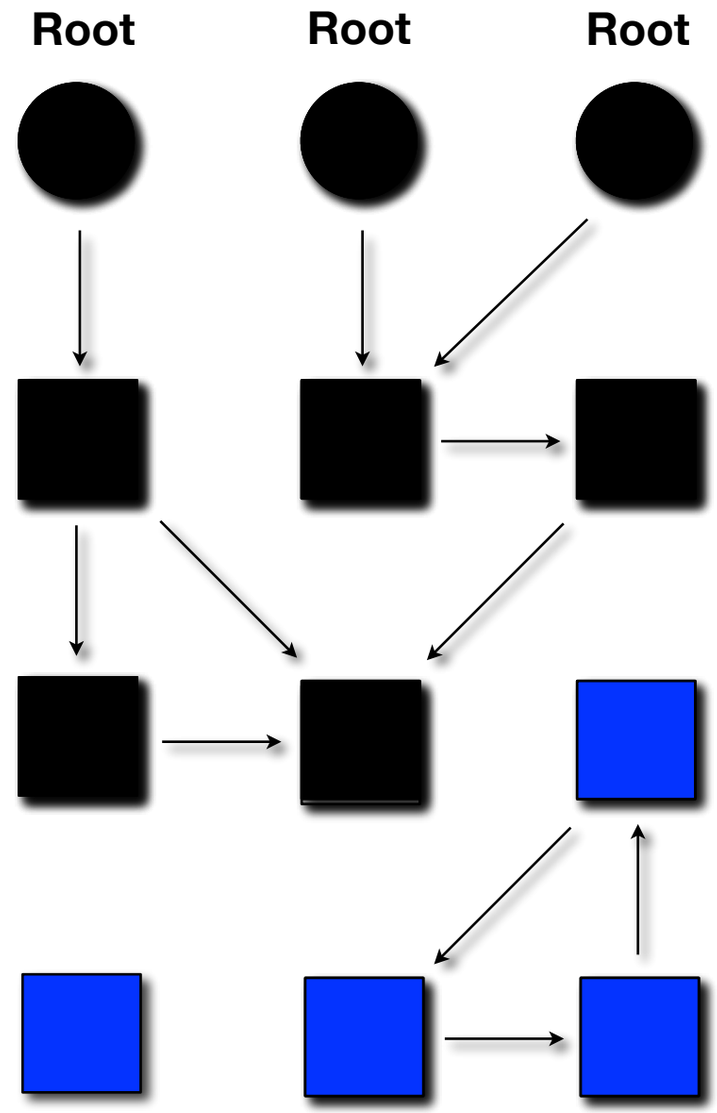
Sequential mark & sweep

Periodically stop user code and perform:

- Mark: graph traversal from the roots
- Sweep: free cells remaining unmarked

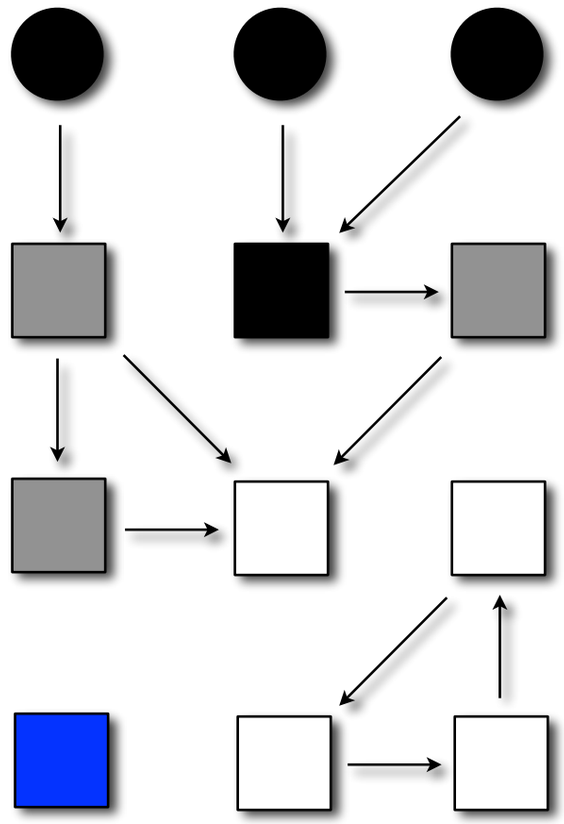
Color conventions:

- Blue: free cells
- White: not marked
- Gray: being visited (pending nodes)
- Black: marked

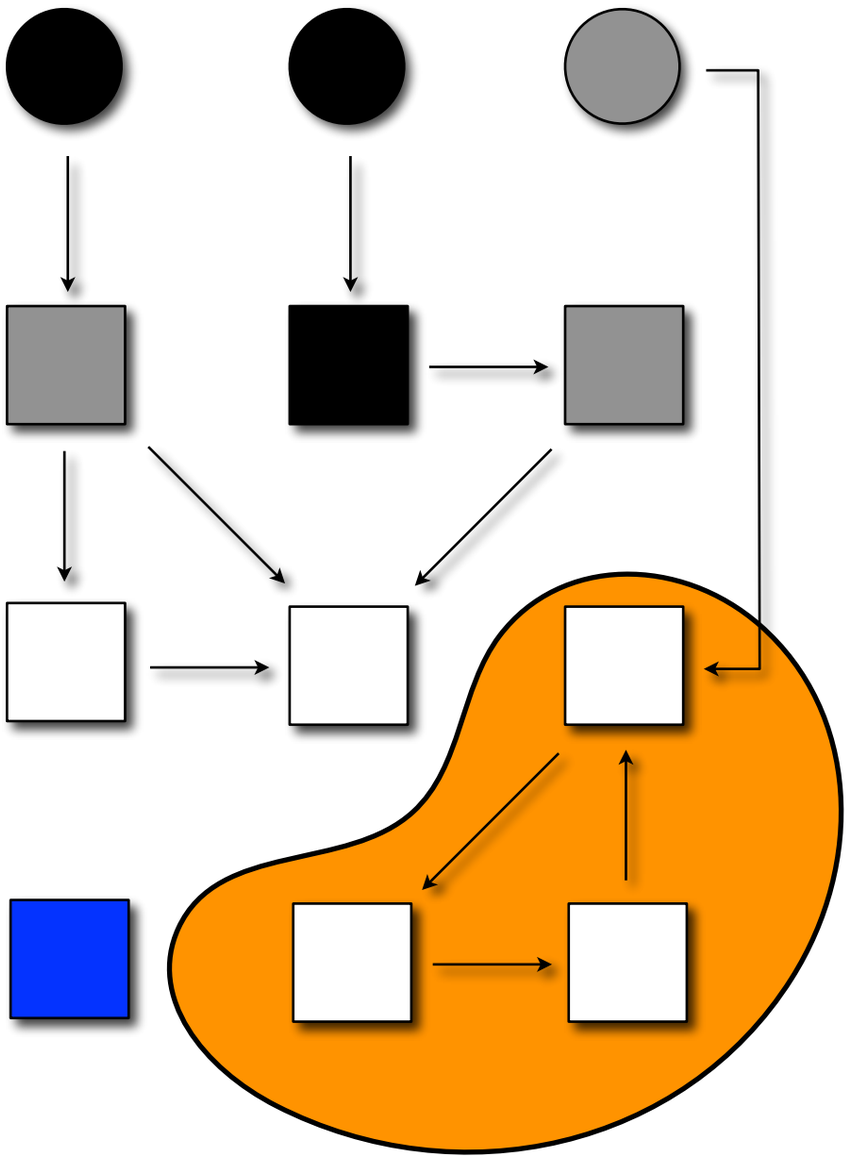


Concurrent mark & sweep

1. A need for the *mutators'* collaboration

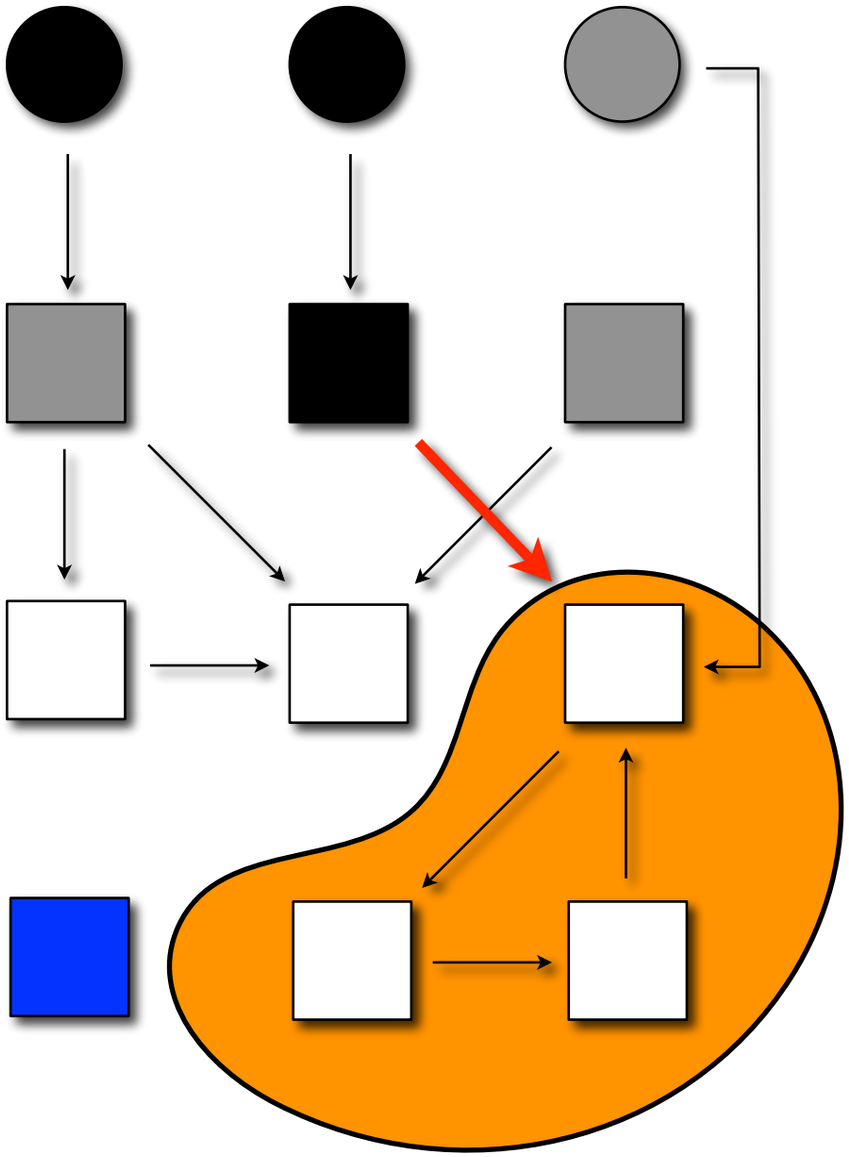


Concurrent mark & sweep



While the collector runs, the user thread can do:

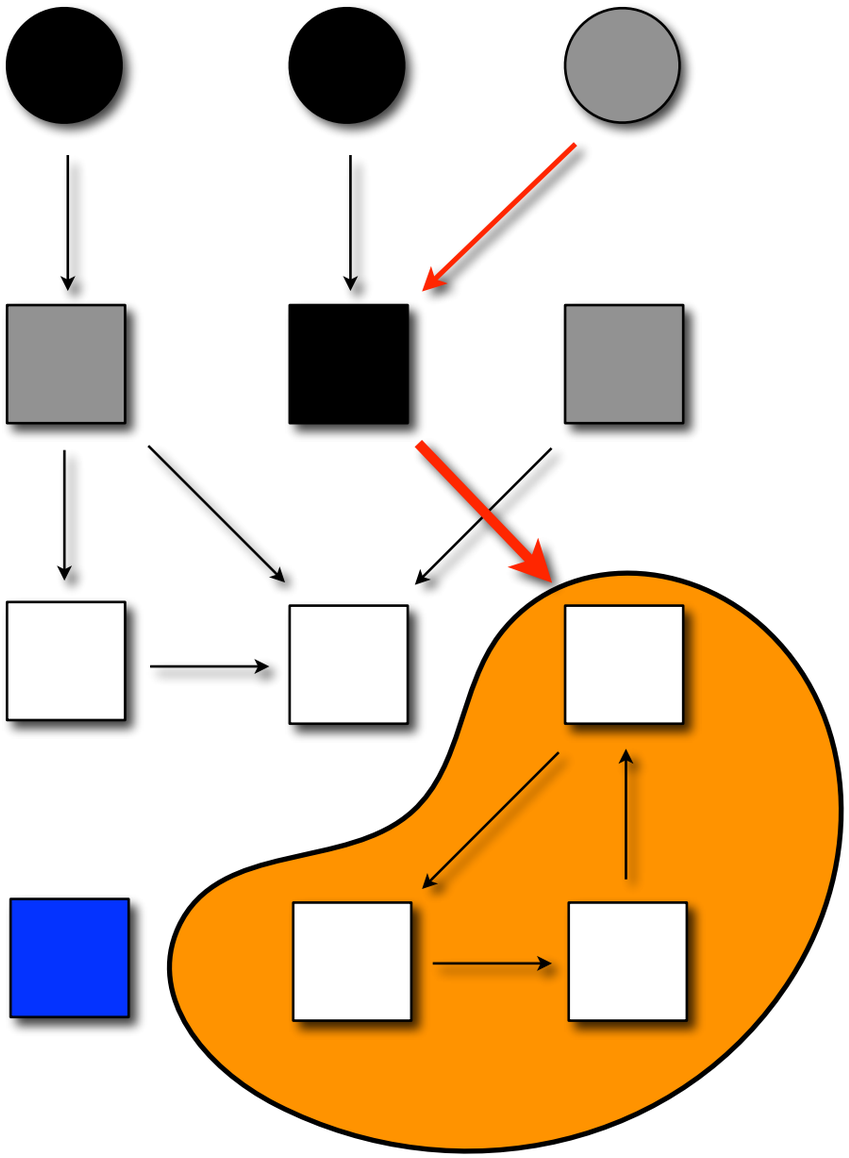
Concurrent mark & sweep



While the collector runs, the user thread can do:

1. an update;

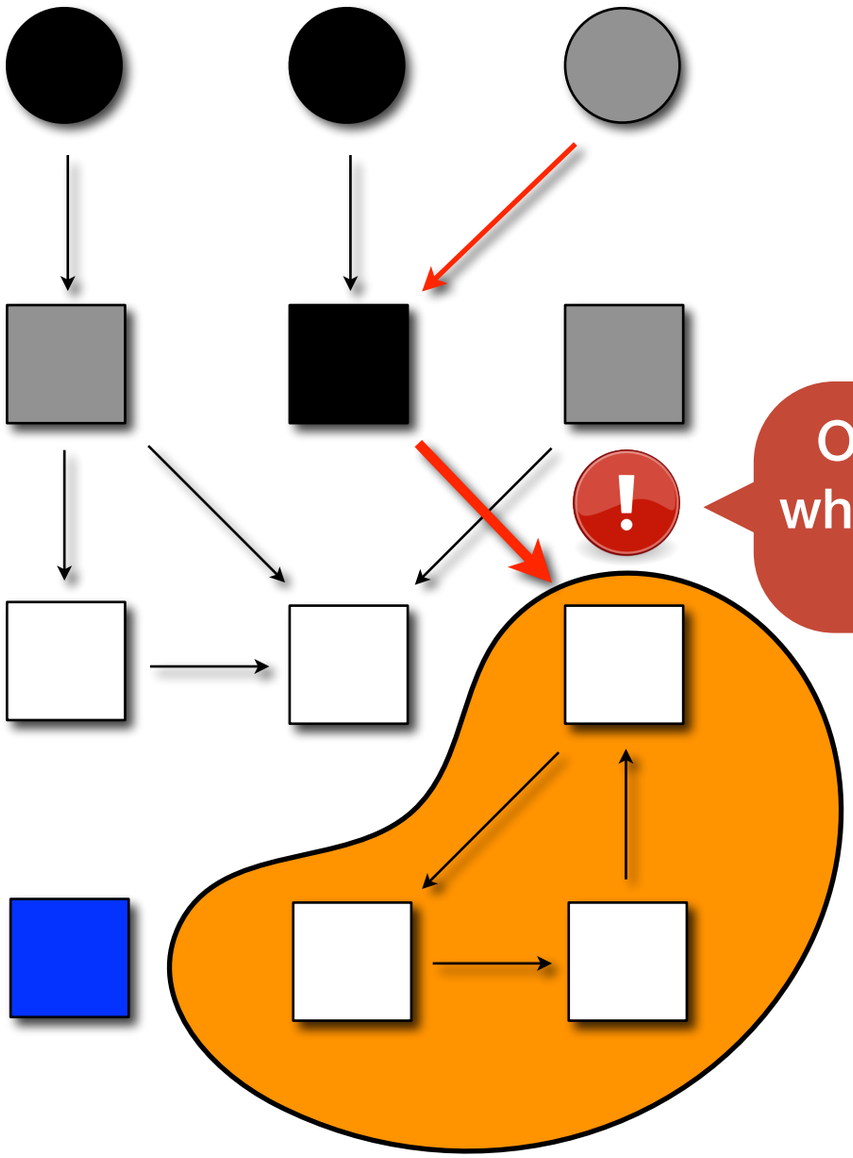
Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**
2. **a store.**

Concurrent mark & sweep



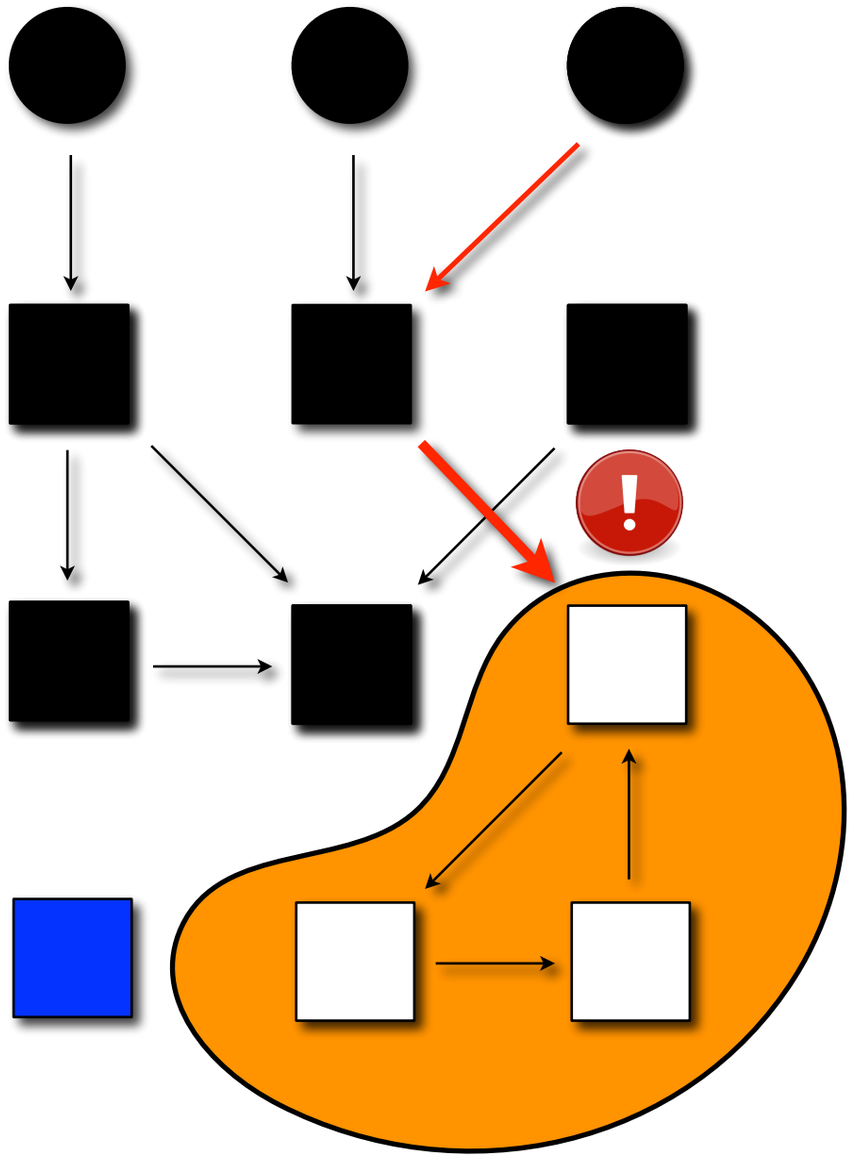
While the collector runs, the user thread can do:

1. an update;

2. a store.

Only access to white subgraph is black!

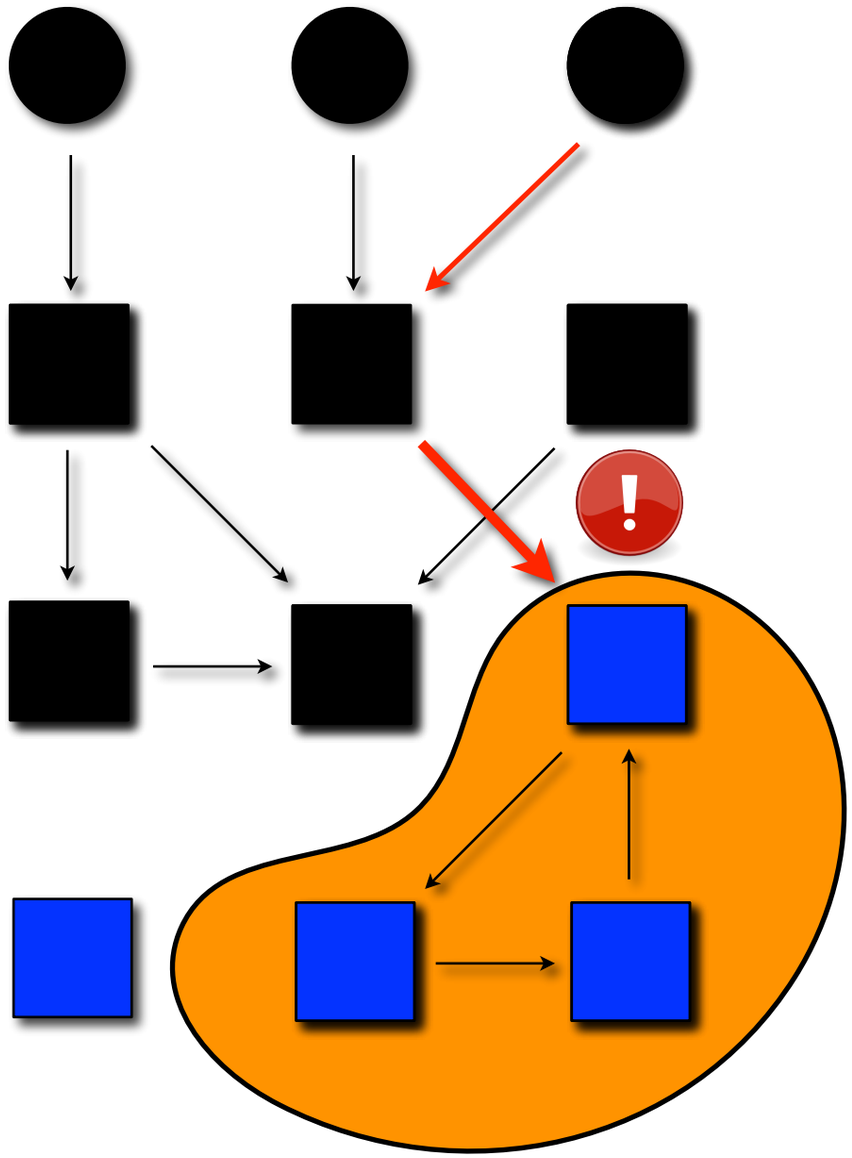
Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**
2. **a store.**

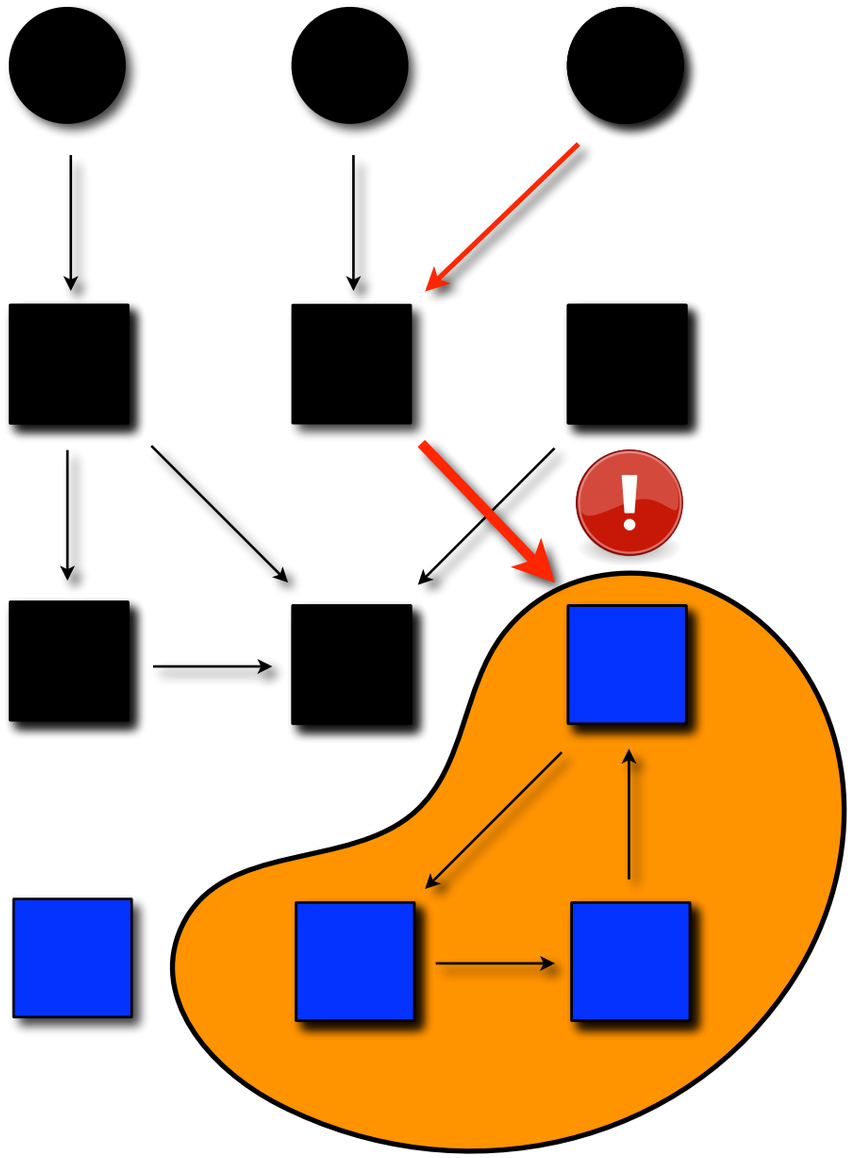
Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**
2. **a store.**

Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**

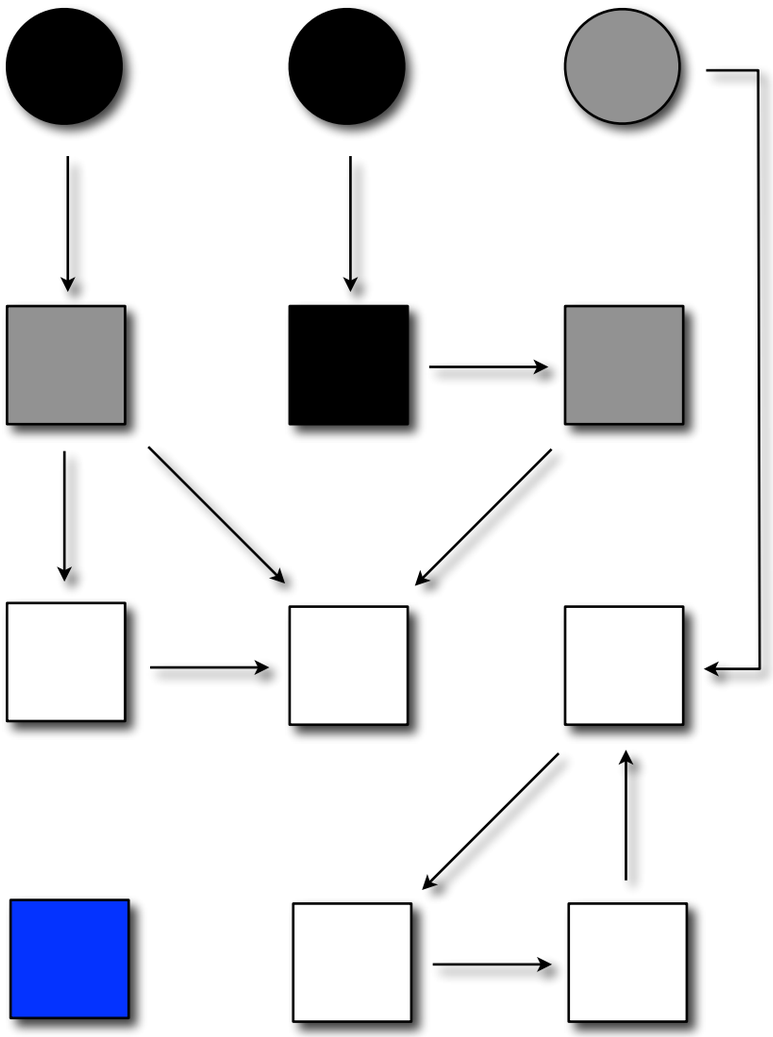
2. **a store.**

Updates go through write barriers

```
Update(x,f,y) ==
    MarkGray(y);
    x.f = y

MarkGray(x) ==
    if x.color = WHITE
    then x.color = GRAY
```

Concurrent mark & sweep



While the collector runs, the user thread can do:

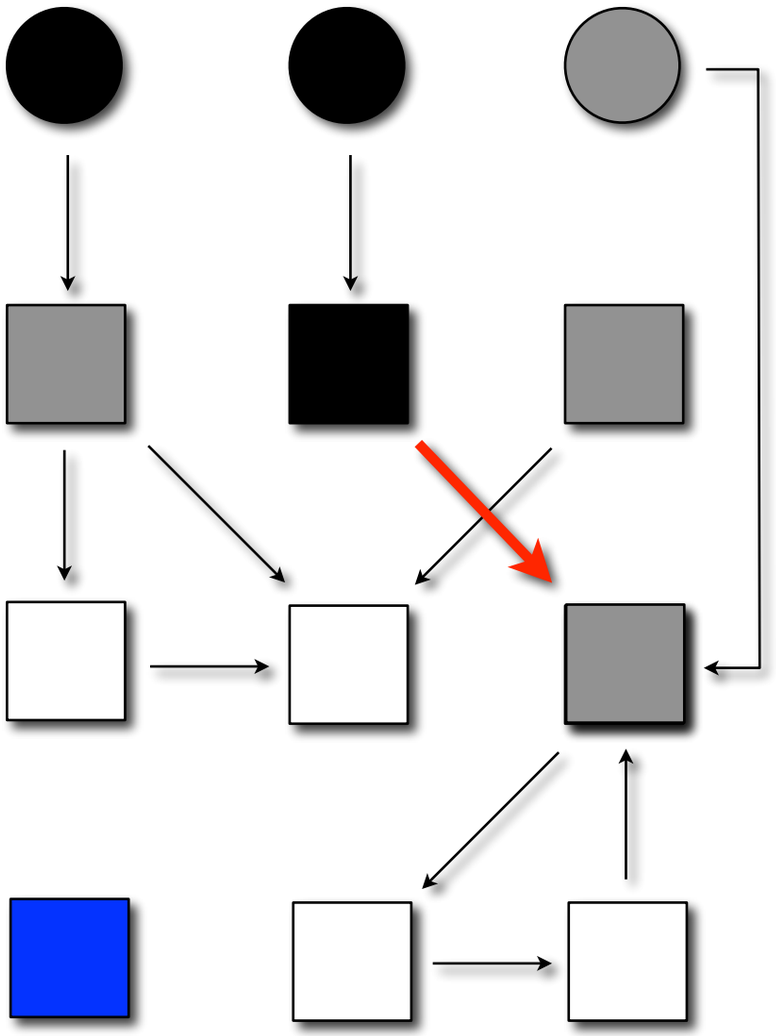
1. **an update;**
2. **a store.**

Updates go through write barriers

```
Update(x,f,y) ==
    MarkGray(y);
    x.f = y

MarkGray(x) ==
    if x.color = WHITE
    then x.color = GRAY
```

Concurrent mark & sweep



While the collector runs, the user thread can do:

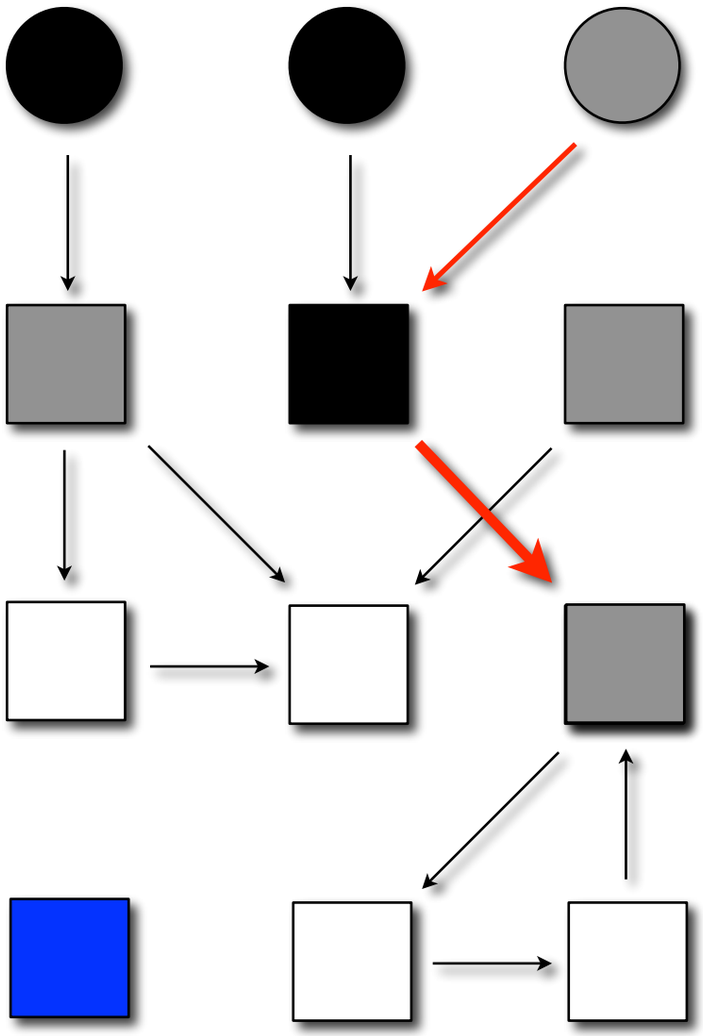
1. **an update;**
2. **a store.**

Updates go through write barriers

```
Update(x,f,y) ==
    MarkGray(y);
    x.f = y

MarkGray(x) ==
    if x.color = WHITE
    then x.color = GRAY
```

Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**

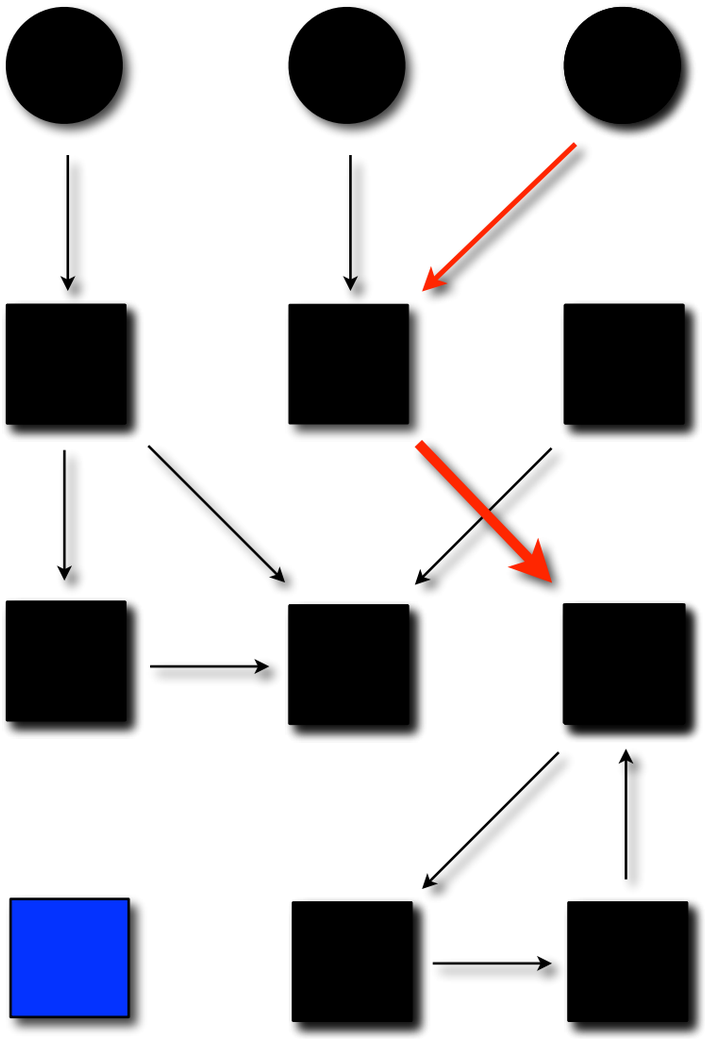
2. **a store.**

Updates go through write barriers

```
Update(x,f,y) ==
    MarkGray(y);
    x.f = y

MarkGray(x) ==
    if x.color = WHITE
    then x.color = GRAY
```

Concurrent mark & sweep



While the collector runs, the user thread can do:

1. **an update;**

2. **a store.**

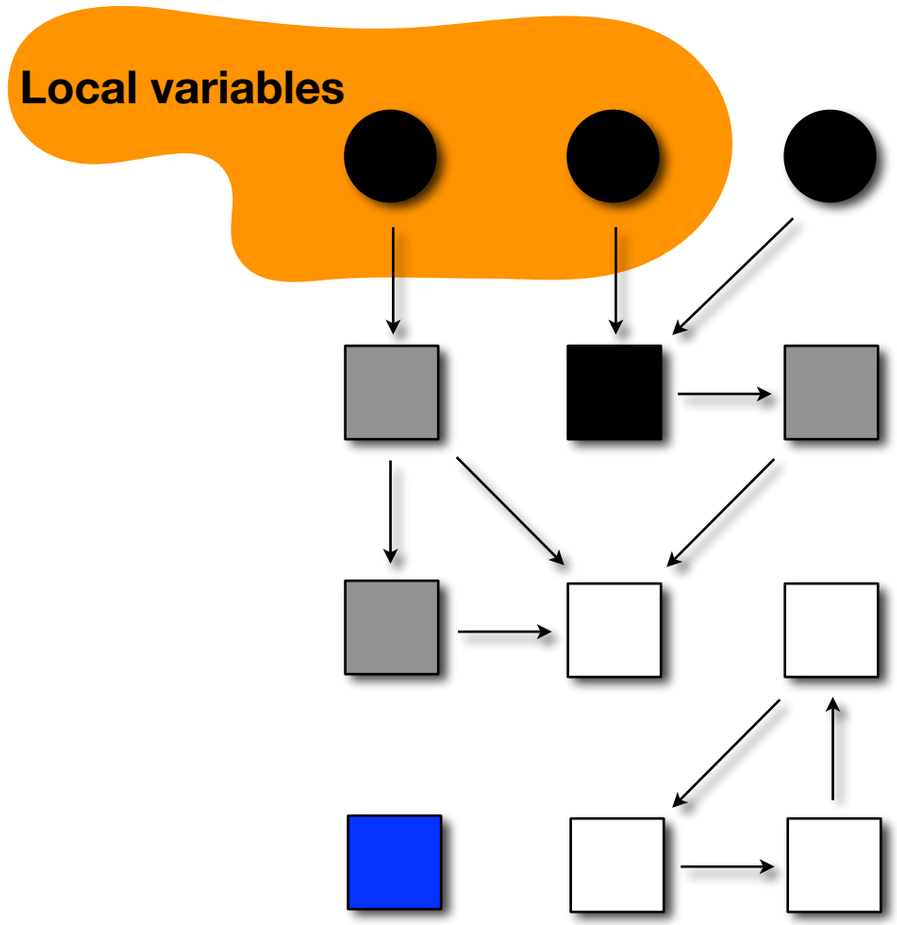
Updates go through write barriers

```
Update(x,f,y) ==
    MarkGray(y);
    x.f = y

MarkGray(x) ==
    if x.color = WHITE
    then x.color = GRAY
```

Concurrent mark & sweep

2. A need for synchronisation



Concurrent mark & sweep

MUTATOR

```
[...]
  Update(x,f,y);
[...]
  Alloc();
[…]
```

COLLECTOR

```
Scan:
  repeat
    no_gray = true;
    foreach x ∈ OBJECTS
      if x.color == GRAY
        no_gray = false;
        foreach f ∈ fields(x) do
          MarkGray(x.f);
        x.color = BLACK
  until no_gray
 Sweep:
  foreach x ∈ OBJECTS
    if x.color == WHITE
    then FREE(x)
Clear:
  foreach x ∈ OBJECTS
    x.color = WHITE
```

Concurrent mark & sweep

MUTATOR

```
[...]
  Update(x,f,y);
[...]
  Alloc();
[…]
```

COLLECTOR

```
Scan:
  repeat
    no_gray = true;
    foreach x ∈ OBJECTS
      if x.color == GRAY
        no_gray = false;
        foreach f ∈ fields(x) do
          MarkGray(x.f);
        x.color = BLACK
  until no_gray
 Sweep:
  foreach x ∈ OBJECTS
    if x.color == WHITE
    then FREE(x)
Clear:
  foreach x ∈ OBJECTS
    x.color = WHITE
```

The collector has not access to all mutator roots…

Concurrent mark & sweep

MUTATOR

```
[...]
  Update(x,f,y);
[...]
  Alloc();
[…]
```

mark your roots please

COLLECTOR

```
Scan:
  repeat
    no_gray = true;
    foreach x ∈ OBJECTS
      if x.color == GRAY
        no_gray = false;
        foreach f ∈ fields(x) do
          MarkGray(x.f);
        x.color = BLACK
  until no_gray
Sweep:
  foreach x ∈ OBJECTS
    if x.color == WHITE
    then FREE(x)
Clear:
  foreach x ∈ OBJECTS
    x.color = WHITE
```

The collector has not access to all mutator roots…

Concurrent mark & sweep

MUTATOR

```
[...]
  Udpate(x1,f1,y1);
[...]
  Cooperate();
[...]
  Alloc();
[...]
```

They need to synchronise!

```
Mark:
  Handshake();
  foreach x ∈ GLOBALS
  do MarkGray(x)

        ay = true;
      ch x ∈ OBJECTS
      color == GRAY
        no_gray = false;
        foreach f ∈ fields(x) do
          MarkGray(x.f);
        x.color = BLACK
  until no_gray
 Sweep:
  foreach x ∈ OBJECTS
    if x.color == WHITE
    then FREE(x)
Clear:
  foreach x ∈ OBJECTS
    x.color = WHITE
```

Concurrent mark & sweep

MUTATOR

```
[...]
  Udpate(x1,f1,y1);
[...]
  Cooperate();
[...]
  Alloc();
[...]
```

They need to synchronise!

But user threads should not wait!

```
Mark:
  Handshake();
  foreach x ∈ GLOBALS
  do MarkGray(x)

     ay = true;
    ch x ∈ OBJECTS
    color == GRAY
  no_gray = false;
  foreach f ∈ fields(x) do
    MarkGray(x.f);
    olor = BLACK
  o_gray

  h x ∈ OBJECTS
  olor == WHITE
 then FREE(x)
Clear:
 foreach x ∈ OBJECTS
  x.color = WHITE
```

Concurrent mark & sweep

Mutators mark their roots

```
Cooperate() =
  if status_m = status_C
  then
    skip
  else
    foreach r ∈ LOCAL_ROOTS do
        MarkGray(r);
    status_m = status_C;
```

Collector awaits for mutators to mark their roots

```
Handshake() =
  status_C = Next(status_C);
  while (status_m ≠ status_C) skip;
```

Concurrent mark & sweep

Mutators mark their roots

```
Cooperate() =
  if statusm = statusC
  then
    skip
  else
    foreach r ∈ LOCAL_ROOTS do
        MarkGray(r);
    statusm = statusC;
```

Collector awaits for mutators to mark their roots

```
Handshake() =
  statusC = Next(statusC);
  while (statusm ≠ statusC) skip;
```

Updates its status

Actively waits

Concurrent mark & sweep

Mutators mark their roots

```
Cooperate() =
  if status_m = status_C
  then
    skip
  else
    foreach r ∈ LOCAL_ROOTS do
       MarkGray(r);
    status_m = status_C;
```

Checks if in sync
Yes? Back to work
No? Publishes roots

} **Never waits for anyone**

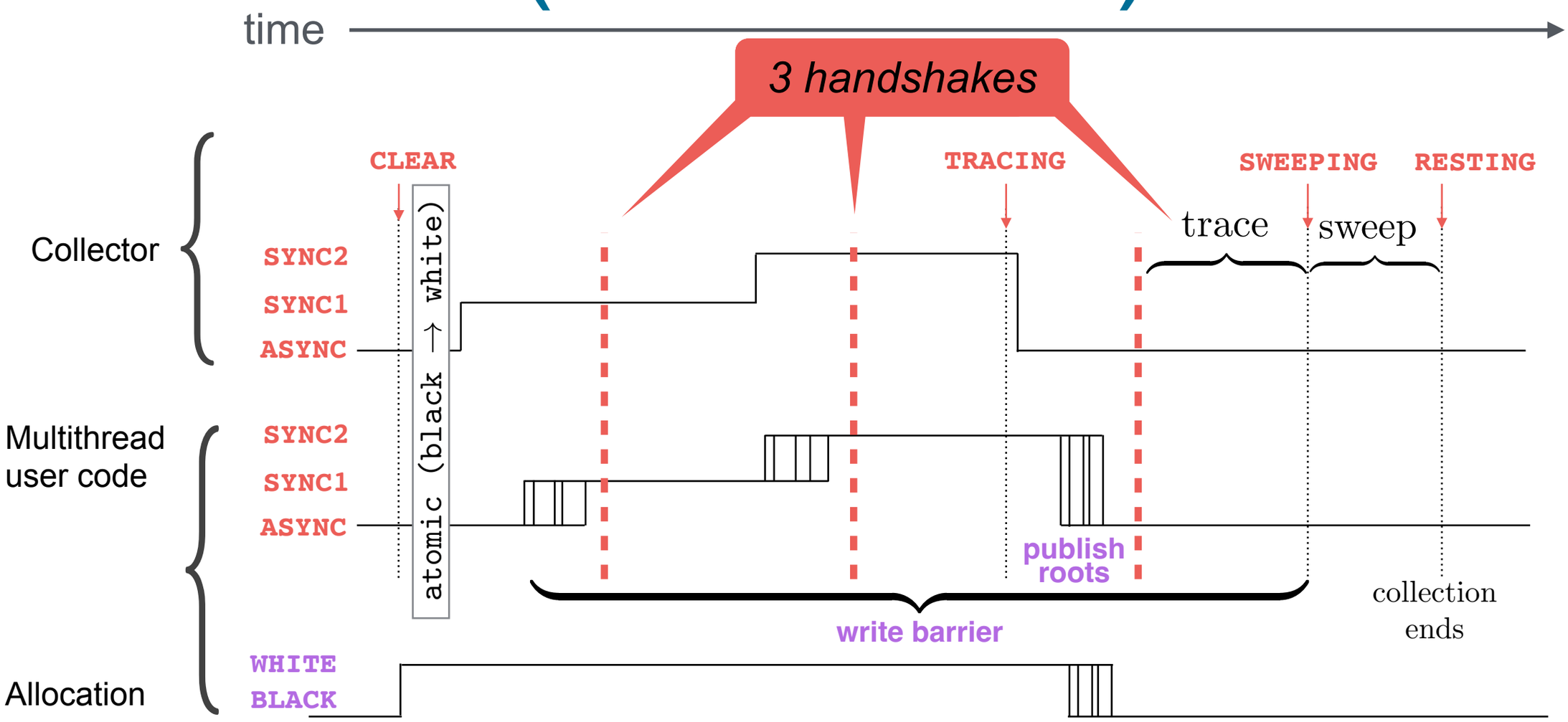
Collector awaits for mutators to mark their roots

```
Handshake() =
  status_C = Next(status_C);
  while (status_m ≠ status_C) skip;
```

Updates its status
Actively waits

Timeline of a collection cycle (DLG 93/94)

Timeline of a collection cycle (DLG 93/94)



- Graph algorithms
- Subtle synchronisation
- Lots of concurrent accesses

} Sophisticated invariants

Verifying the garbage collector

1. What do we prove?
2. How do we prove it?

What do we prove?

We rely on a *most general client* (mgc)

$$\texttt{mgc} \triangleq \texttt{collector} \parallel \texttt{mutator} \parallel \ldots \parallel \texttt{mutator}$$

$$\texttt{mutator} \triangleq \mathsf{loop}\big(\texttt{update}(\texttt{x},\texttt{f},\texttt{v}) \oplus \texttt{load}(\texttt{x},\texttt{f}) \oplus \texttt{alloc}() \oplus \texttt{cooperate}() \oplus \texttt{changeRoots}()\big)$$

What do we prove?

We rely on a *most general client* (mgc)

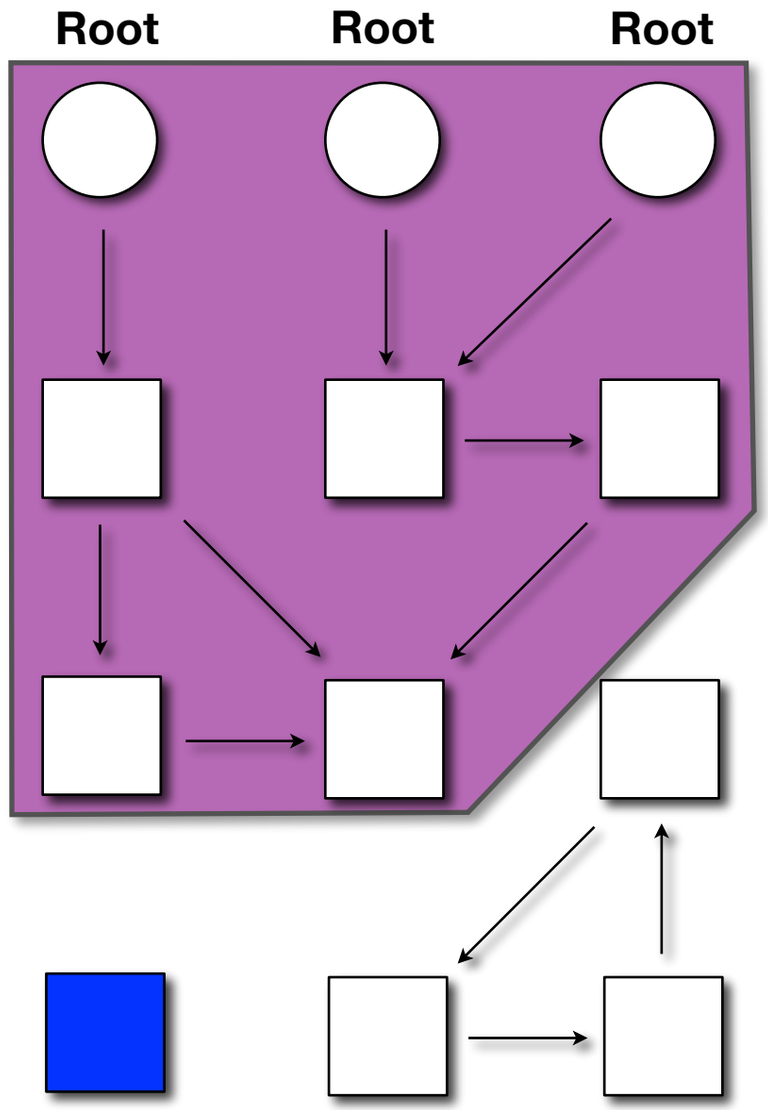
$$\texttt{mgc} \triangleq$$
$$\texttt{collector} \parallel \texttt{mutator} \parallel \ldots \parallel \texttt{mutator}$$

$$\texttt{mutator} \triangleq$$
$$\texttt{loop}\big(\texttt{update}(\texttt{x},\texttt{f},\texttt{v}) \oplus \texttt{load}(\texttt{x},\texttt{f}) \oplus \texttt{alloc}() \oplus \texttt{cooperate}() \oplus \texttt{changeRoots}()\big)$$

Abstraction of any user thread

What do we prove?

Theorem:
In any execution state of the mgc, cells reachable by a mutator are never blue



An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} := & & \\
\quad | & \quad \texttt{skip} & \quad | \quad x = \texttt{alloc}(rn) \\
\quad | & \quad \texttt{assume}\ e & \quad | \quad \texttt{free}(x) \\
\quad | & \quad x = [y].f & \quad | \quad x = y.\texttt{empty?}() \\
\quad | & \quad [x].f = e & \quad | \quad x = y.\texttt{top}() \\
\quad | & \quad \texttt{atomic}\ c & \quad | \quad x.\texttt{push}(y) \\
\quad | & \quad c_1\ ;\ c_2 & \quad | \quad x.\texttt{pop}() \\
\quad | & \quad c_1 \oplus c_2 & \quad | \quad \texttt{isFree?}(x) \\
\quad | & \quad \texttt{loop}(c) & \quad | \quad \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od}
\end{array}
$$

An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} := & & \\
\quad | & \texttt{skip} & | \quad x = \texttt{alloc}(rn) \\
\quad | & \texttt{assume}\ e & | \quad \texttt{free}(x) \\
\quad | & x = [y].f & | \quad x = y.\texttt{empty?}() \\
\quad | & [x].f = e & | \quad x = y.\texttt{top}() \\
\quad | & \texttt{atomic}\ c & | \quad x.\texttt{push}(y) \\
\quad | & c_1\ ;\ c_2 & | \quad x.\texttt{pop}() \\
\quad | & c_1 \oplus c_2 & | \quad \texttt{isFree?}(x) \\
\quad | & \texttt{loop}(c) & | \quad \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od}
\end{array}
$$

An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} := & & \\
\quad | & \texttt{skip} & | \quad x = \texttt{alloc}(rn) \\
\quad | & \texttt{assume}\ e & | \quad \texttt{free}(x) \\
\quad | & x = [y].f & | \quad x = y.\texttt{empty?}() \\
\quad | & [x].f = e & | \quad x = y.\texttt{top}() \\
\quad | & \texttt{atomic}\ c & | \quad x.\texttt{push}(y) \\
\quad | & c_1\ ;\ c_2 & | \quad x.\texttt{pop}() \\
\quad | & c_1 \oplus c_2 & | \quad \texttt{isFree?}(x) \\
\quad | & \texttt{loop}(c) & | \quad \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od}
\end{array}
$$

An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} & := & \\
 & \mid \quad \texttt{skip} & \mid \quad x = \texttt{alloc}(rn) \\
 & \mid \quad \texttt{assume}\ e & \mid \quad \texttt{free}(x) \\
 & \mid \quad x = [y].f & \mid \quad x = y.\texttt{empty?}() \\
 & \mid \quad [x].f = e & \mid \quad x = y.\texttt{top}() \\
 & \mid \quad \texttt{atomic}\ c & \mid \quad x.\texttt{push}(y) \\
 & \mid \quad c_1\ ;\ c_2 & \mid \quad x.\texttt{pop}() \\
 & \mid \quad c_1 \oplus c_2 & \mid \quad \texttt{isFree?}(x) \\
 & \mid \quad \texttt{loop}(c) & \mid \quad \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od}
\end{array}
$$

- Abstract buffers:
  - concrete implementations are linearizable

```
MarkGray(m, x) ==
    if x.color = WHITE then
        push(buffer[m], x)
```

An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} := & & \\
\quad | & \texttt{skip} & \\
\quad | & \texttt{assume}\ e & \\
\quad | & x = [y].f & \\
\quad | & [x].f = e & \\
\quad | & \texttt{atomic}\ c & \\
\quad | & c_1\ ;\ c_2 & \\
\quad | & c_1 \oplus c_2 & \\
\quad | & \texttt{loop}(c) & \\
\quad | & x = \texttt{alloc}(rn) & \\
\quad | & \texttt{free}(x) & \\
\quad | & x = y.\texttt{empty?}() & \\
\quad | & x = y.\texttt{top}() & \\
\quad | & x.\texttt{push}(y) & \\
\quad | & x.\texttt{pop}() & \\
\quad | & \texttt{isFree?}(x) & \\
\quad | & \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od} &
\end{array}
$$

- Intrinsic support for threads, roots, and objects
- Built-in iterator constructs : disciplined access

An IR to program the GC

$$
\begin{array}{lll}
\texttt{cmd} := & & \\
\quad | & \texttt{skip} & \\
\quad | & \texttt{assume}\ e & \\
\quad | & x = [y].f & \\
\quad | & [x].f = e & \\
\quad | & \texttt{atomic}\ c & \\
\quad | & c_1\ ;\ c_2 & \\
\quad | & c_1 \oplus c_2 & \\
\quad | & \texttt{loop}(c) & \\
\quad | & x = \texttt{alloc}(rn) & \\
\quad | & \texttt{free}(x) & \\
\quad | & x = y.\texttt{empty?}() & \\
\quad | & x = y.\texttt{top}() & \\
\quad | & x.\texttt{push}(y) & \\
\quad | & x.\texttt{pop}() & \\
\quad | & \texttt{isFree?}(x) & \\
\quad | & \texttt{foreach}\ (\texttt{x}\ in\ \texttt{l})\ \texttt{do}\ c\ \texttt{od} &
\end{array}
$$

Right level of abstraction: proofs are conducted with respect to the operational semantics of the IR, directly over the code

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash_t \{P\} \;\; c \;\; \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness
Invariant

Annotations

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness
Invariant

Annotations

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness Invariant

Annotations

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

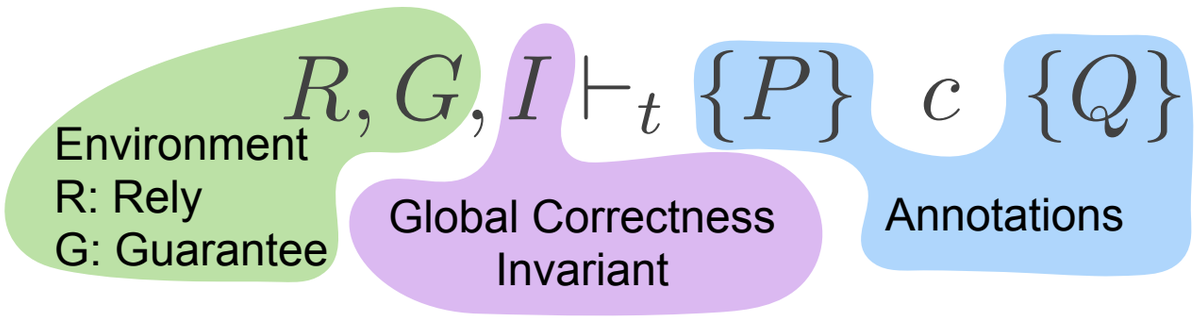
Environment
R: Rely
G: Guarantee

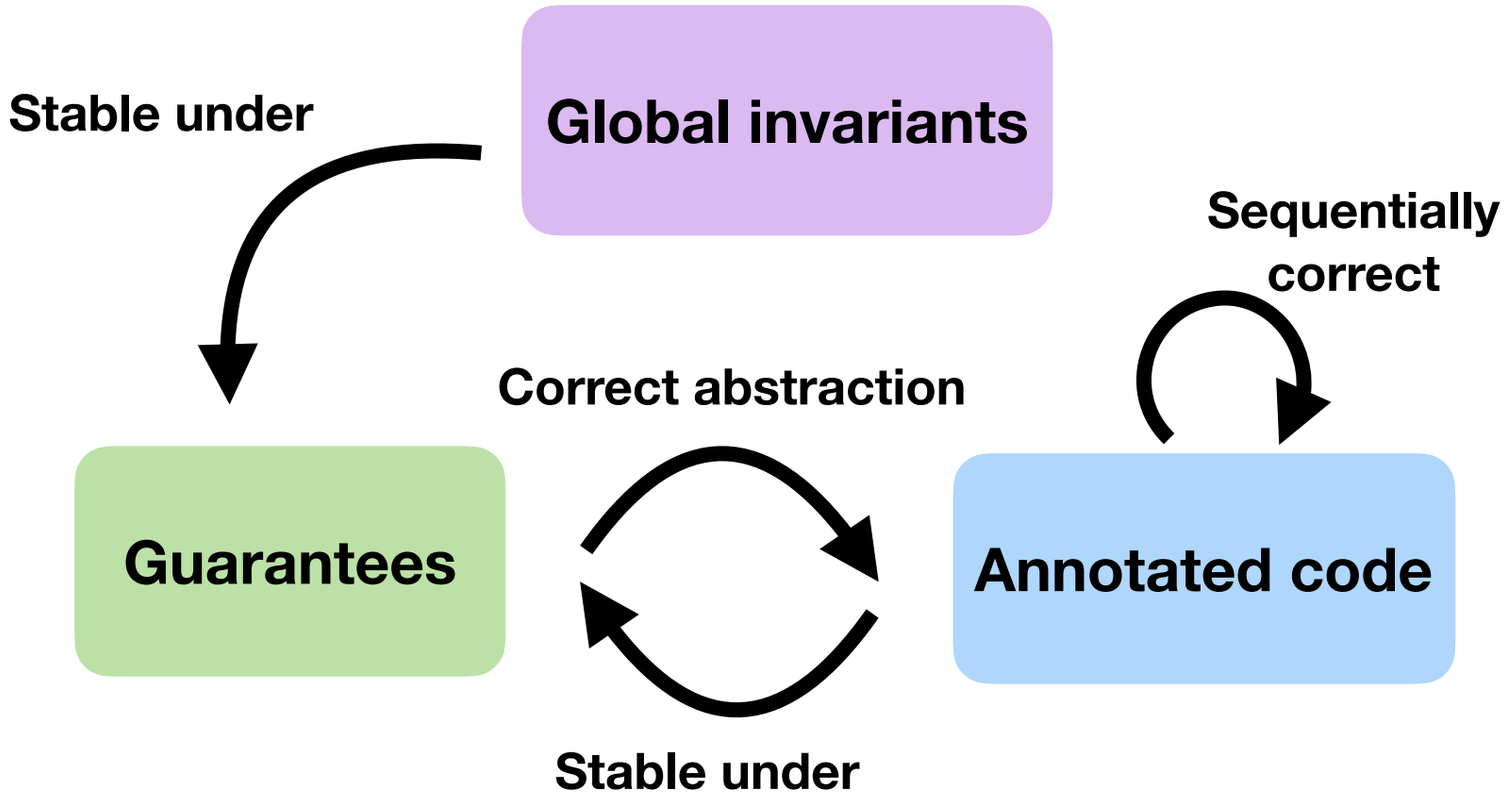
Global Correctness Invariant

Annotations

Rely Guarantee reasoning

[Jones81]





Rely Guarantee reasoning

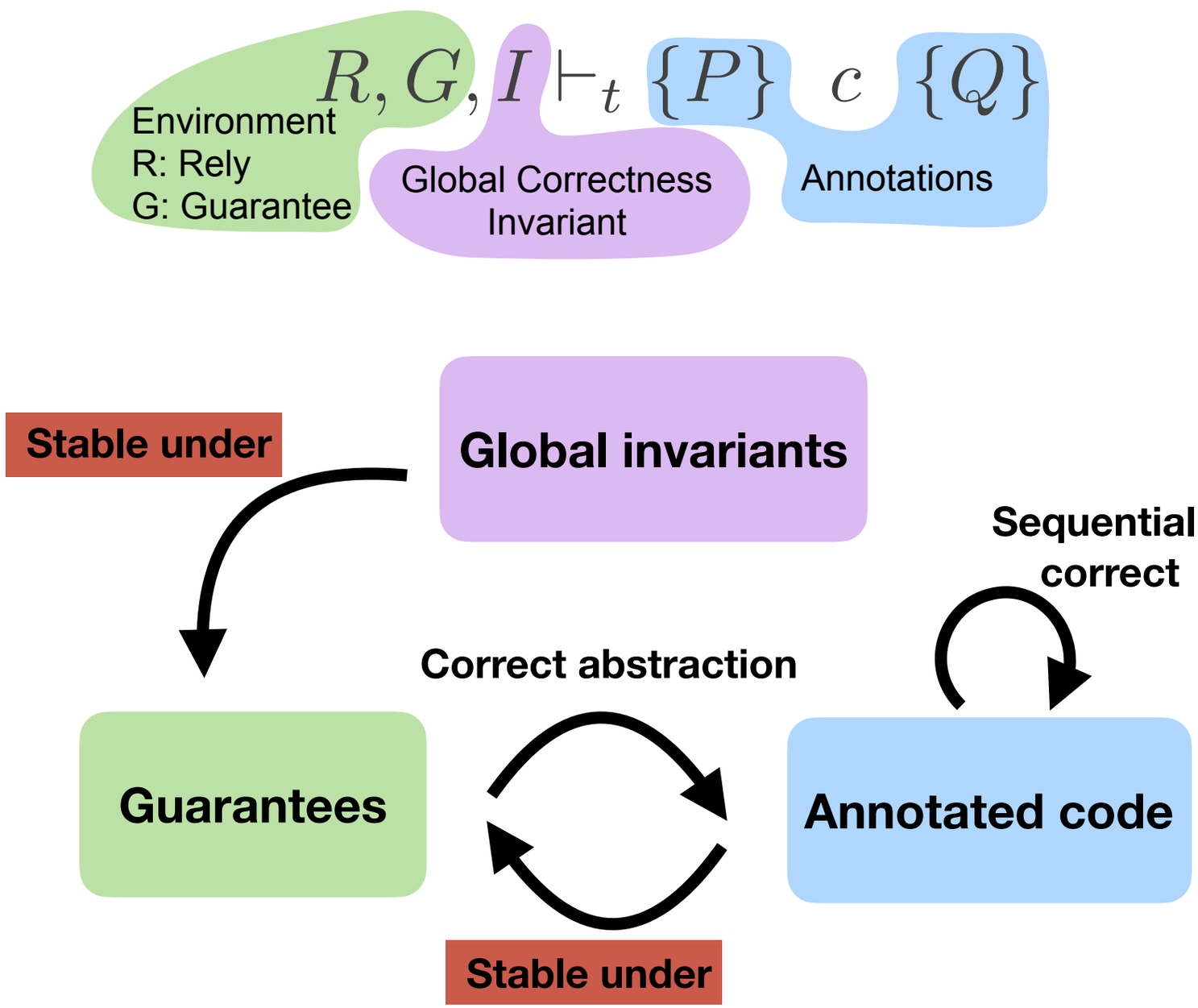
[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness Invariant

Annotations



Rely Guarantee reasoning

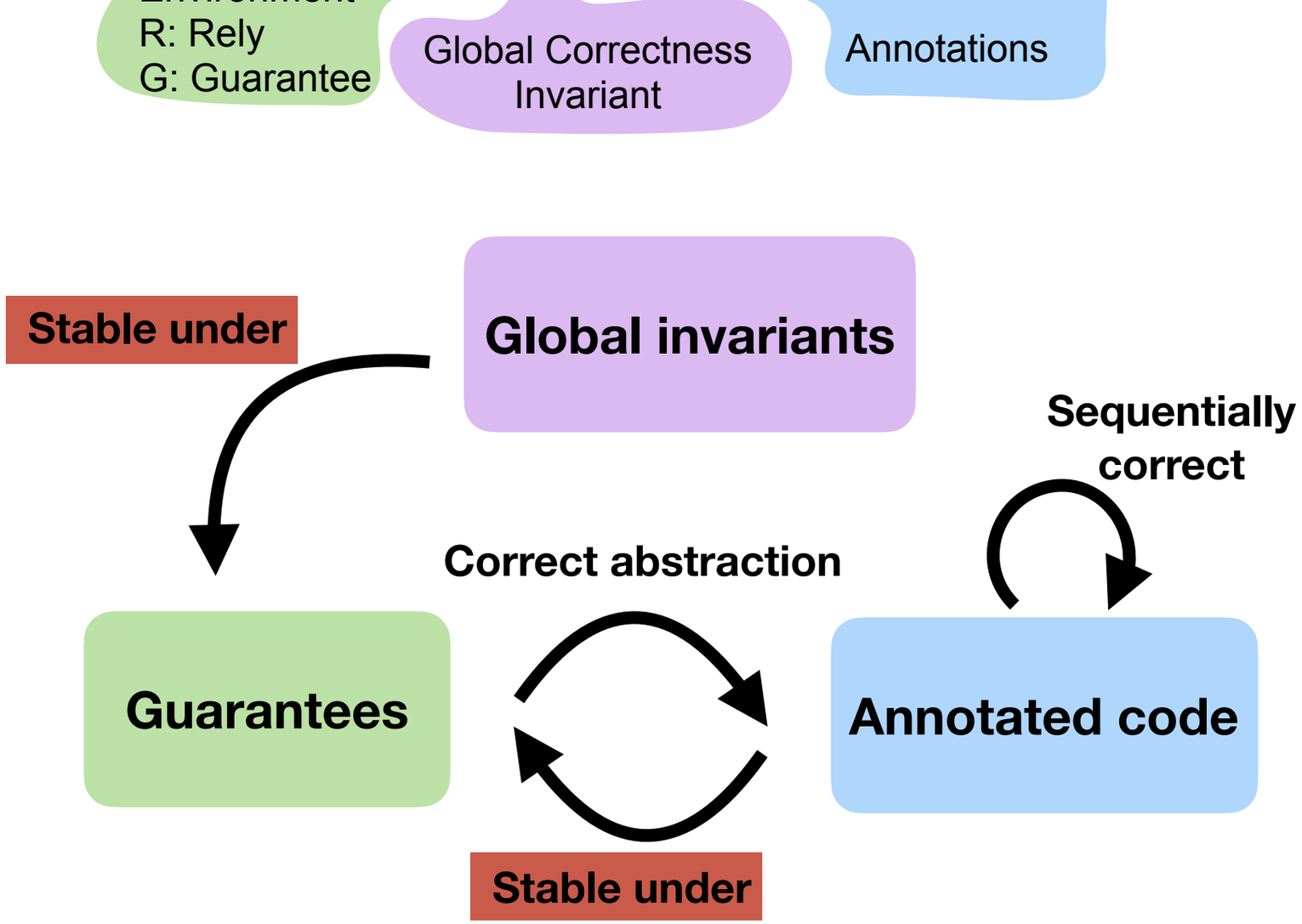
[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness Invariant

Annotations



Predicate P

stable under

interference R

$$\forall \sigma \vdash P,\ \forall \sigma',\ \sigma\ R\ \sigma' \rightarrow \sigma' \vdash P$$

Rely Guarantee reasoning

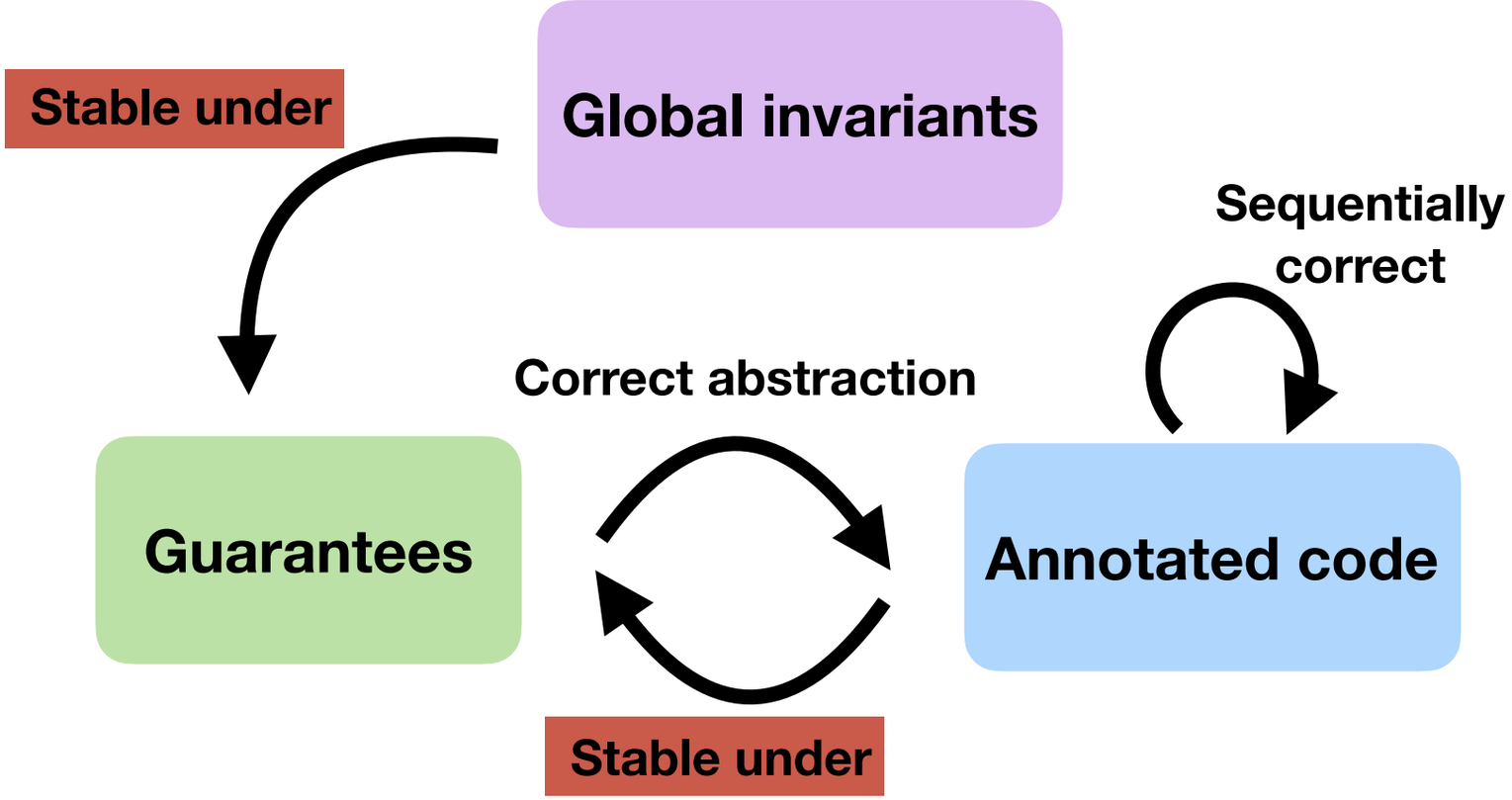
[Jones81]

$$R, G, I \vdash_t \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness Invariant

Annotations



Predicate P

stable under

interference R

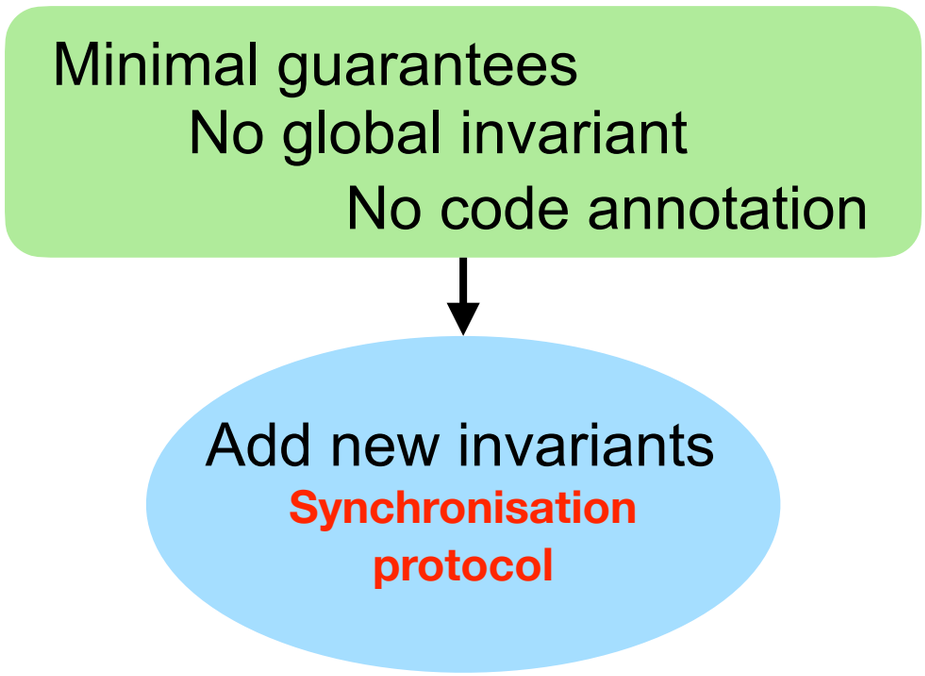
$$\forall \sigma \vdash P,\ \forall \sigma',$$
$$\sigma\ R\ \sigma' \rightarrow \sigma' \vdash P$$

(#annotations * #interferences) stability proofs required!

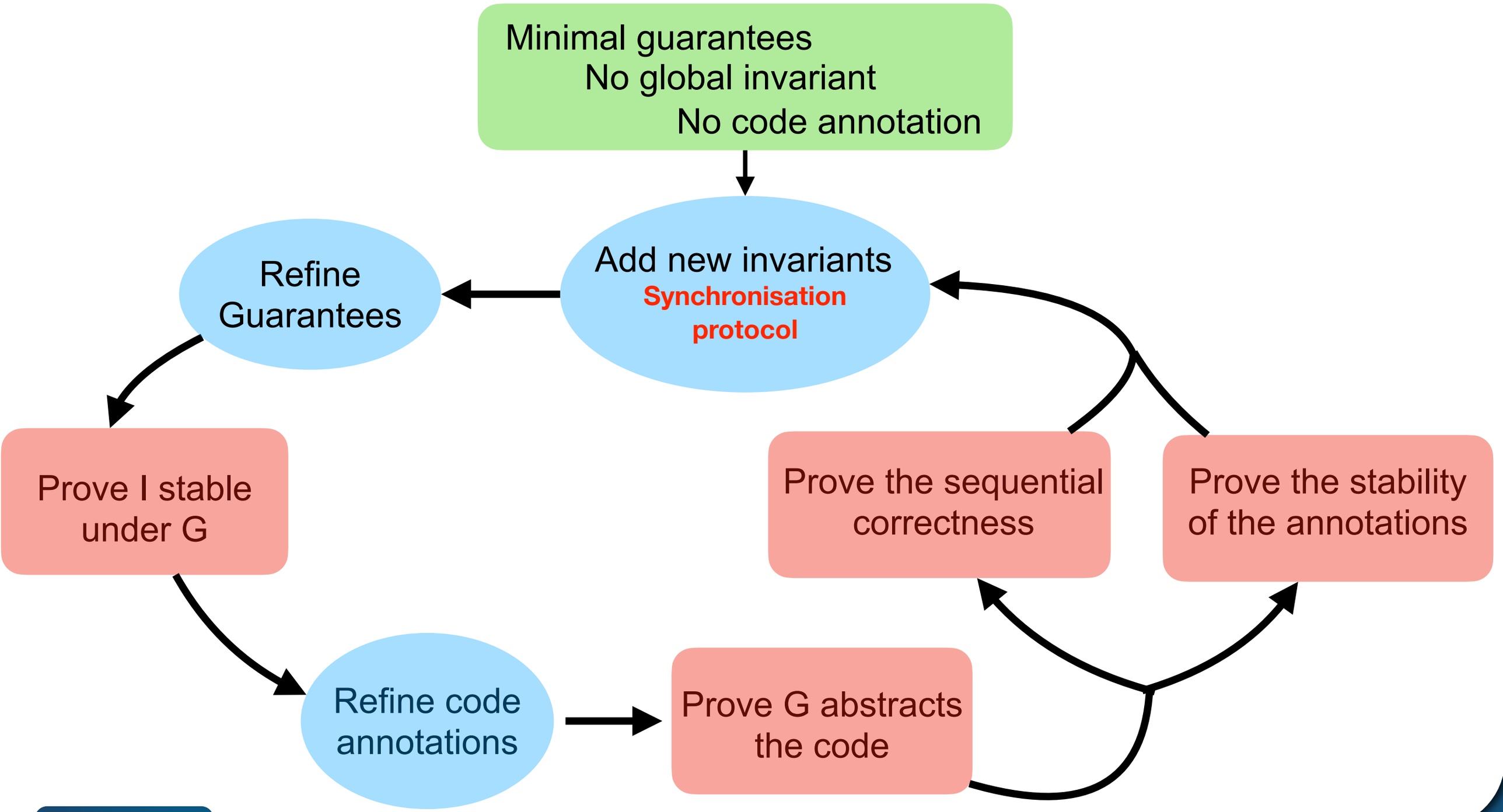
Reusing proofs incrementally

Minimal guarantees
No global invariant
No code annotation

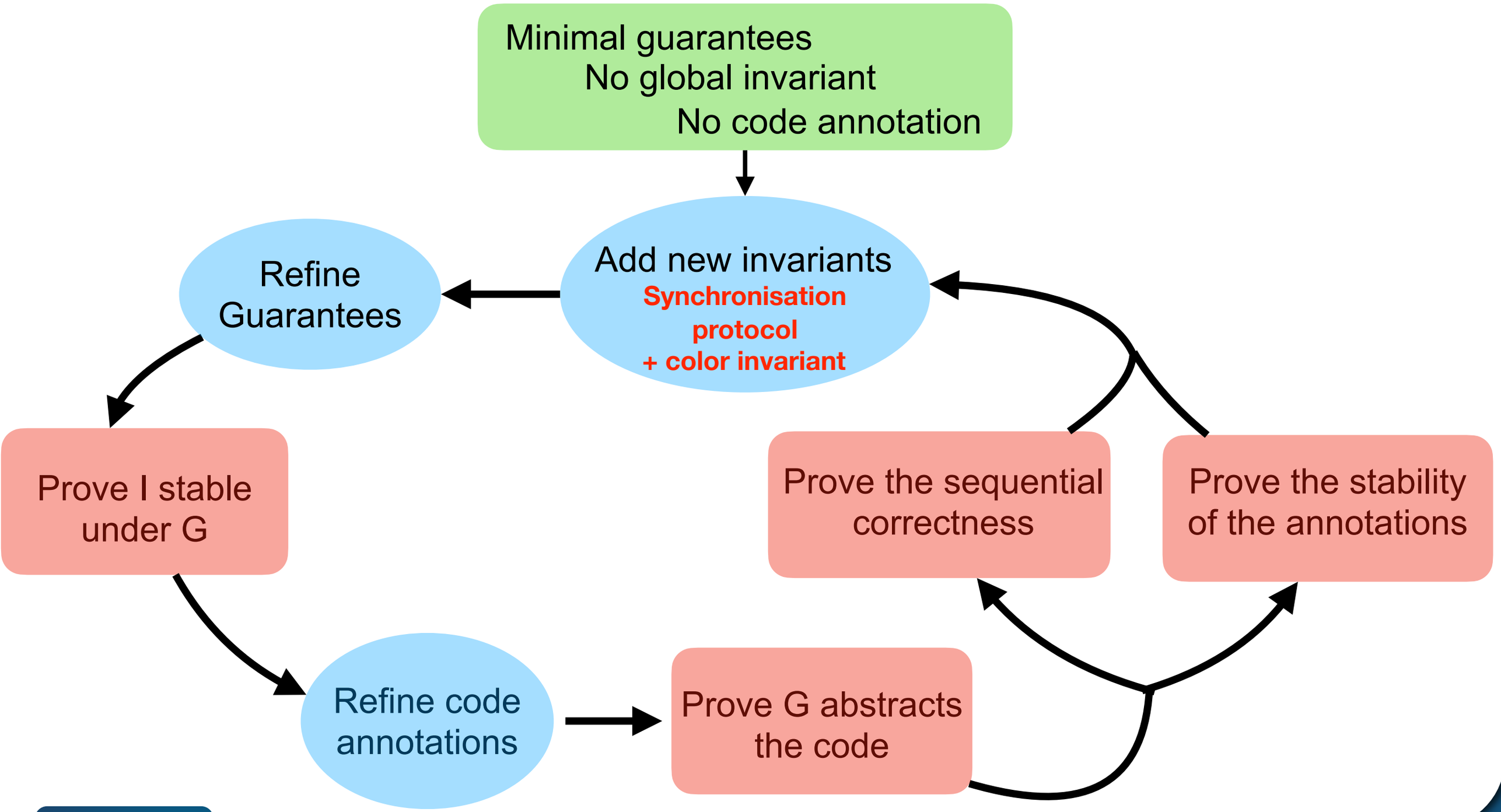
Reusing proofs incrementally



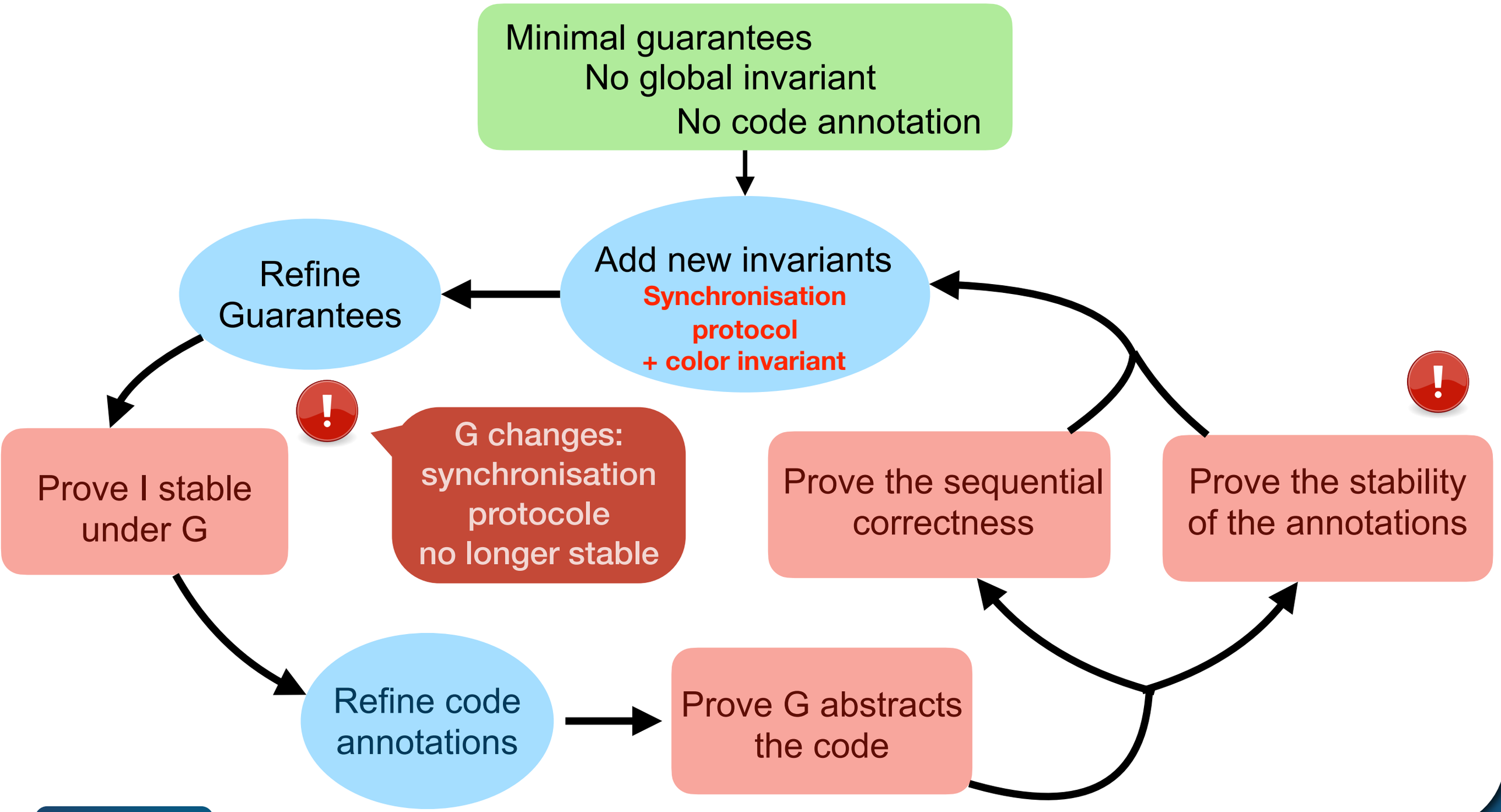
Reusing proofs incrementally



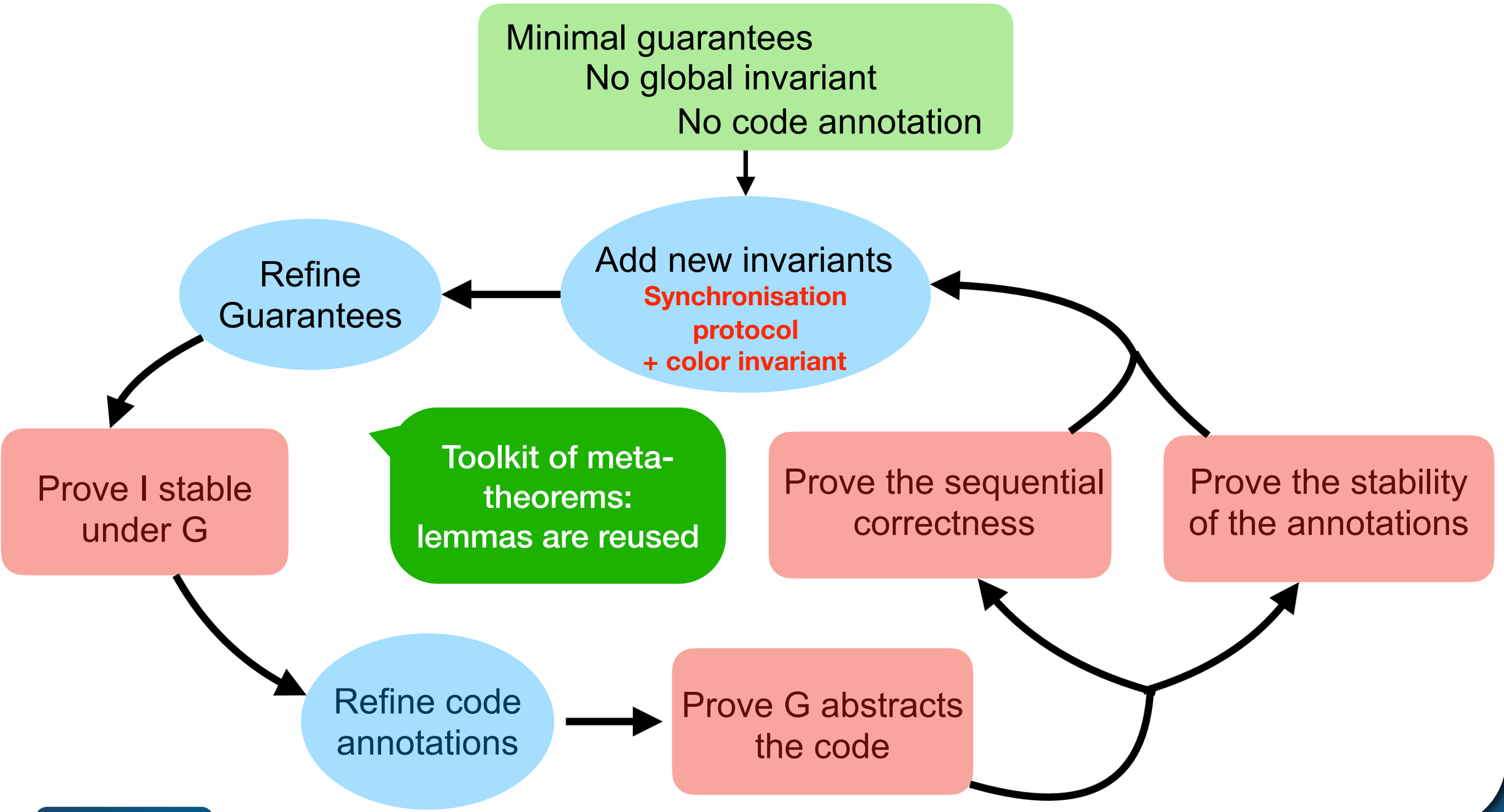
Reusing proofs incrementally



Reusing proofs incrementally



Reusing proofs incrementally



Incremental invariants of the GC

Proof = six layers of invariants (partially ordered)

5- Colors, Reachability, Correctness

black_to_white correct sweep_no_grey

pointsto_freelist trace_grey_reach_white

0- Synch Protocol

phase_val synch

late_mut_trace

sweep_asynch

1- Colors in heap

white_or_black

2- Buffers aux.

buffers_exist

in_buffer_no_null

phant_buffers_exist

3- Stage aux.

stage_val

4- Phantom aux.

hs_clear-trace

hs_flipped

flipped_clear

Conclusion

Summary

- **Dedicated IR: right level of abstraction**
  - **Abstract concurrent queues**
  - **Native support of roots, objects, freelist**
  - **Iterators**
- **RG logics + soundness theorem + incremental workflow**
- **Realistic On-The-Fly garbage collector**
  - **Significant subset of Domani et al. GC**
  - **Proofs conducted w.r.t. code's semantics**
  - **Most-General-Client theorem**
  - **Proof: incremental invariants**

Perspectives

- Dedicated IR: right level of abstraction
  - **Abstract concurrent queues**
  - Native support of roots, objects, freelist
  - Iterators
- RG logics + soundness theorem + incremental workflow
- Realistic On-The-Fly garbage collector
  - **Significant subset of Domani et al. GC**
  - Proofs conducted w.r.t. code's semantics
  - Most-General-Client theorem
  - Proof: incremental invariants

Proving atomic refinement of linearisable data-structures —> ongoing

We left out orthogonal optimisations ex : Generational GC