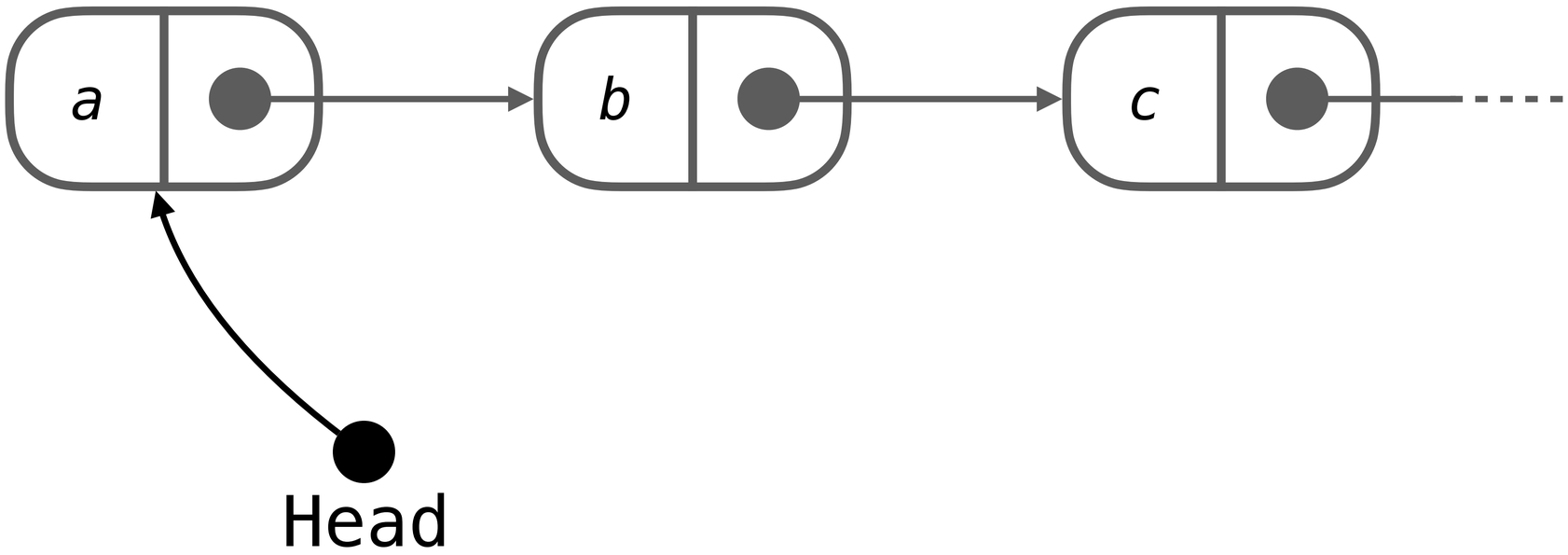
Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation

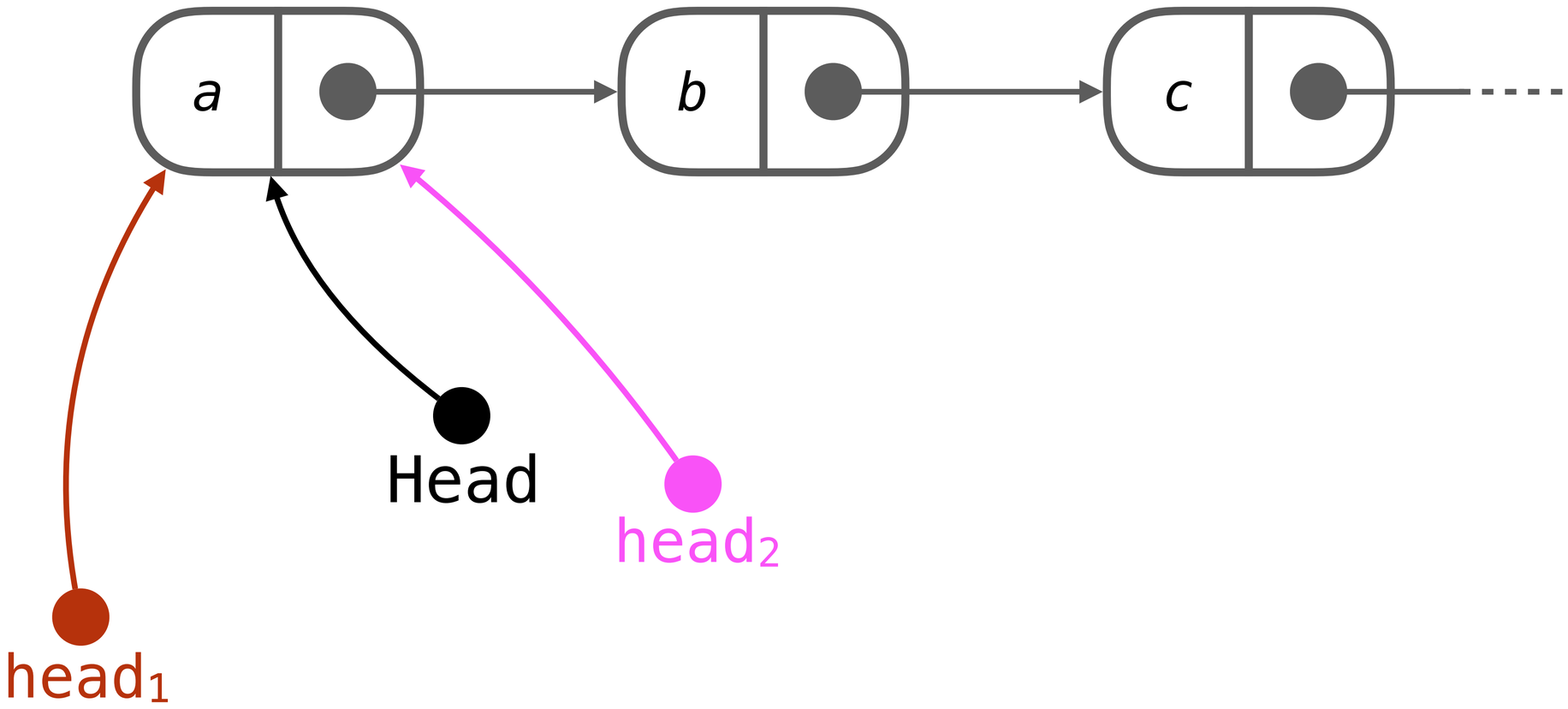
Roland Meyer and Sebastian Wolff

TU Braunschweig, Germany

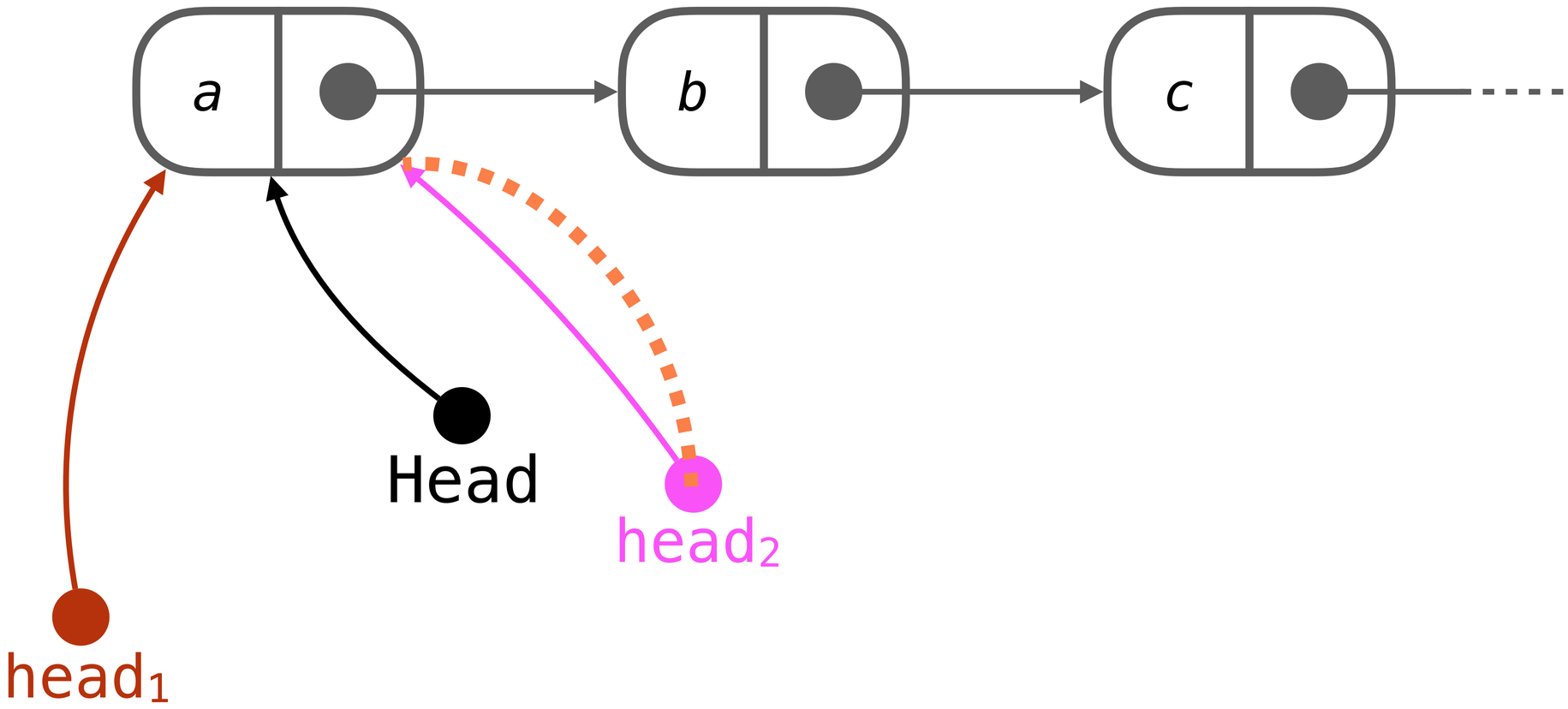
Lock-free Queue



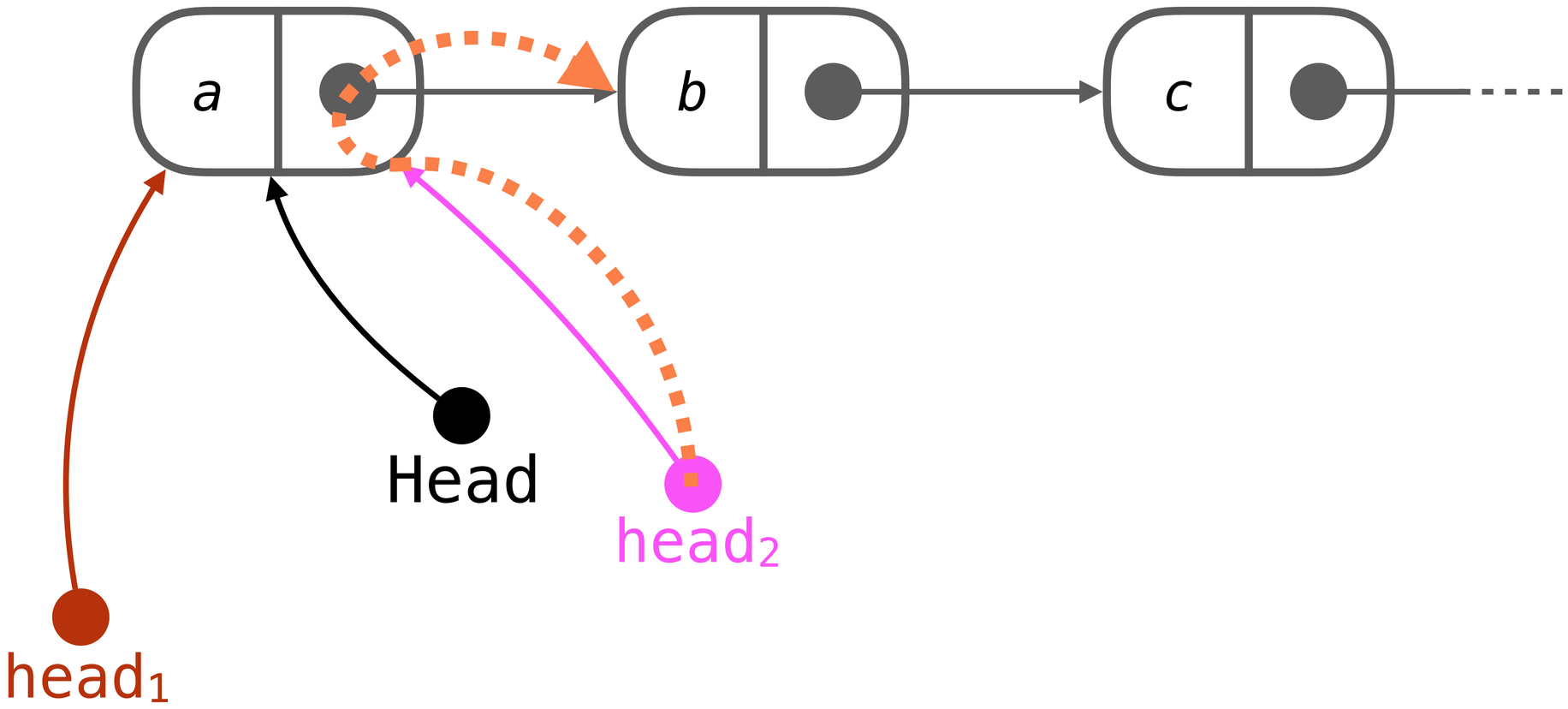
Lock-free Queue



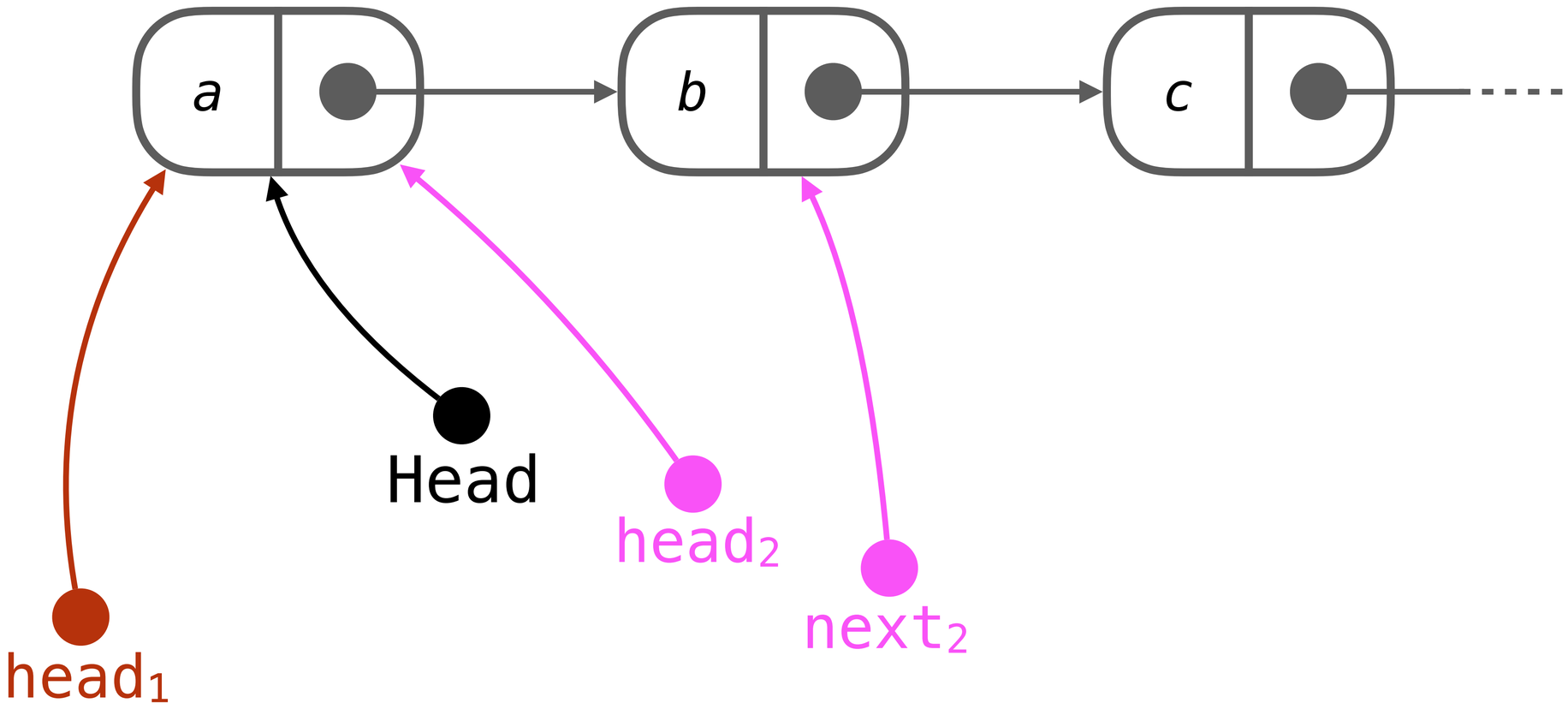
Lock-free Queue



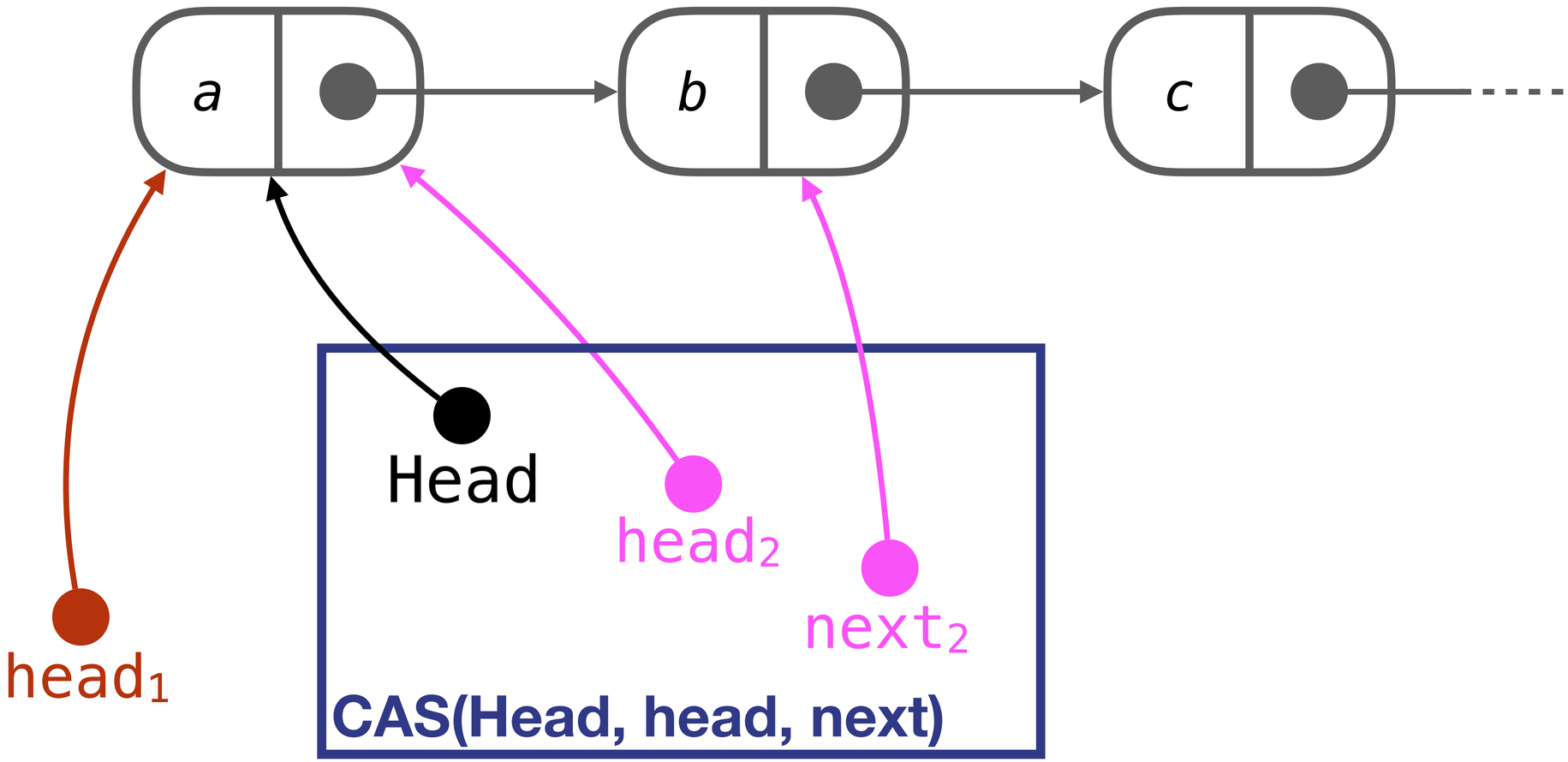
Lock-free Queue



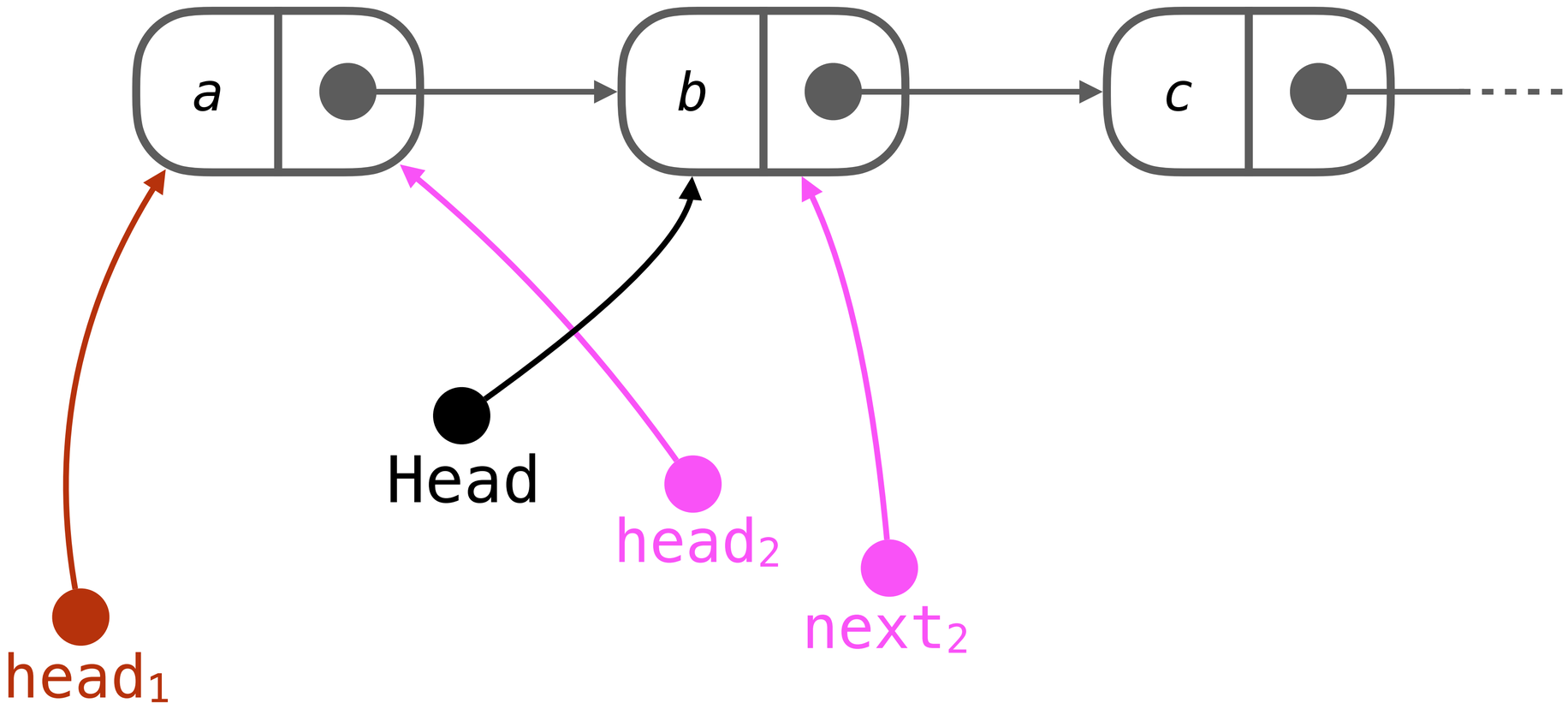
Lock-free Queue



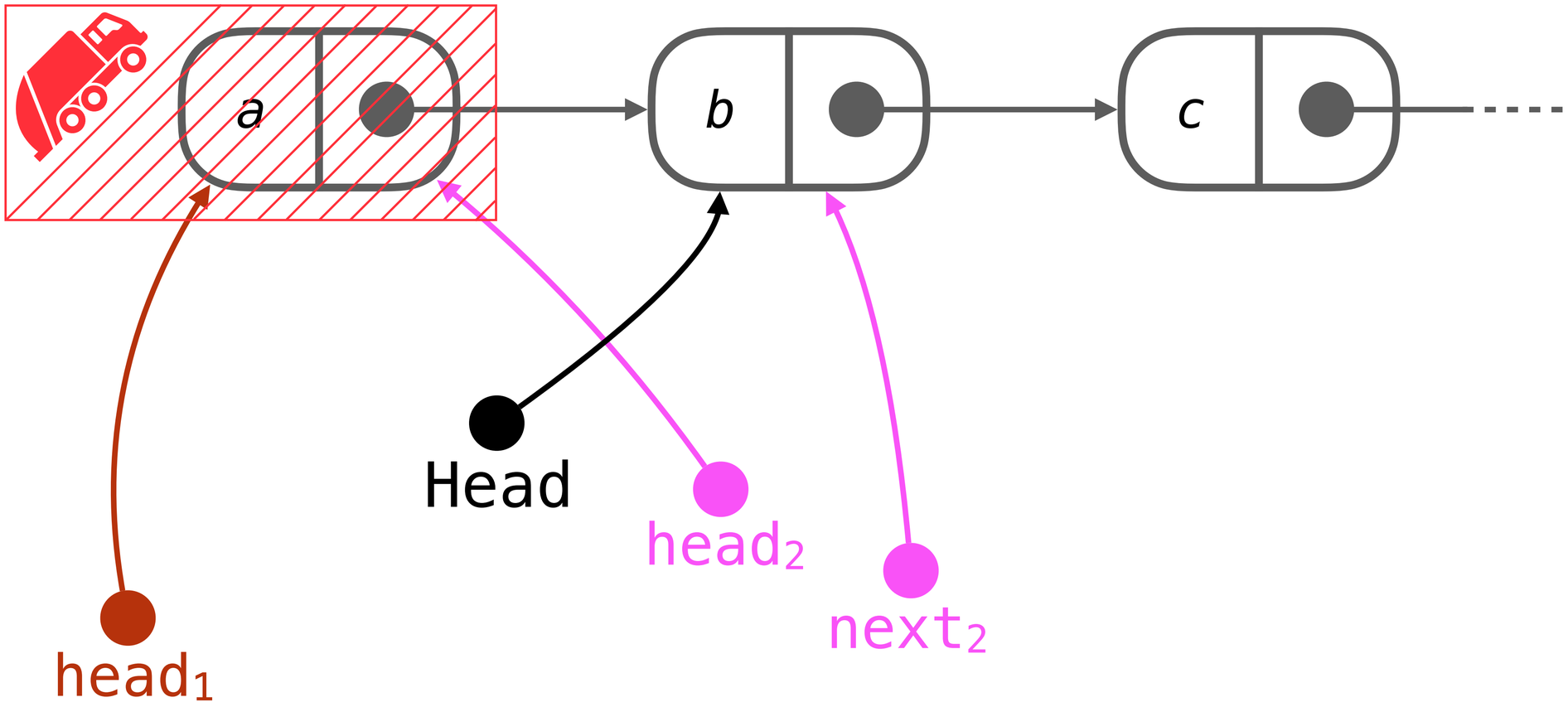
Lock-free Queue



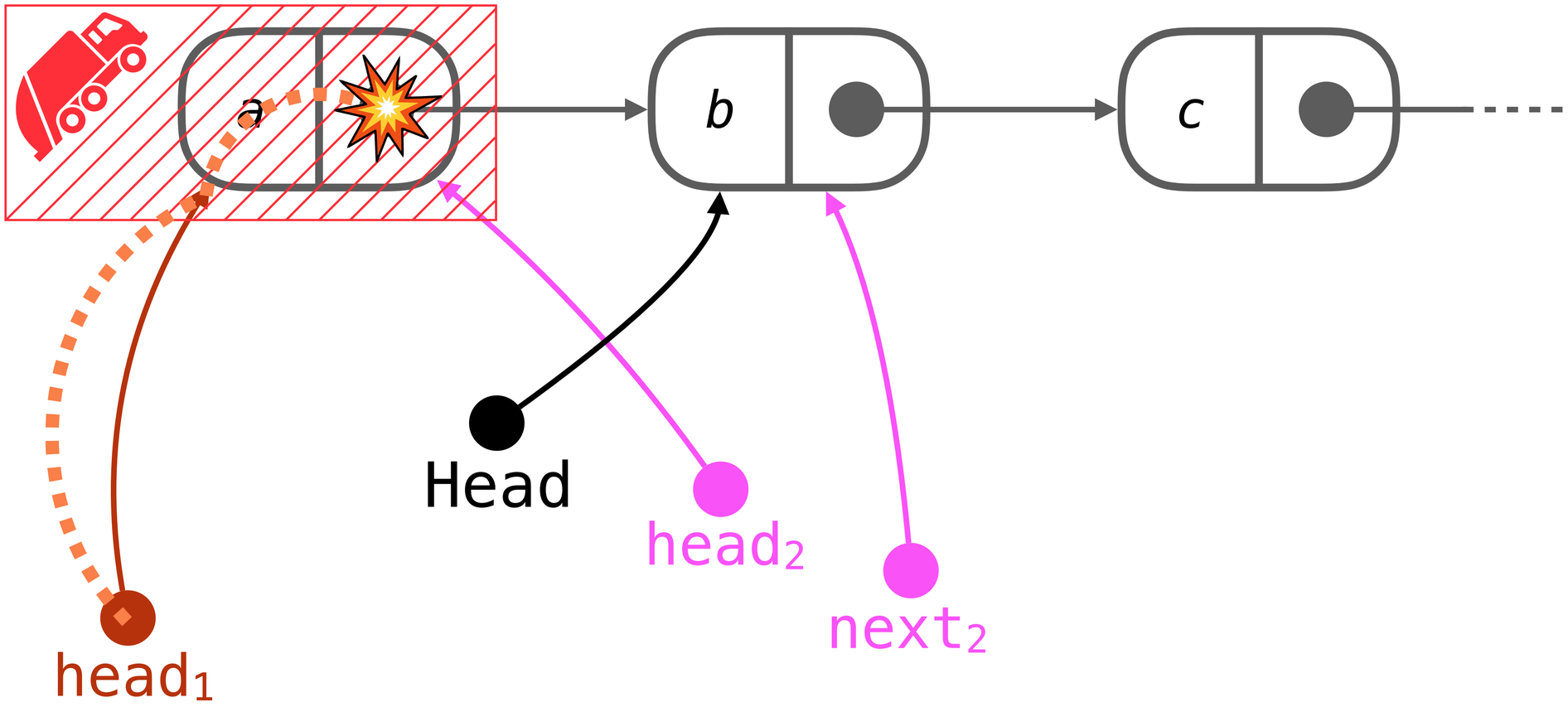
Lock-free Queue



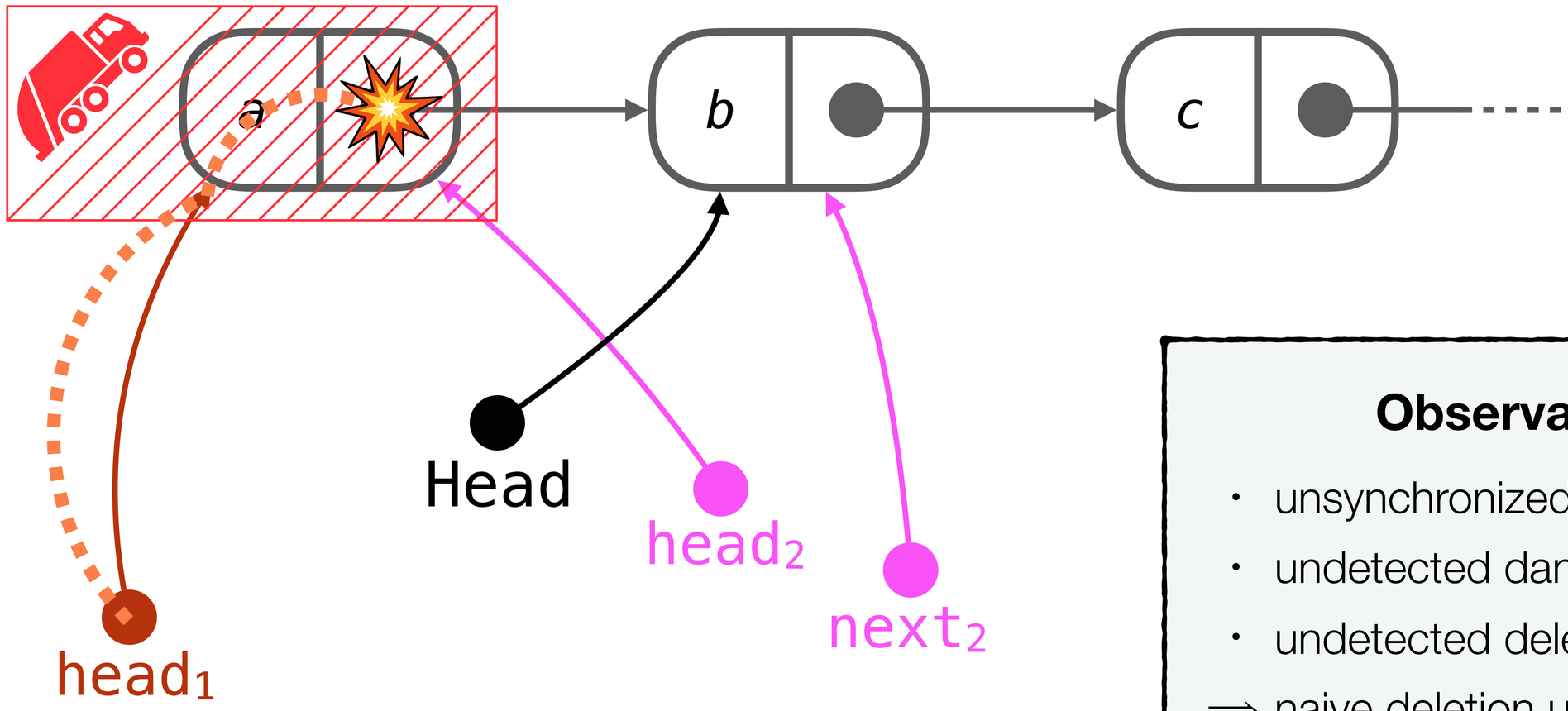
Lock-free Queue



Lock-free Queue



Lock-free Queue

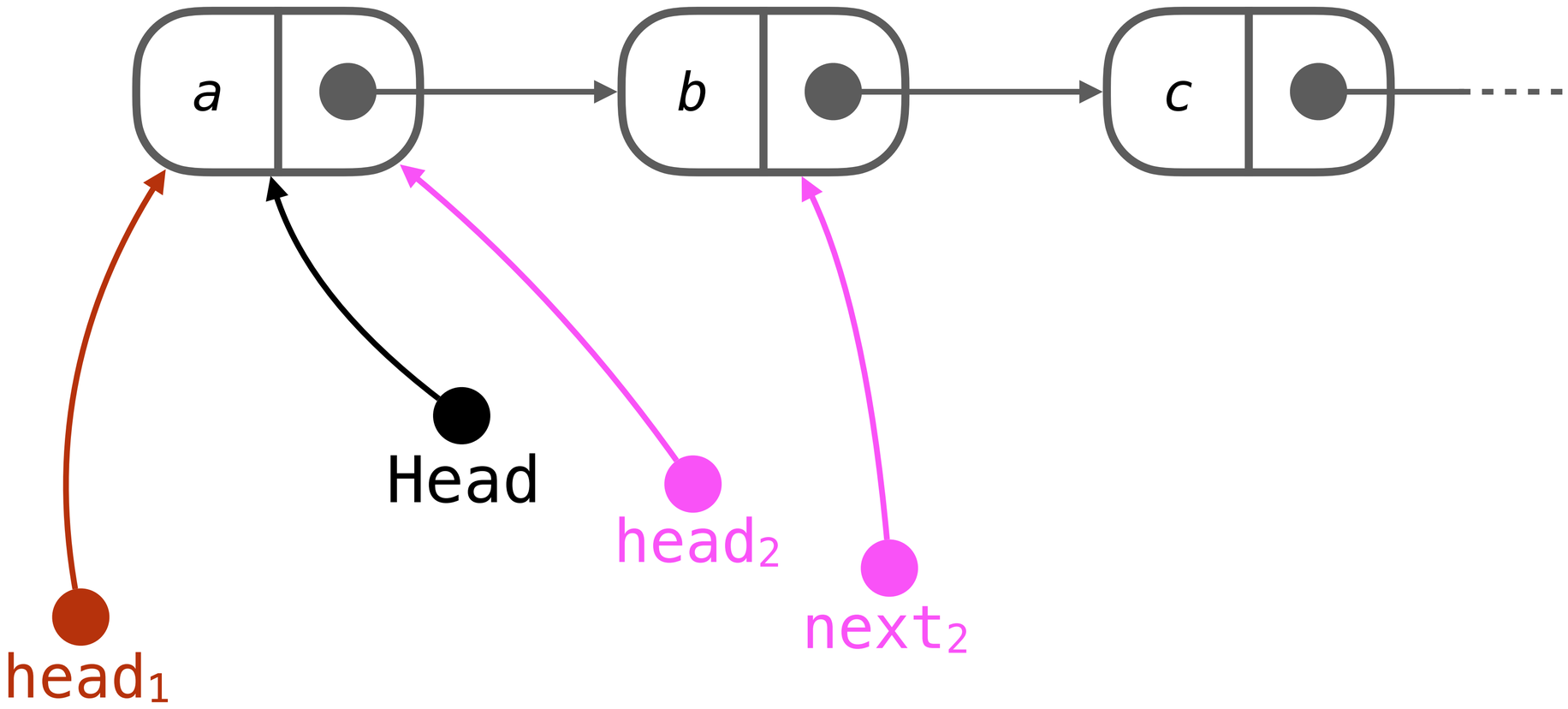


Observation

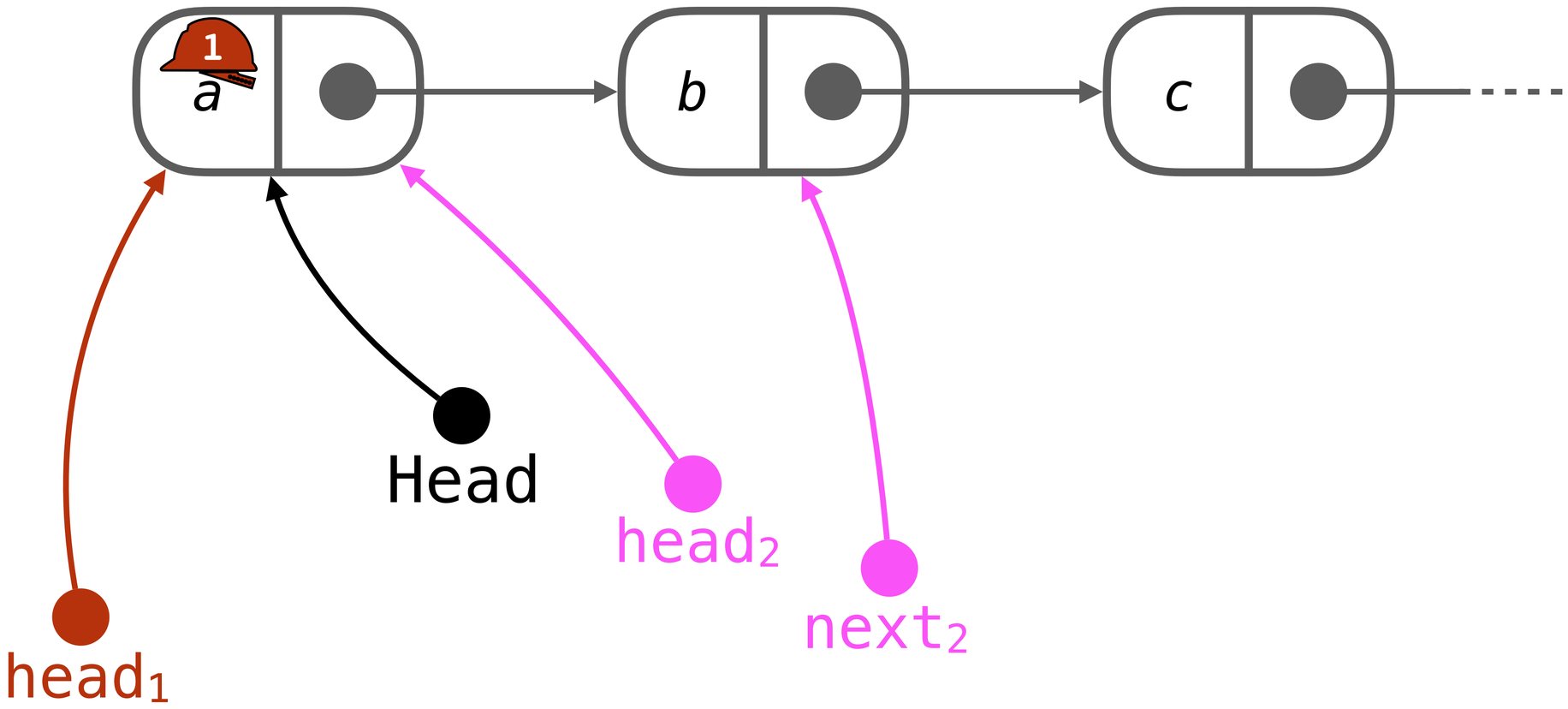
- unsynchronized traversal
- undetected dangling readers
- undetected deletion

$\Longrightarrow$ naive deletion unsafe

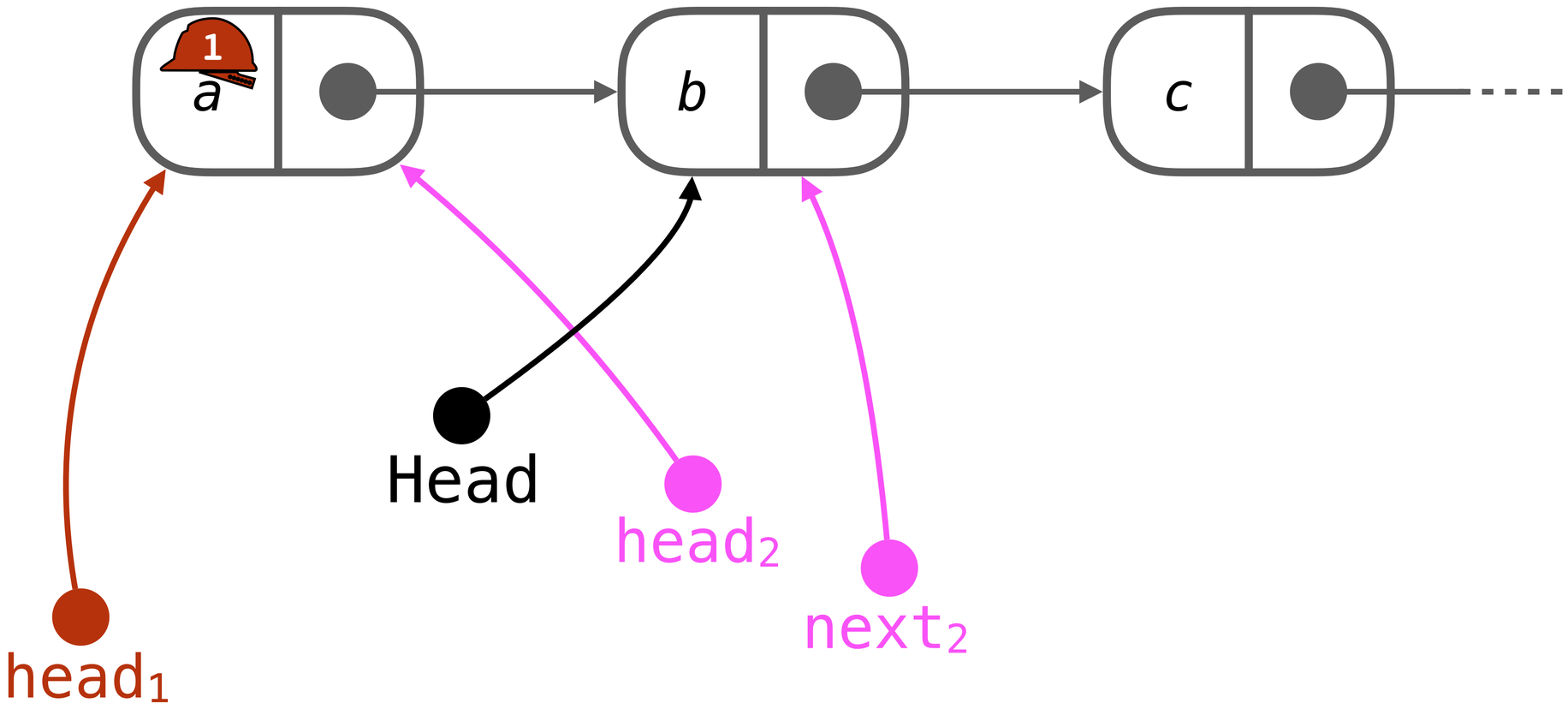
Safe Memory Reclamation (SMR)



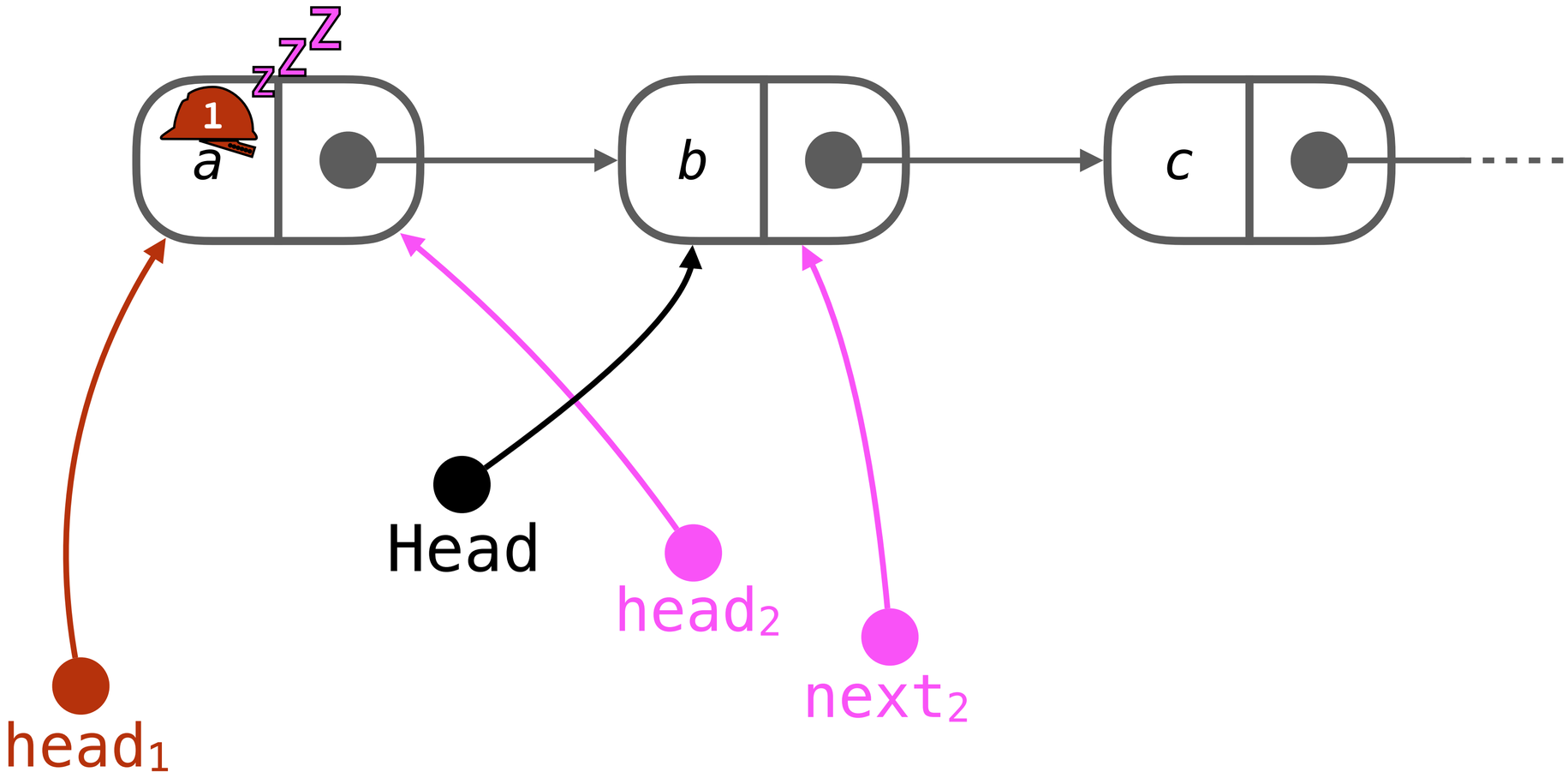
Safe Memory Reclamation (SMR)



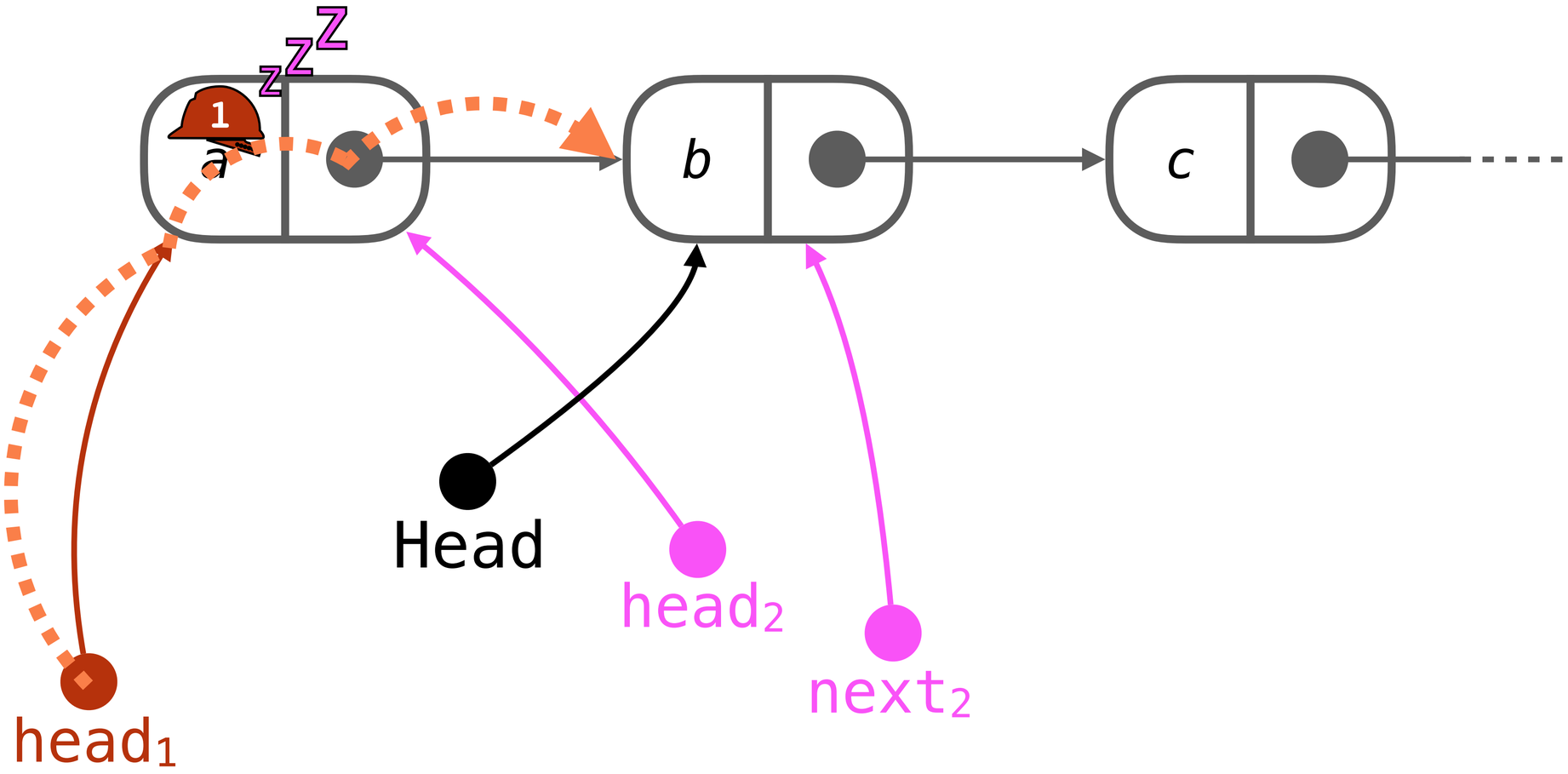
Safe Memory Reclamation (SMR)



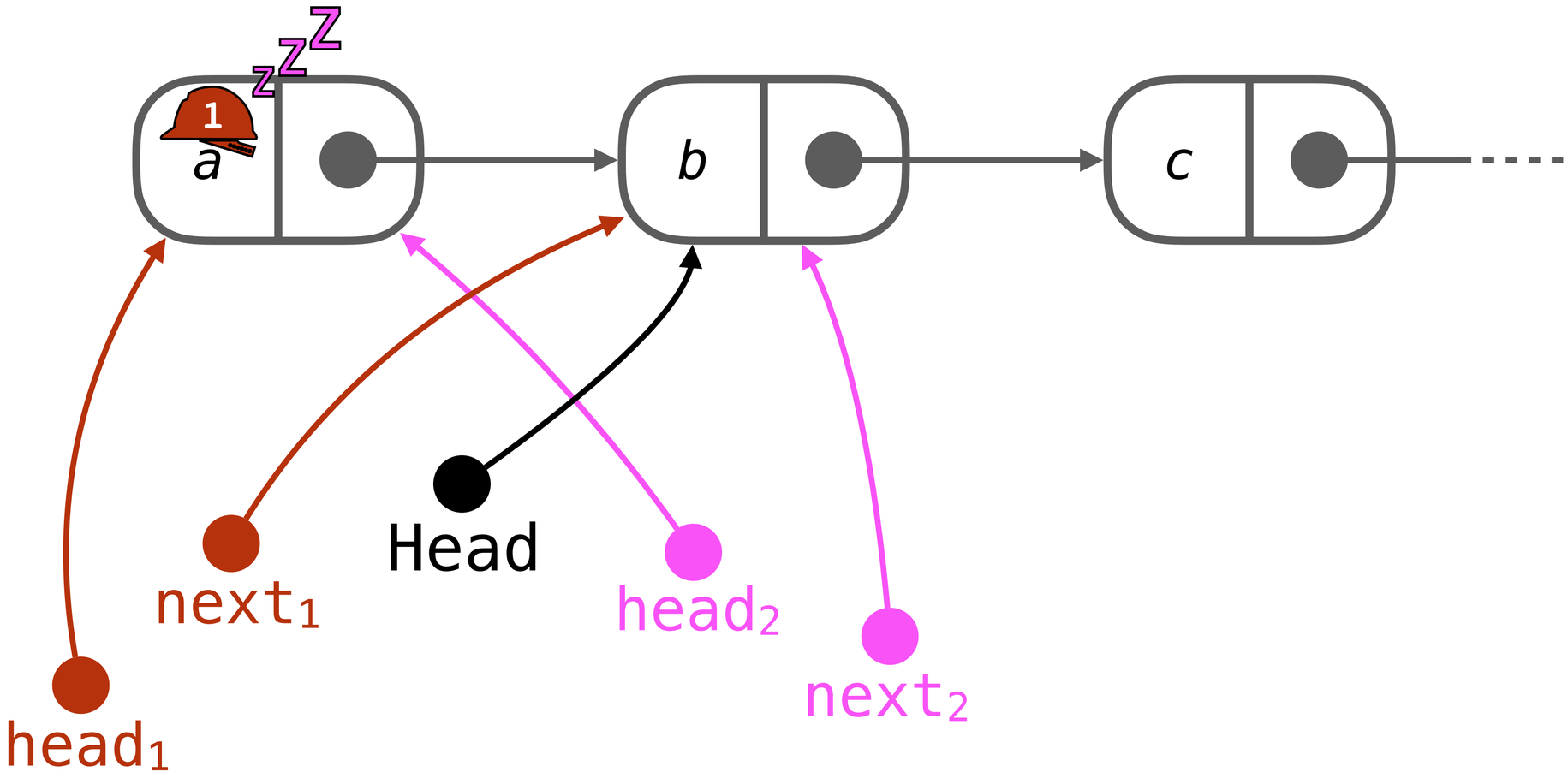
Safe Memory Reclamation (SMR)



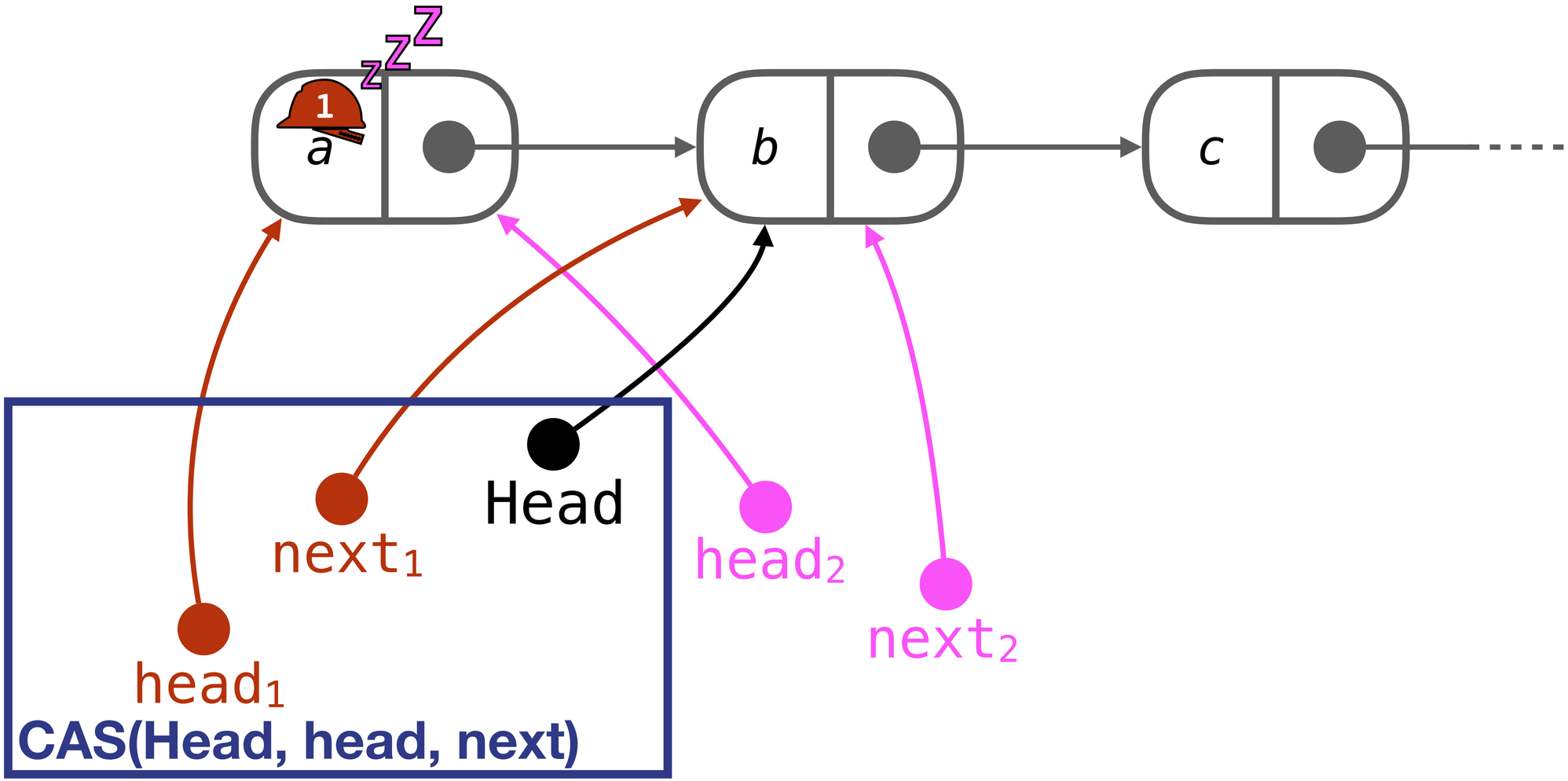
Safe Memory Reclamation (SMR)



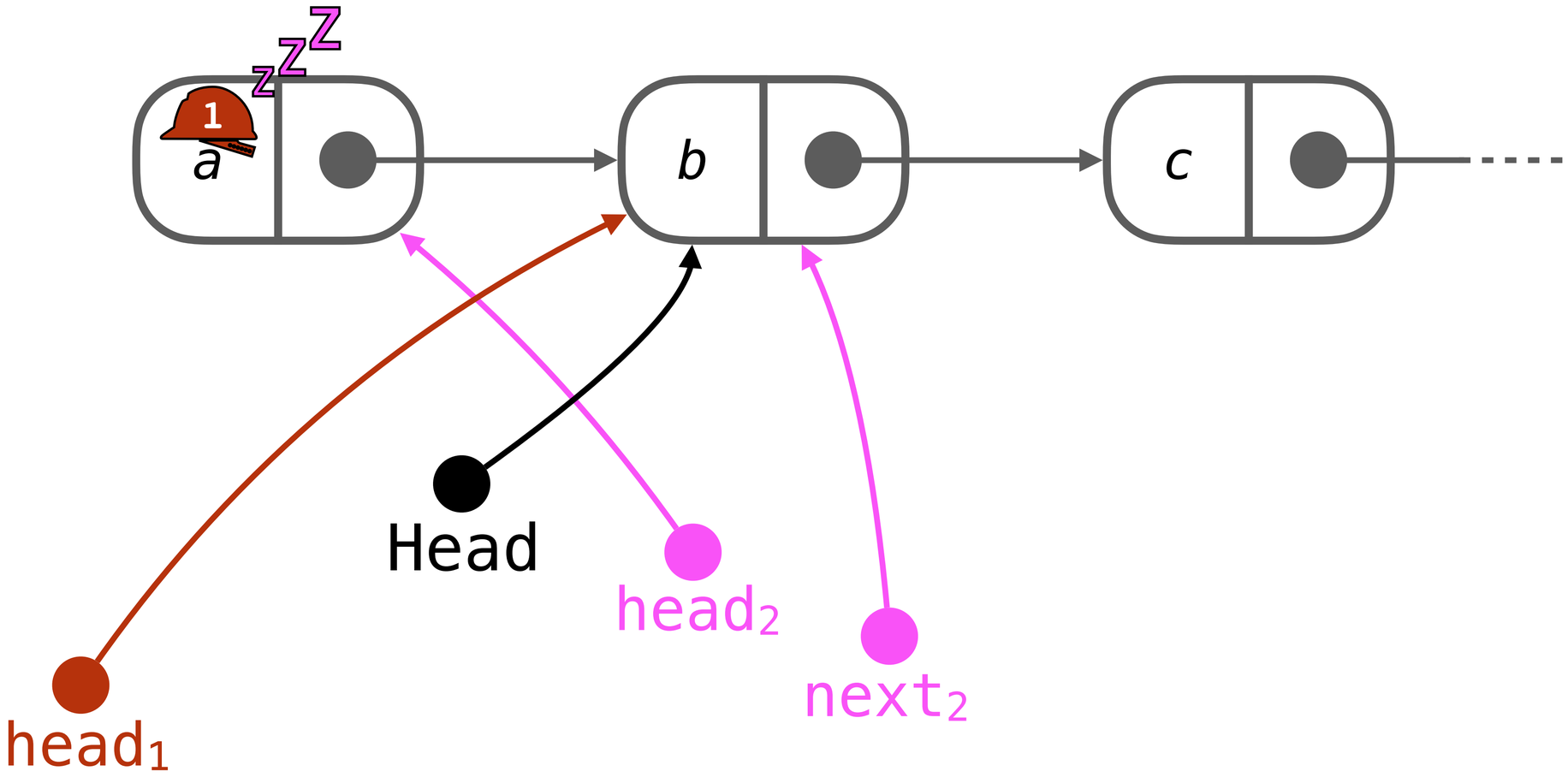
Safe Memory Reclamation (SMR)



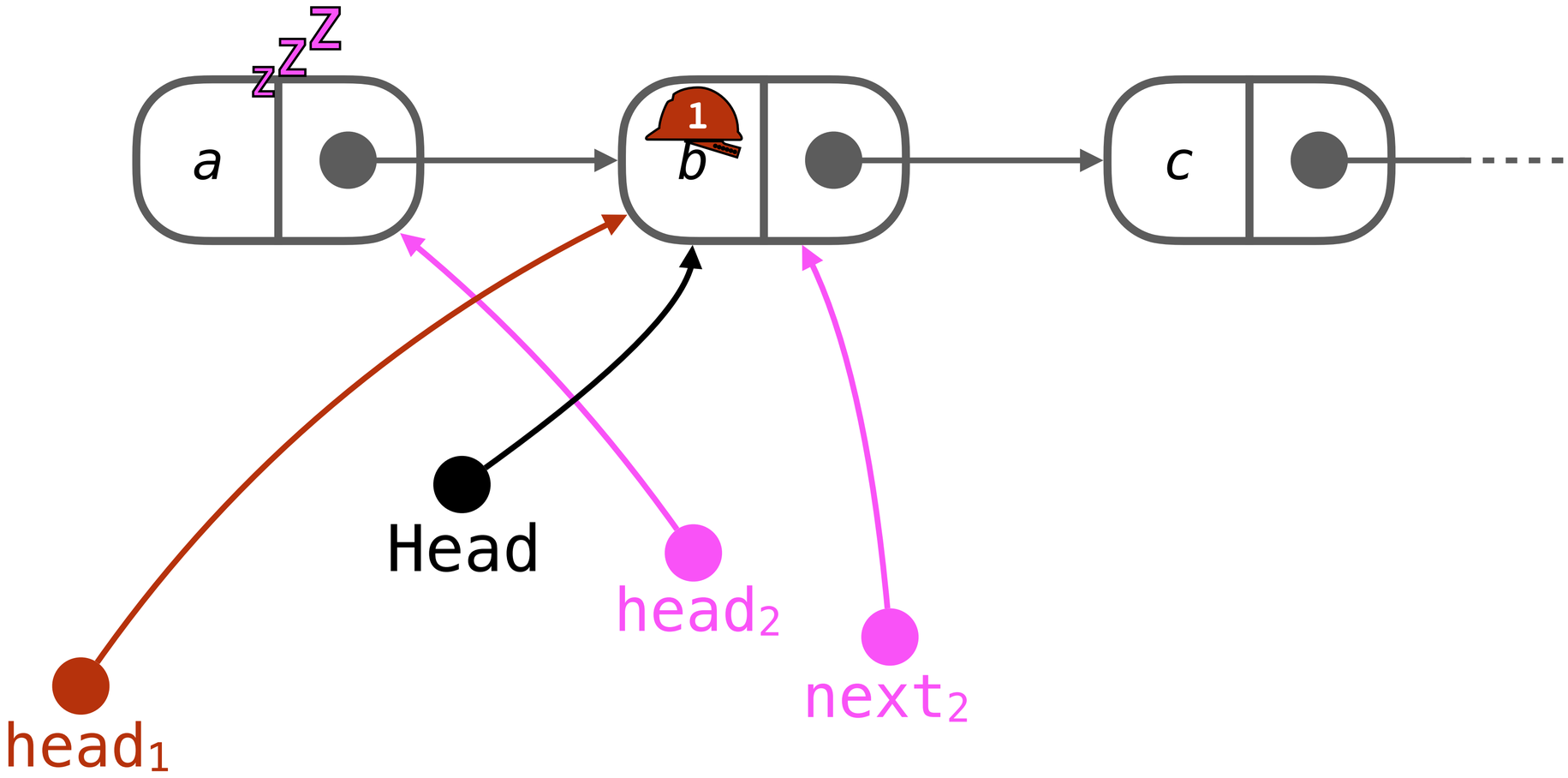
Safe Memory Reclamation (SMR)



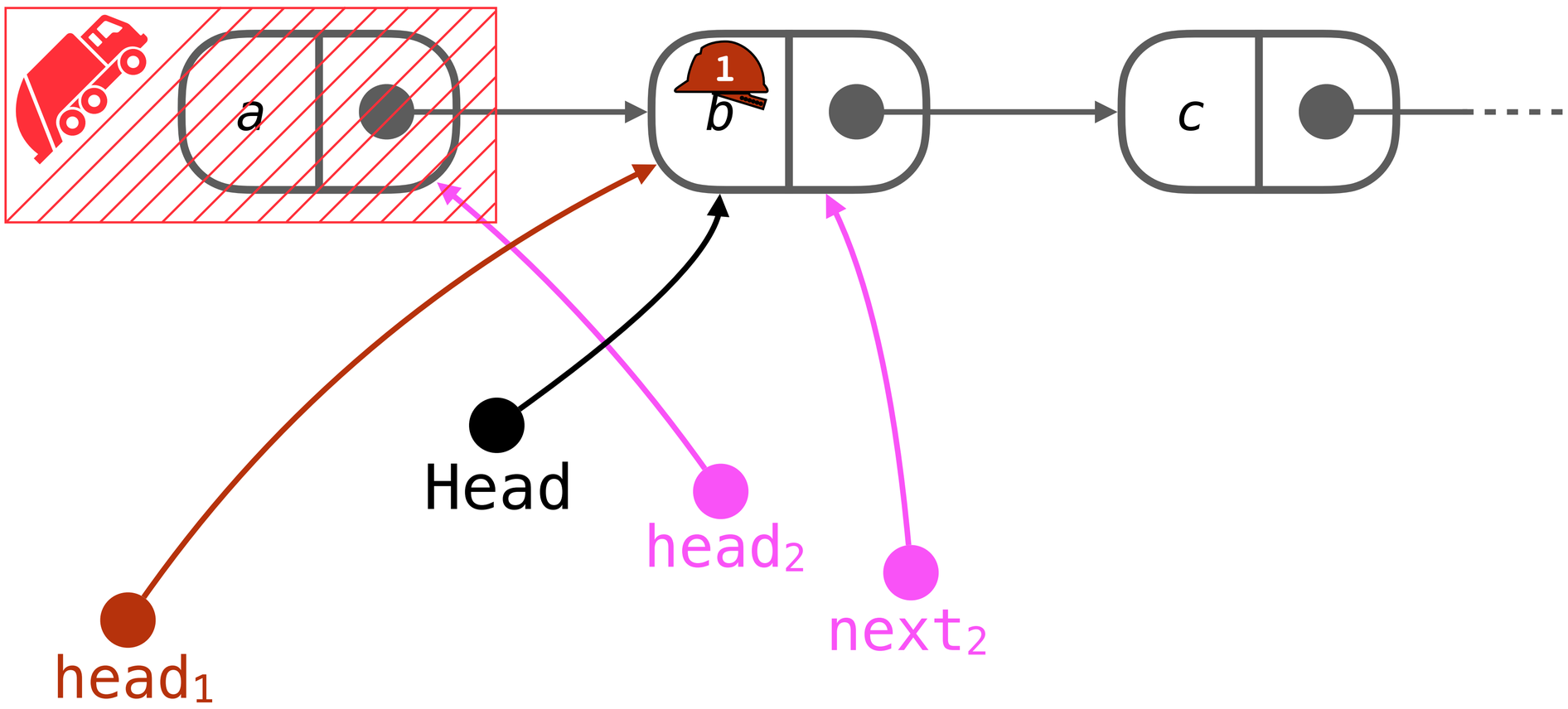
Safe Memory Reclamation (SMR)



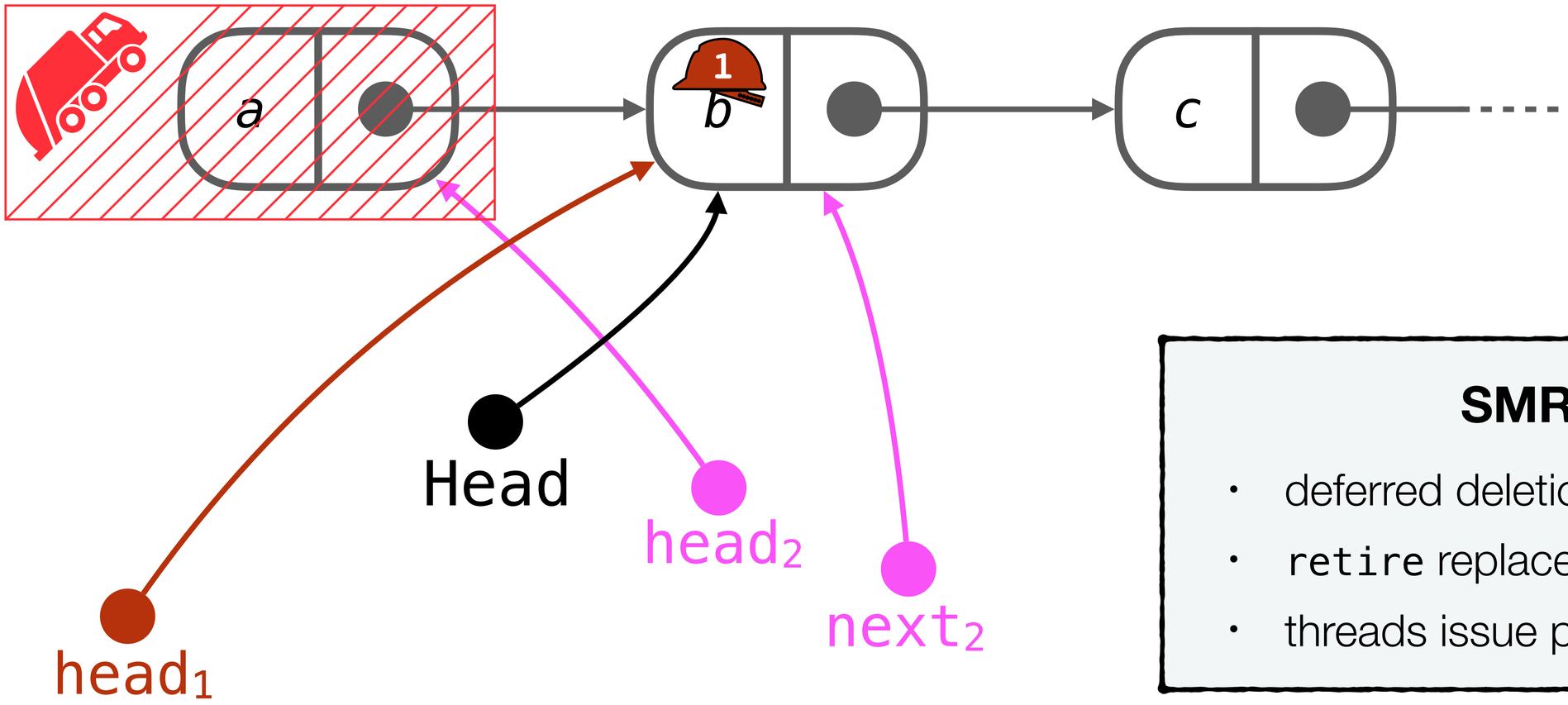
Safe Memory Reclamation (SMR)



Safe Memory Reclamation (SMR)



Safe Memory Reclamation (SMR)



SMR

- deferred deletion
- `retire` replaces `delete`
- threads issue protections

Lock-Free Queue

```
struct Node {
    data_t data;
    Node* node;
}

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;

        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}

data_t dequeue() {
    while (true) {
        Node* head = Head;

        Node* tail = Tail;
        Node* next = head->next;

        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {

                return data;
            }
        }
    }
}
```

queue

lock-free

GC

37 LOC

Lock-Free Queue

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

lock-free **SMR**

37+6 LOC

Lock-Free Queue

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

lock-free

SMR

37+6 LOC

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}
```

```
void protect0(Node* ptr) {
    myRec->hp0 = ptr;
}

void protect1(Node* ptr) {
    myRec->hp1 = ptr;
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
    }
}
```

lock-free

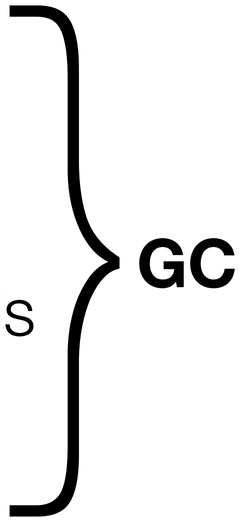
HP

42 LOC

no reclamation

Verification Challenges

1. unbounded shared heap
2. unbounded data domain
3. unbounded number of threads
4. fine-grained concurrency



Verification Challenges

1. unbounded shared heap
2. unbounded data domain
3. unbounded number of threads
4. fine-grained concurrency

GC

Automated Verification for GC

- Vafeiadis, CAV'10 + VMCAI'10
- Abdulla et al., TACAS'13
- Zhu et al., CAV'15
- Abdulla et al., SAS'16 + ESOP'18

Verification Challenges

1. unbounded shared heap
2. unbounded data domain
3. unbounded number of threads
4. fine-grained concurrency

GC

Automated Verification for GC

- Vafeiadis, CAV'10 + VMCAI'10
- Abdulla et al., TACAS'13
- Zhu et al., CAV'15
- Abdulla et al., SAS'16 + ESOP'18

5. **DS code + SMR code**
6. **reclamation**

Verification Challenges

1. unbounded shared heap
2. unbounded data domain
3. unbounded number of threads
4. fine-grained concurrency

GC

Automated Verification for GC

- Vafeiadis, CAV'10 + VMCAI'10
- Abdulla et al., TACAS'13
- Zhu et al., CAV'15
- Abdulla et al., SAS'16 + ESOP'18

5. **DS code + SMR code**
6. **reclamation**

GC-techniques do not work (well)!

In particular: ownership reasoning

Verification Challenges

Data structure + **SMR impl** $\models$ **Property**

Verification Challenges

SMR impl $\models$ **SMR spec**

Data structure + **SMR spec** $\models$ **Property**

Data structure + **SMR impl** $\models$ **Property**

Compositional Verification

```
struct Node {
    data_t data;
    Node* node;
}

shared:
    Node* Head;
    Node* Tail;

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

SMR

lock-free

37+6 LOC

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}

void protect0(Node* ptr) {
    myRec->hp0 = ptr;
}

void protect1(Node* ptr) {
    myRec->hp1 = ptr;
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
    }
}
```

lock-free

HP

no reclamation

42 LOC

Compositional Verification

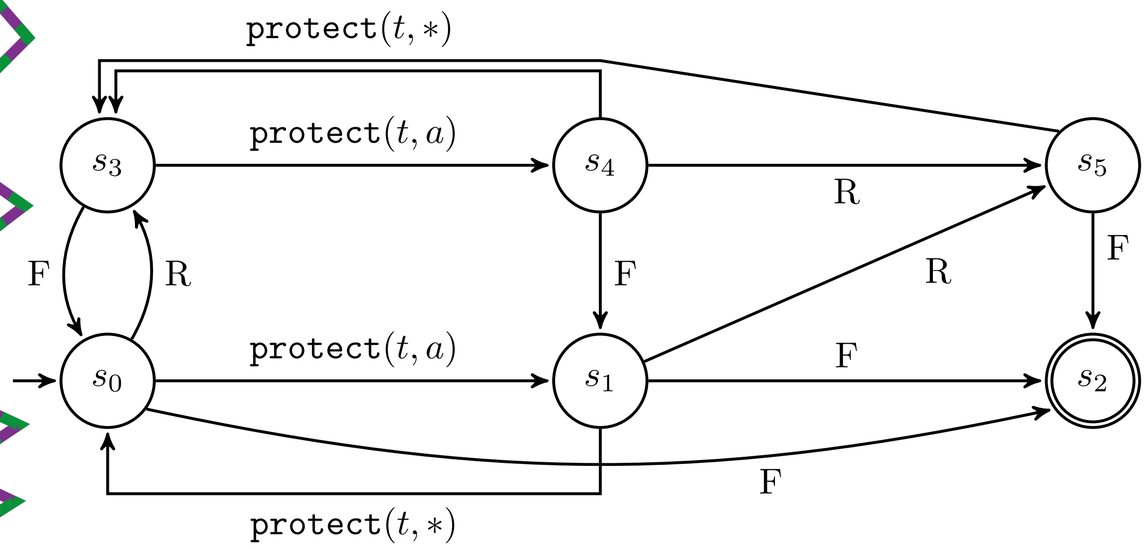
```
struct Node {
    data_t data;
    Node* node;
}

shared:
    Node* Head;
    Node* Tail;

void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

lock-free

SMR

37+6 LOC



$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$ $R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$

automaton

Verification Challenges

SMR impl $\models$ **SMR spec**

Data structure + **SMR spec** $\models$ **Property**

Data structure + **SMR impl** $\models$ **Property**

Verification Challenges

SMR impl $\models$ **SMR spec**

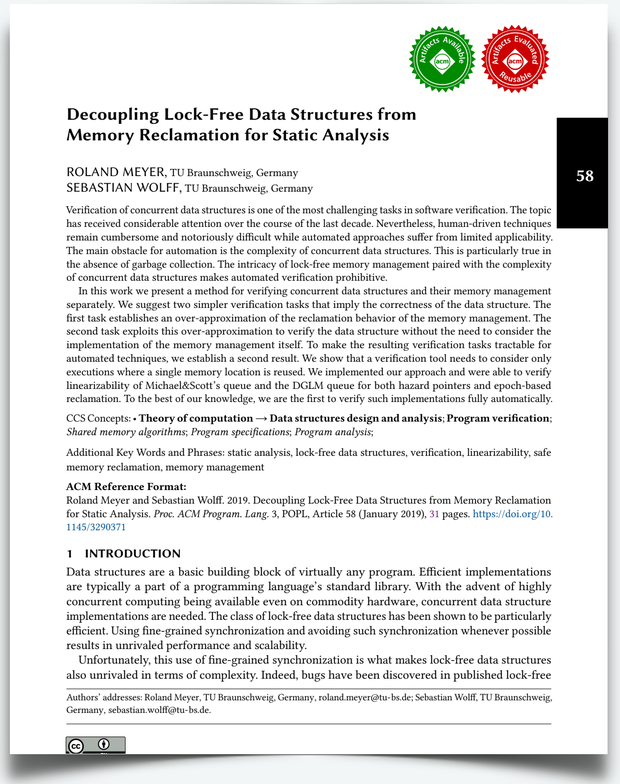
~~**Data structure** + **SMR spec** $\models$ **Property**~~

Data structure + **Garbage Collection** $\models$ **Property**

Data structure + **SMR spec** $\models$ **Pointer Race Freedom**

Data structure + **SMR impl** $\models$ **Property**

Sound by [POPL'19]:



Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis

58

ROLAND MEYER, TU Braunschweig, Germany
SEBASTIAN WOLFF, TU Braunschweig, Germany

Verification of concurrent data structures is one of the most challenging tasks in software verification. The topic has received considerable attention over the course of the last decade. Nevertheless, human-driven techniques remain cumbersome and notoriously difficult while automated approaches suffer from limited applicability. The main obstacle for automation is the complexity of concurrent data structures. This is particularly true in the absence of garbage collection. The intricacy of lock-free memory management paired with the complexity of concurrent data structures makes automated verification prohibitive.

In this work we present a method for verifying concurrent data structures and their memory management separately. We suggest two simpler verification tasks that imply the correctness of the data structure. The first task establishes an over-approximation of the reclamation behavior of the memory management. The second task exploits this over-approximation to verify the data structure without the need to consider the implementation of the memory management itself. To make the resulting verification tasks tractable for automated techniques, we establish a second result. We show that a verification tool needs to consider only executions where a single memory location is reused. We implemented our approach and were able to verify linearizability of Michael&Scott's queue and the DGLM queue for both hazard pointers and epoch-based reclamation. To the best of our knowledge, we are the first to verify such implementations fully automatically.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; **Program verification**; *Shared memory algorithms*; *Program specifications*; *Program analysis*;

Additional Key Words and Phrases: static analysis, lock-free data structures, verification, linearizability, safe memory reclamation, memory management

ACM Reference Format:
Roland Meyer and Sebastian Wolff. 2019. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 58 (January 2019), 31 pages. https://doi.org/10.1145/3290371

1 INTRODUCTION

Data structures are a basic building block of virtually any program. Efficient implementations are typically a part of a programming language's standard library. With the advent of highly concurrent computing being available even on commodity hardware, concurrent data structure implementations are needed. The class of lock-free data structures has been shown to be particularly efficient. Using fine-grained synchronization and avoiding such synchronization whenever possible results in unrivaled performance and scalability.

Unfortunately, this use of fine-grained synchronization is what makes lock-free data structures also unrivaled in terms of complexity. Indeed, bugs have been discovered in published lock-free

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Sebastian Wolff, TU Braunschweig, Germany, sebastian.wolff@tu-bs.de.

Verification Challenges

SMR impl ⊨ SMR spec

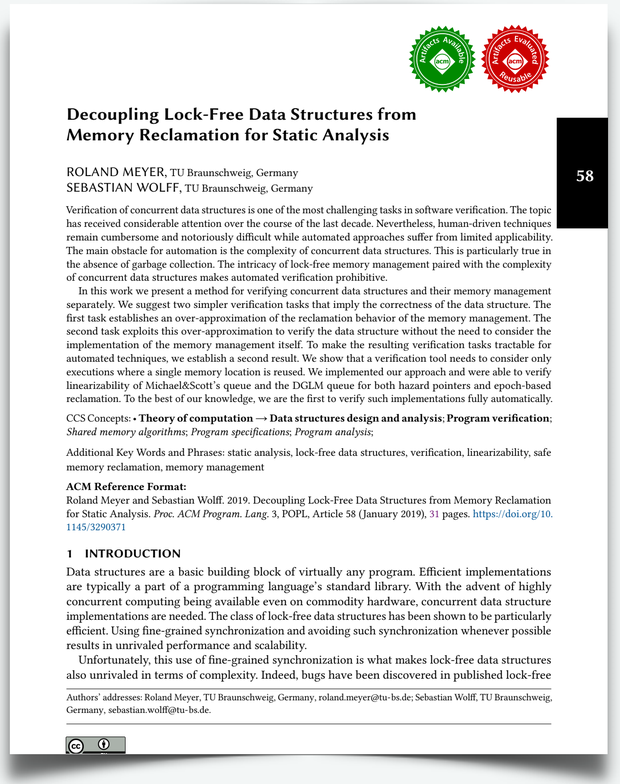
~~Data structure + SMR spec ⊨ Property~~

Data structure + Garbage Collection ⊨ Property

Data structure + SMR spec ⊨ Pointer Race Freedom

Data structure + SMR impl ⊨ Property

Sound by [POPL'19]:



58

Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis

ROLAND MEYER, TU Braunschweig, Germany
SEBASTIAN WOLFF, TU Braunschweig, Germany

Verification of concurrent data structures is one of the most challenging tasks in software verification. The topic has received considerable attention over the course of the last decade. Nevertheless, human-driven techniques remain cumbersome and notoriously difficult while automated approaches suffer from limited applicability. The main obstacle for automation is the complexity of concurrent data structures. This is particularly true in the absence of garbage collection. The intricacy of lock-free memory management paired with the complexity of concurrent data structures makes automated verification prohibitive.

In this work we present a method for verifying concurrent data structures and their memory management separately. We suggest two simpler verification tasks that imply the correctness of the data structure. The first task establishes an over-approximation of the reclamation behavior of the memory management. The second task exploits this over-approximation to verify the data structure without the need to consider the implementation of the memory management itself. To make the resulting verification tasks tractable for automated techniques, we establish a second result. We show that a verification tool needs to consider only executions where a single memory location is reused. We implemented our approach and were able to verify linearizability of Michael&Scott's queue and the DGLM queue for both hazard pointers and epoch-based reclamation. To the best of our knowledge, we are the first to verify such implementations fully automatically.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; **Program verification**; *Shared memory algorithms*; *Program specifications*; *Program analysis*;

Additional Key Words and Phrases: static analysis, lock-free data structures, verification, linearizability, safe memory reclamation, memory management

ACM Reference Format:
Roland Meyer and Sebastian Wolff. 2019. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 58 (January 2019), 31 pages. https://doi.org/10.1145/3290371

1 INTRODUCTION

Data structures are a basic building block of virtually any program. Efficient implementations are typically a part of a programming language's standard library. With the advent of highly concurrent computing being available even on commodity hardware, concurrent data structure implementations are needed. The class of lock-free data structures has been shown to be particularly efficient. Using fine-grained synchronization and avoiding such synchronization whenever possible results in unrivaled performance and scalability.

Unfortunately, this use of fine-grained synchronization is what makes lock-free data structures also unrivaled in terms of complexity. Indeed, bugs have been discovered in published lock-free

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Sebastian Wolff, TU Braunschweig, Germany, sebastian.wolff@tu-bs.de.

Contribution

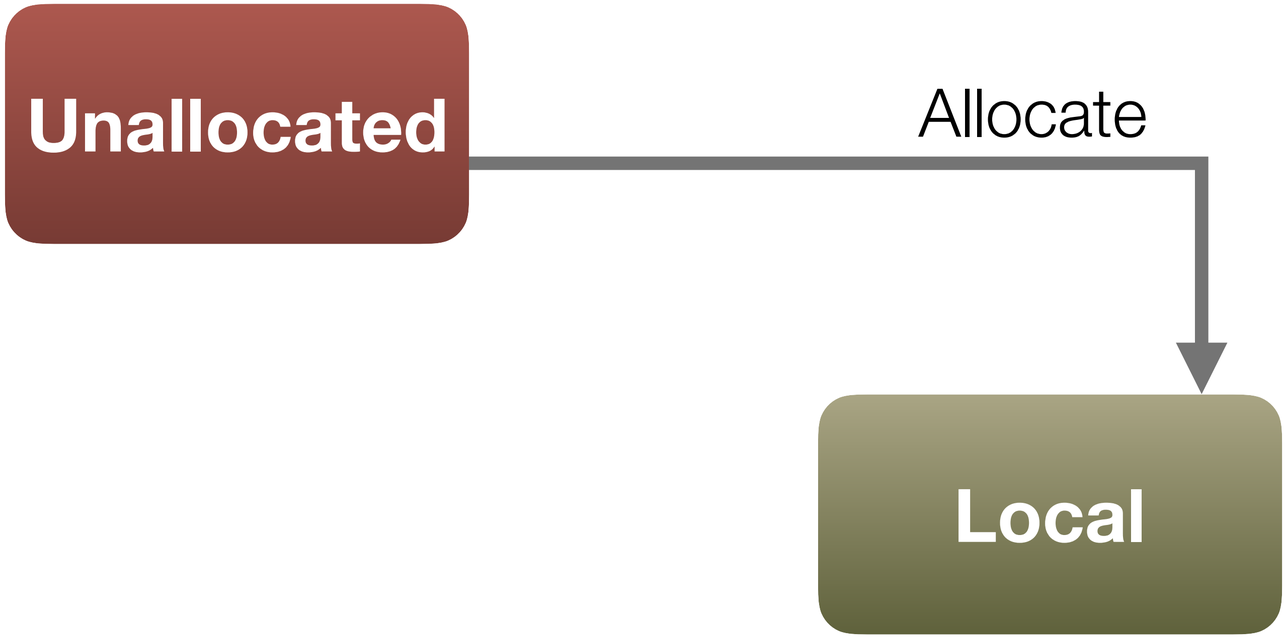
Type system:

- to establish: **Data structure** + **SMR spec** $\models$ **Pointer Race Freedom**
- that is parametrized by the **SMR spec**
- externalizes shape analysis to a **GC** tool

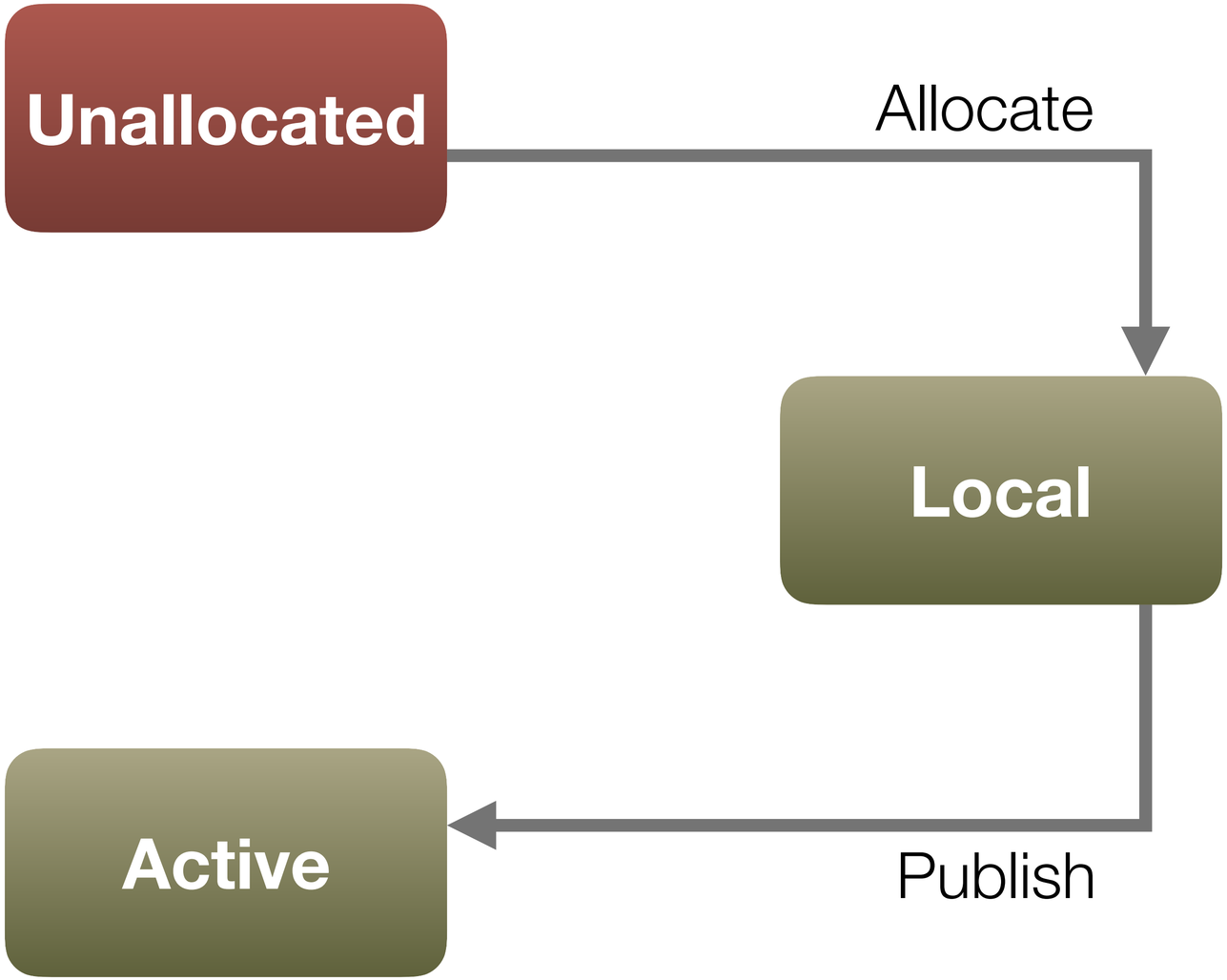
$\Longrightarrow$ **address "reclamation challenges"** 5/6 **separately** from 1/2/3/4

Memory Life Cycle

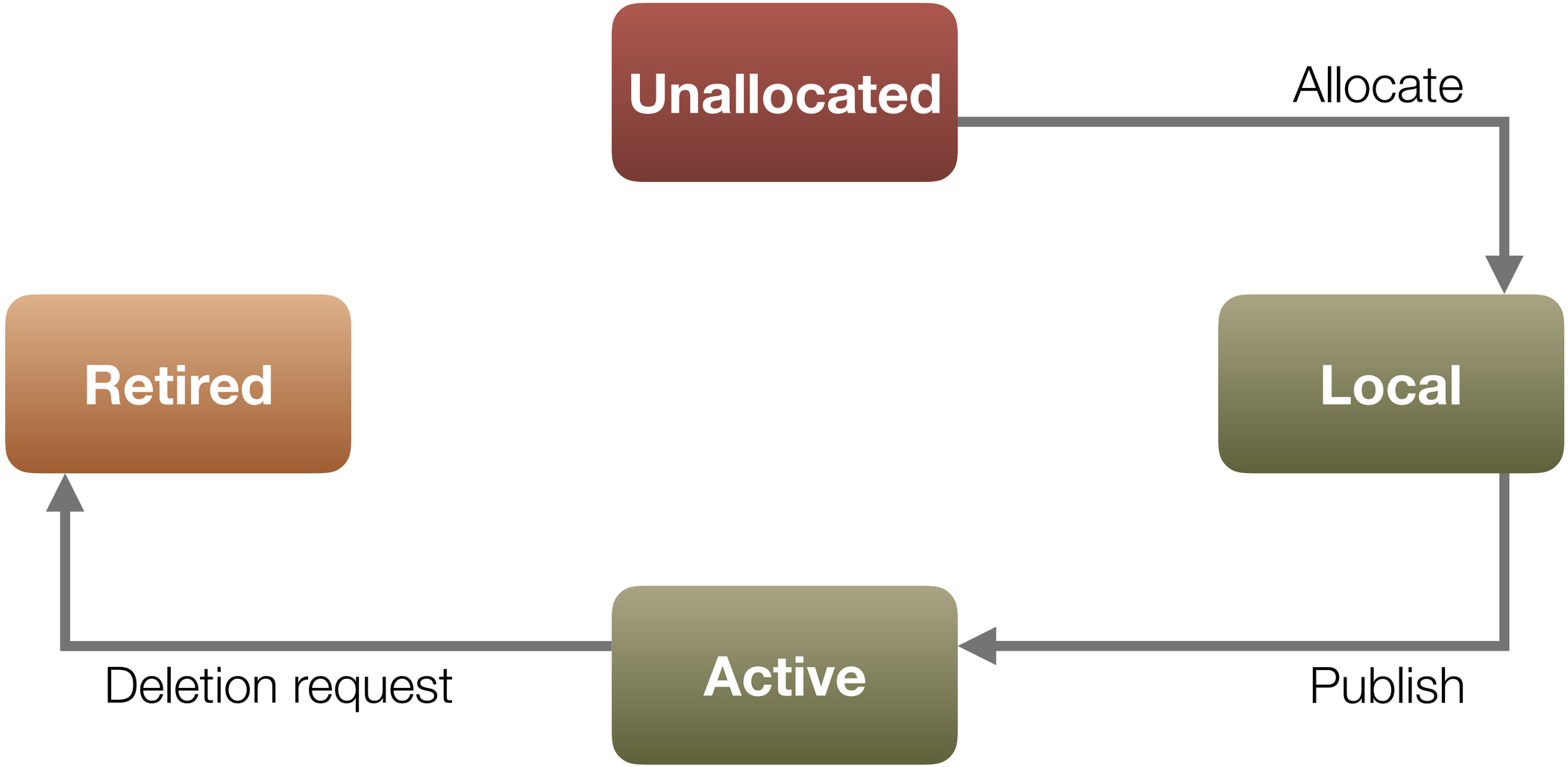
Memory Life Cycle



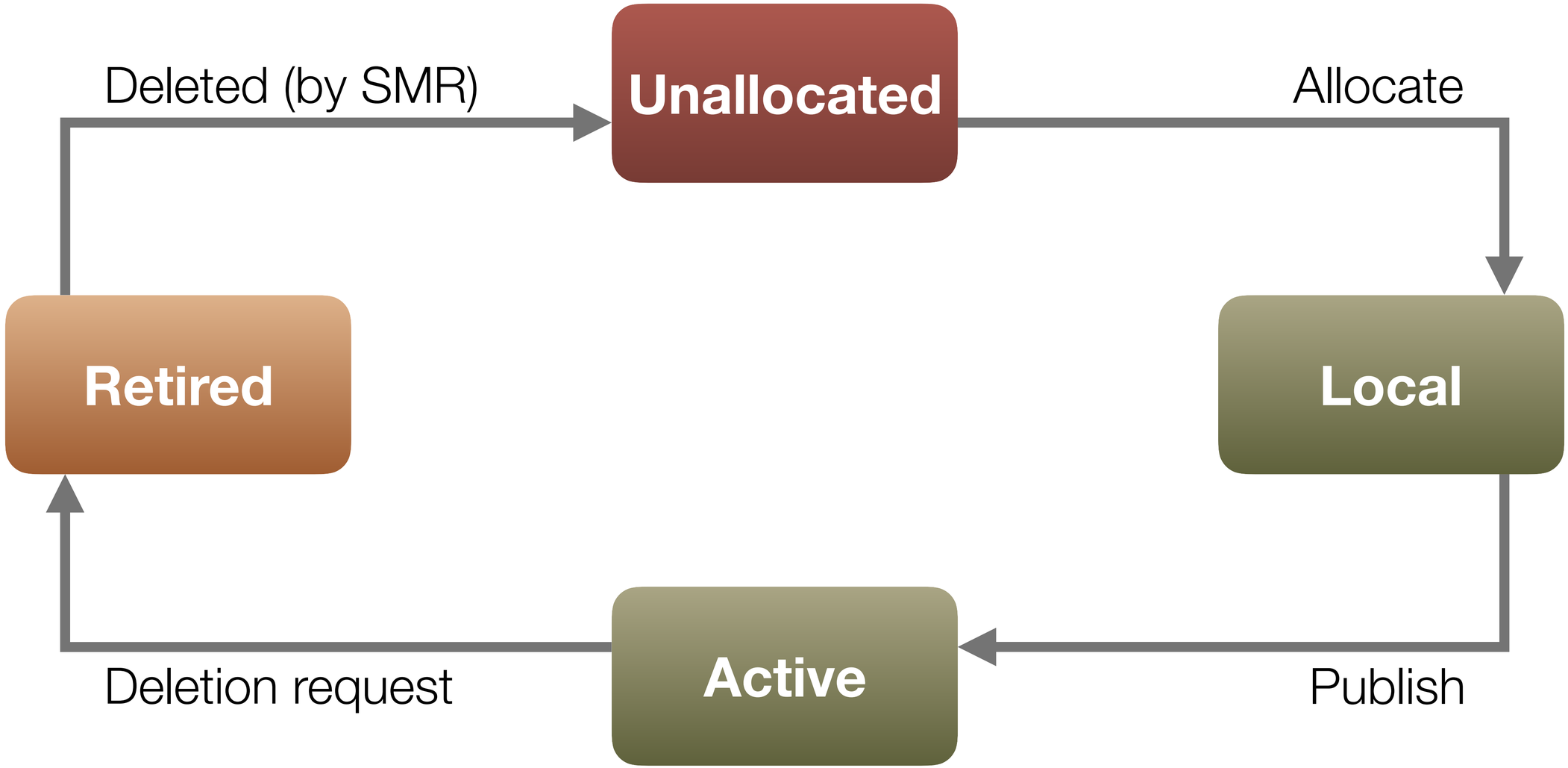
Memory Life Cycle



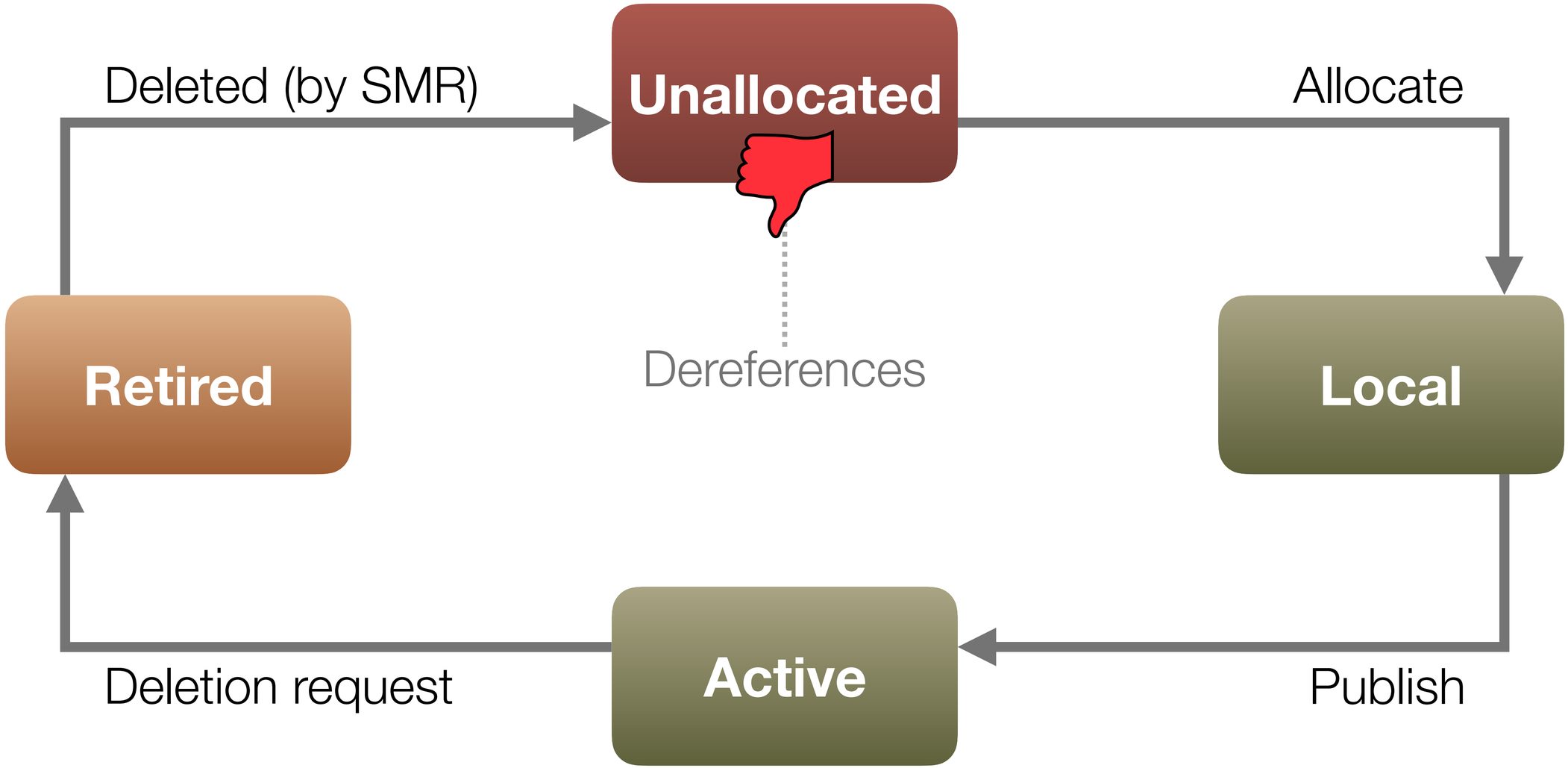
Memory Life Cycle



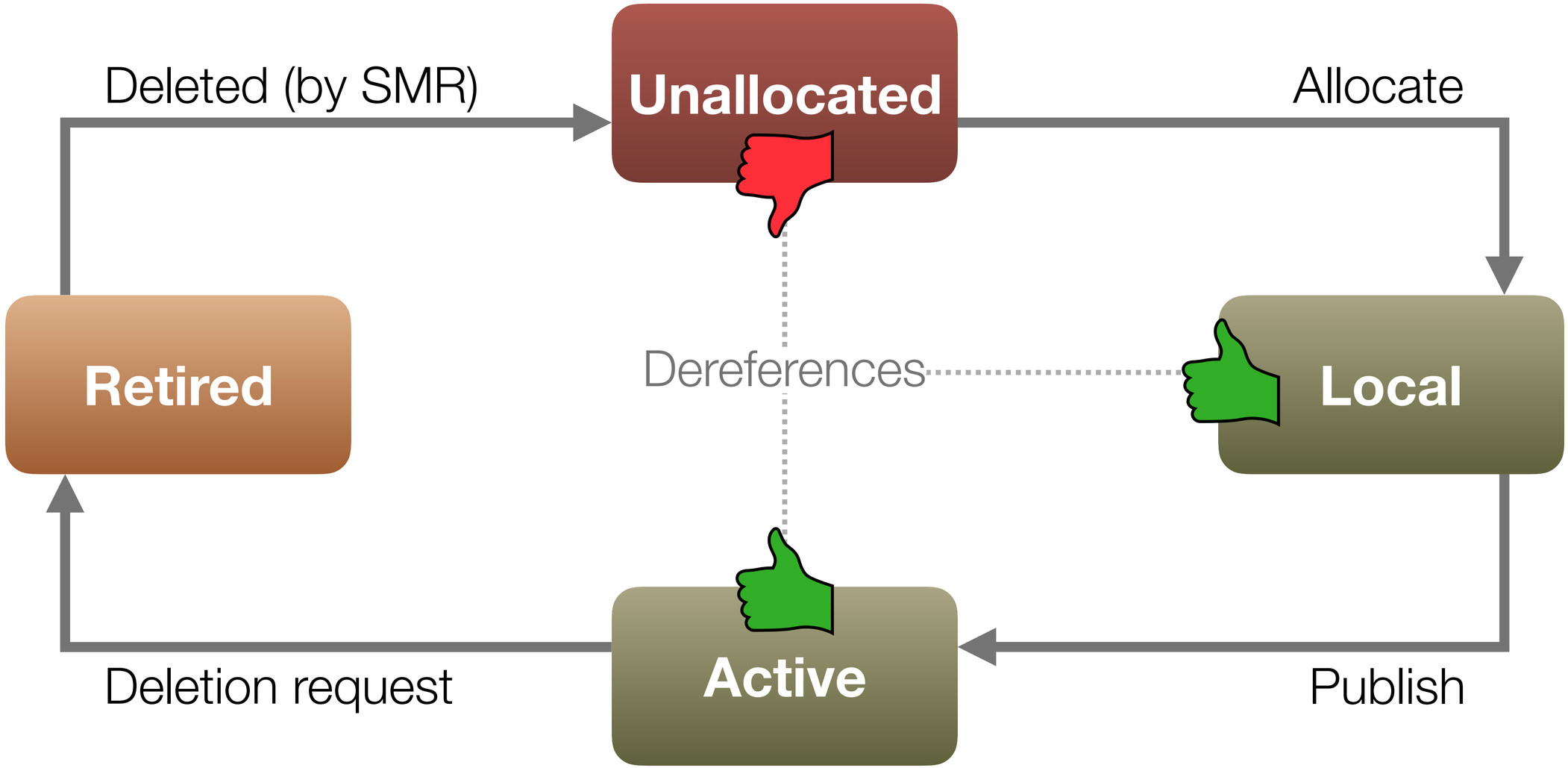
Memory Life Cycle



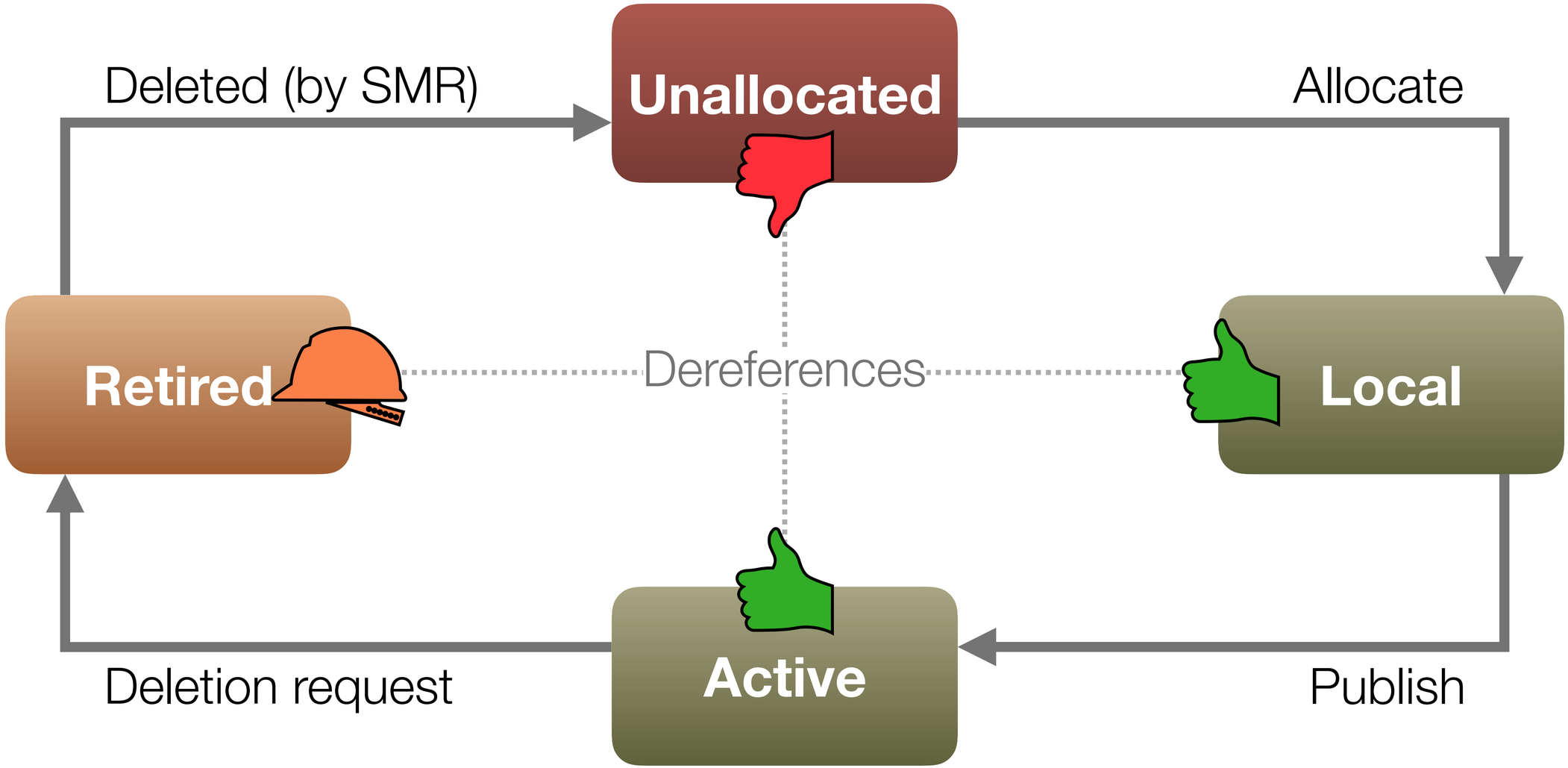
Memory Life Cycle



Memory Life Cycle



Memory Life Cycle



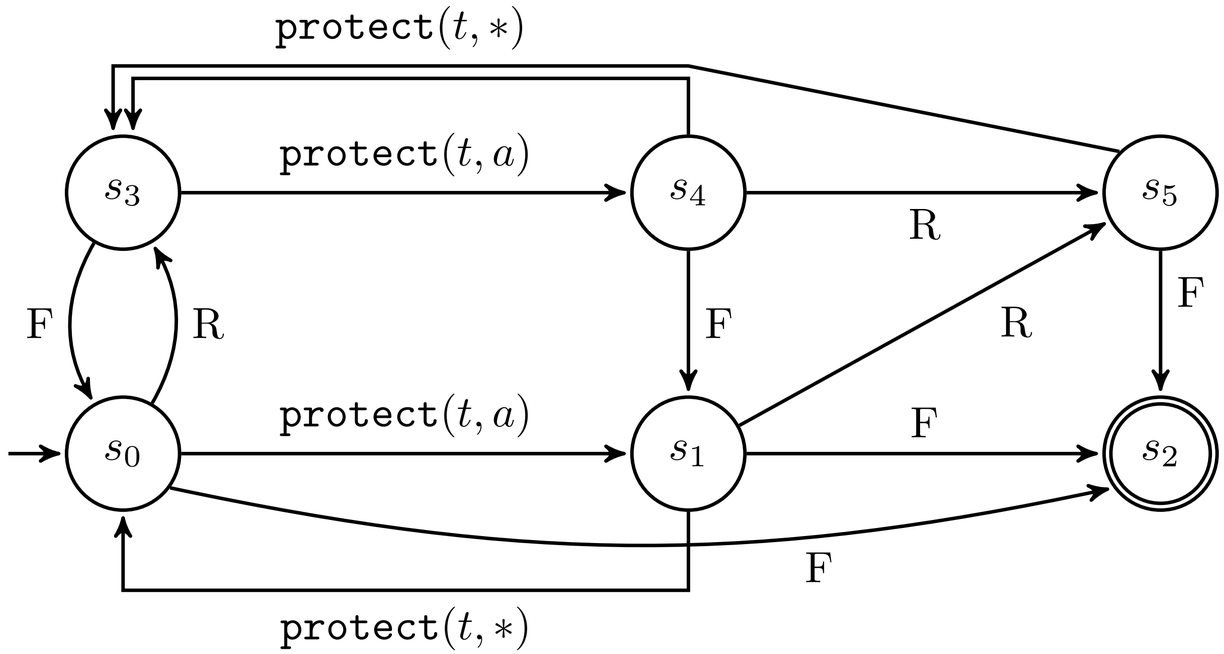
Implications for a Type System

- Idea:
  - $\mathbb{A}$: type to indicate activeness
  - $\mathbb{S}$: type to indicate that dereferences are safe
  - $\mathbb{P}$: type to track protections
- Challenges:
  - $\mathbb{A}$, $\mathbb{S}$, $\mathbb{P}$ need to be deducible **thread-locally**
  - $\mathbb{S}$, $\mathbb{P}$ need to be **interference-free**
  - $\mathbb{S}$, $\mathbb{P}$ **depend on the SMR**

Example

```
Node* head = Head;

protect(head);

atomic {

    @active Head

    assume(head == Head);
```

Type for **head**



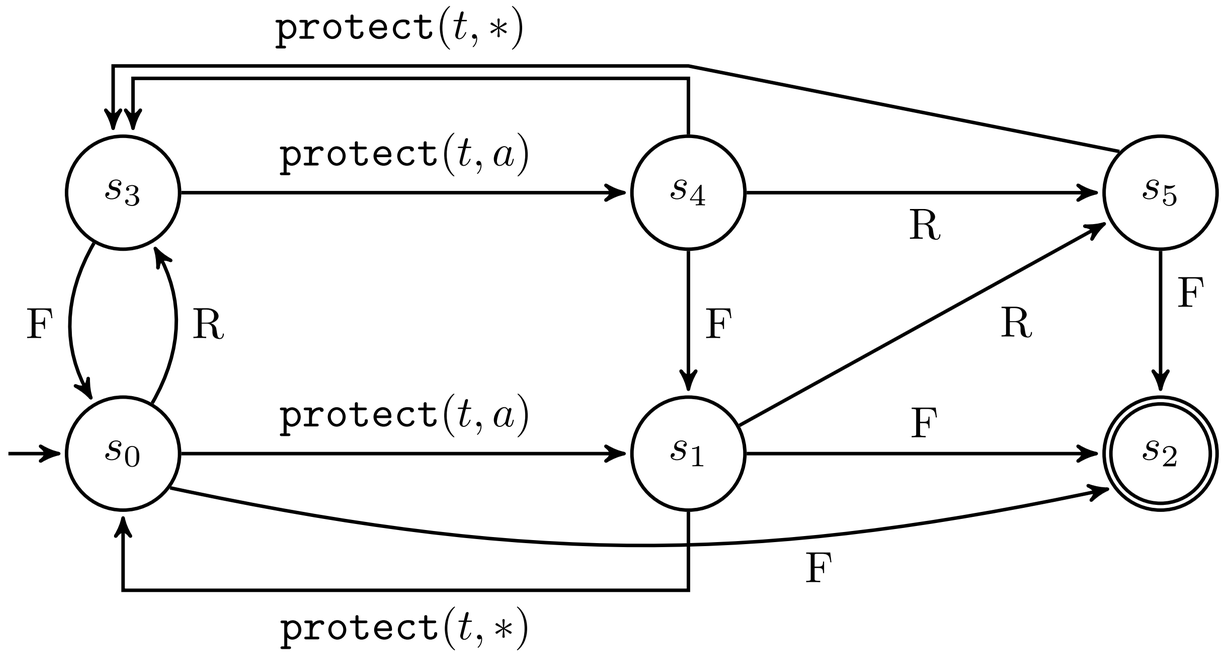
$$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$$

$$R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$$

Example

```
{ Head:∅ }
   Node* head = Head;

   protect(head);

   atomic {

      @active Head

      assume(head == Head);
```

Type for **head**



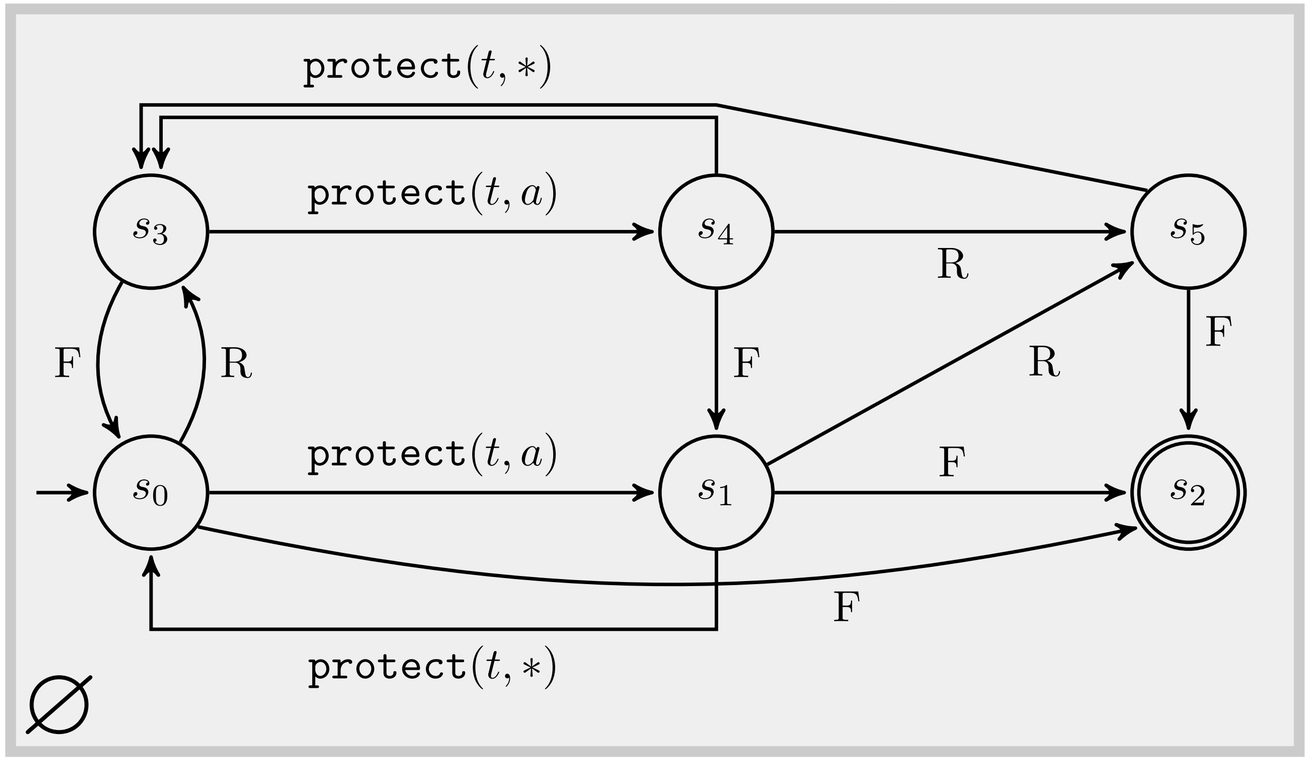
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
   Node* head = Head;

   protect(head);

   atomic {

      @active Head

      assume(head == Head);
```

Type for **head**



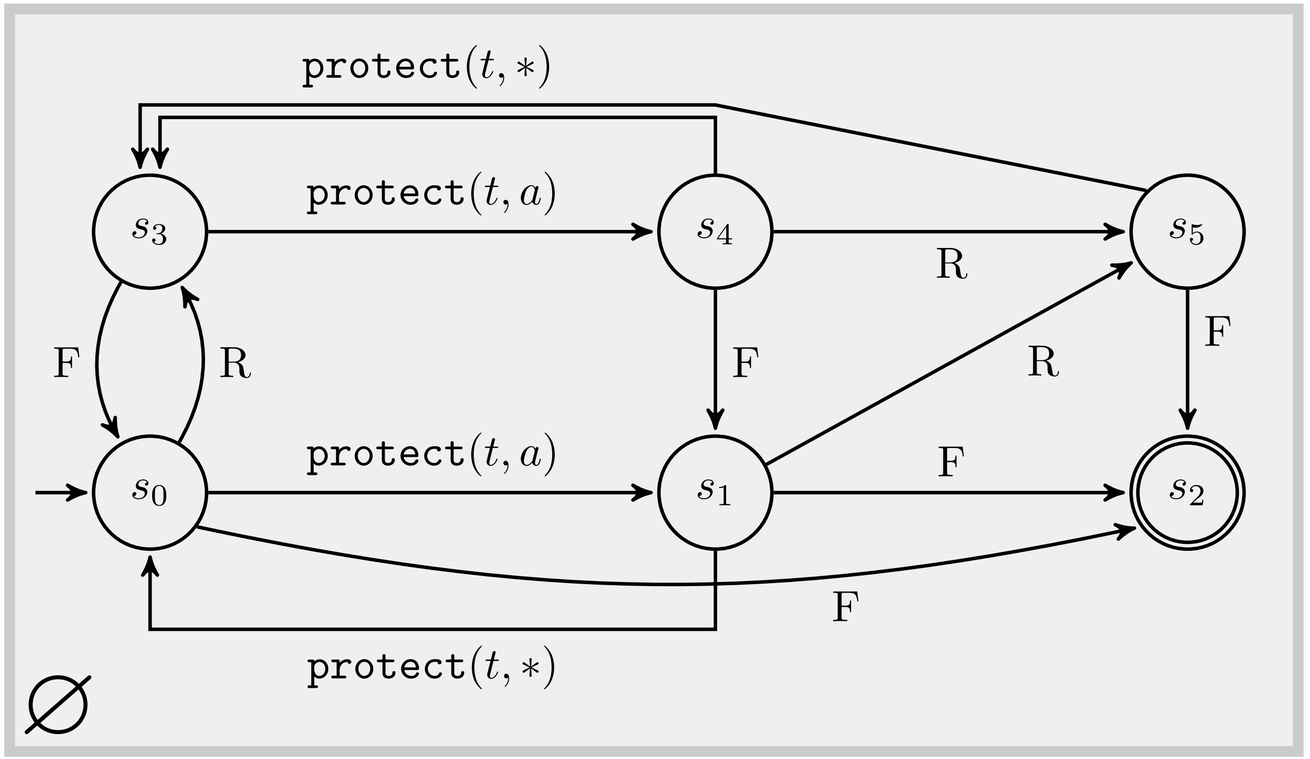
$$F \;:=\; \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R \;:=\; \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);


    atomic {


        @active Head


        assume(head == Head);
```

Type for **head**



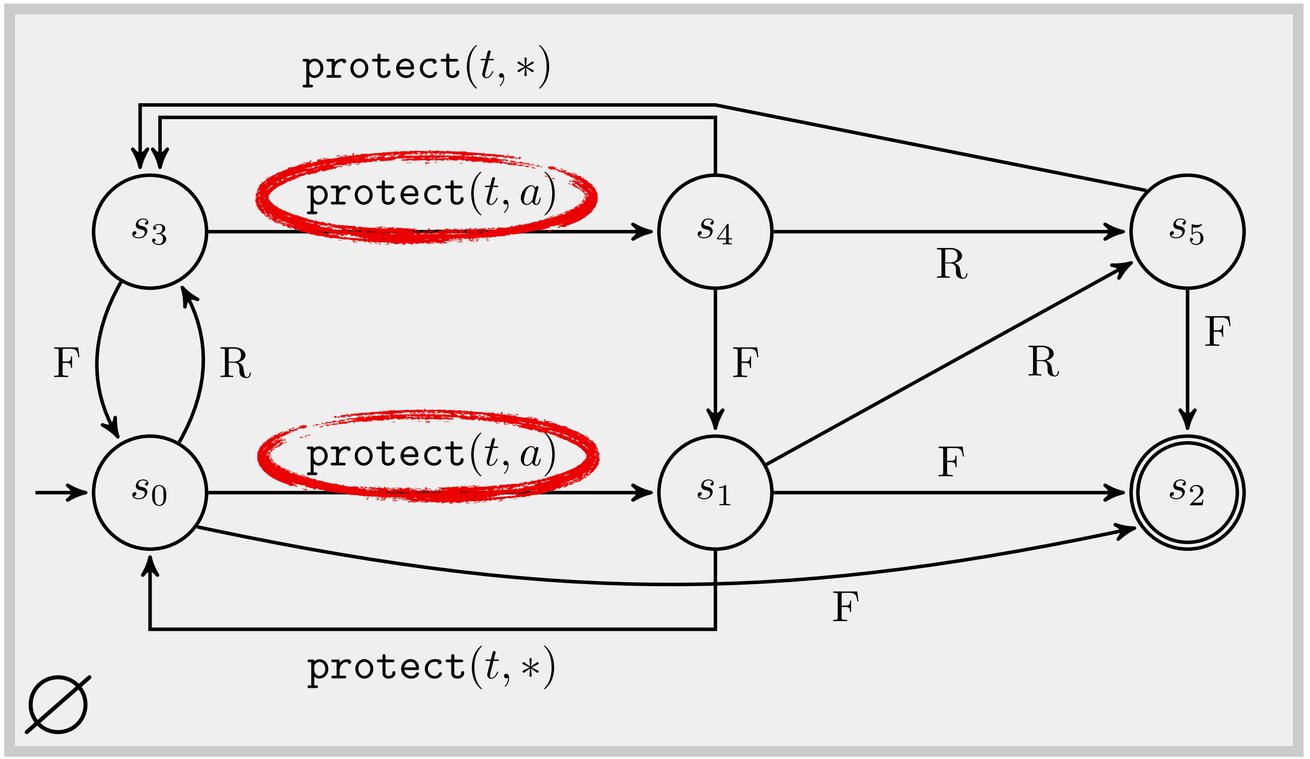
$$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$$

$$R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);


    atomic {


        @active Head


        assume(head == Head);
```

Type for **head**



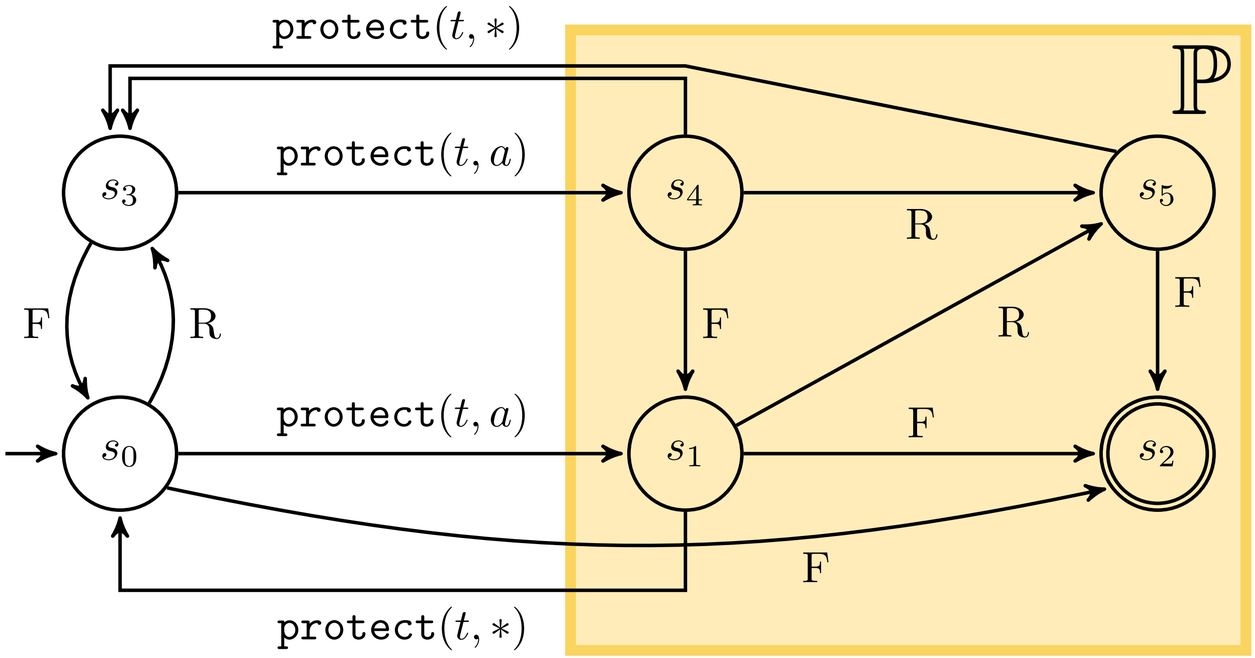
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);


    atomic {


        @active Head


        assume(head == Head);
```

Type for **head**



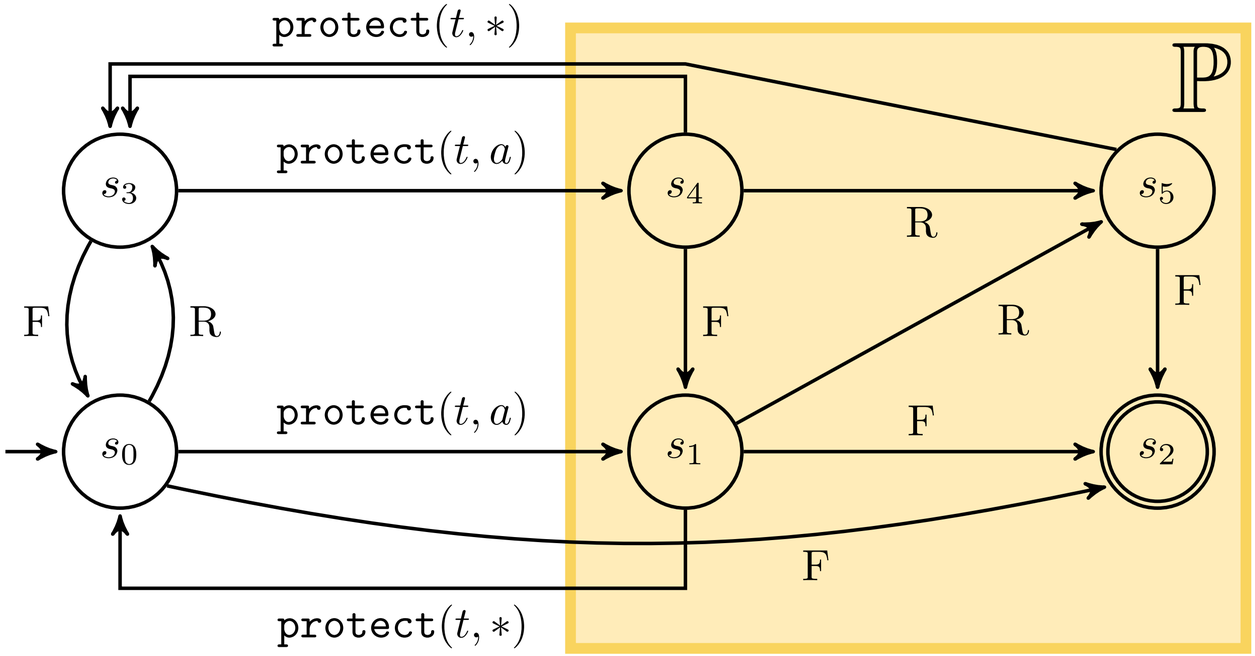
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);
{ Head:∅, head:ℙ }
    atomic {

        @active Head


        assume(head == Head);
```

Type for **head**



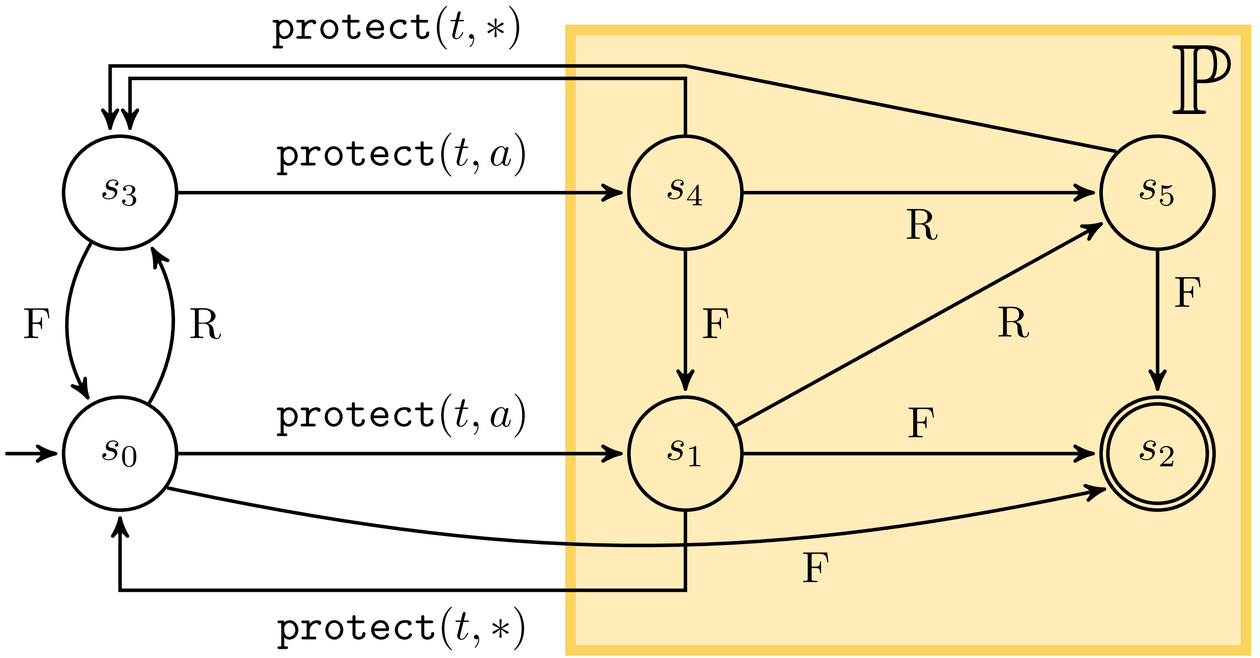
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);
{ Head:∅, head:ℙ }
    atomic {
{ Head:∅, head:ℙ }
        @active Head


        assume(head == Head);
```

Type for **head**



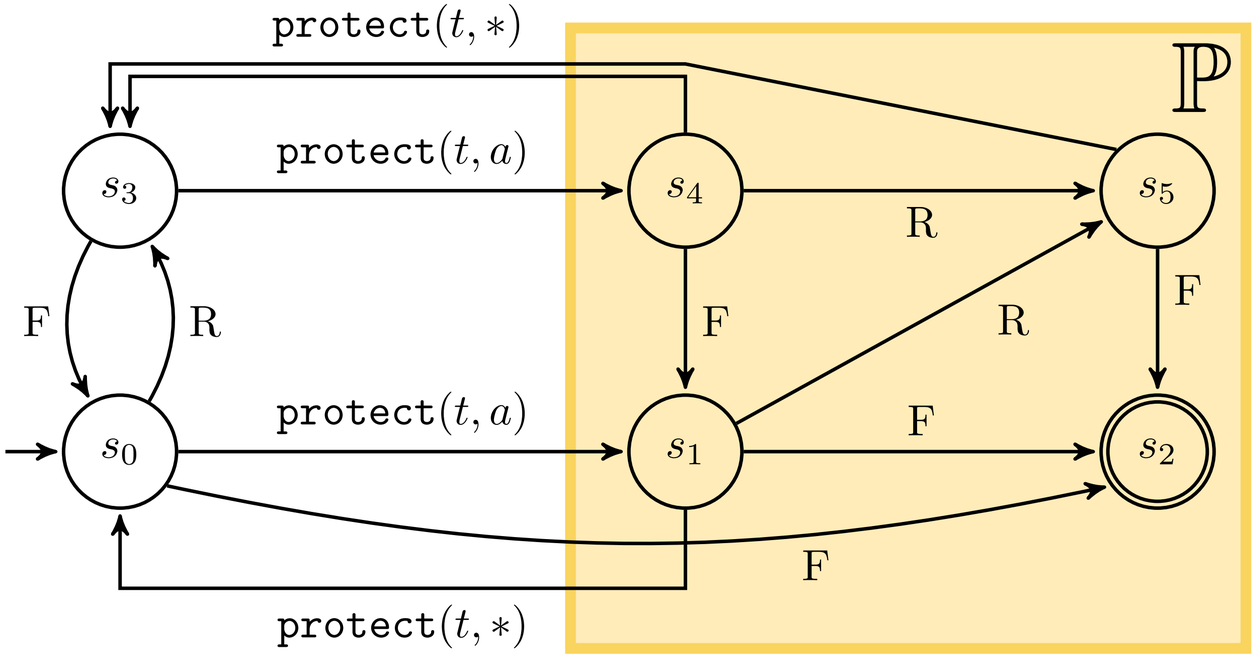
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:∅ }
    Node* head = Head;
{ Head:∅, head:∅ }
    protect(head);
{ Head:∅, head:ℙ }
    atomic {
{ Head:∅, head:ℙ }
        @active Head
{ Head:𝔸, head:ℙ }
        assume(head == Head);
```

Type for **head**



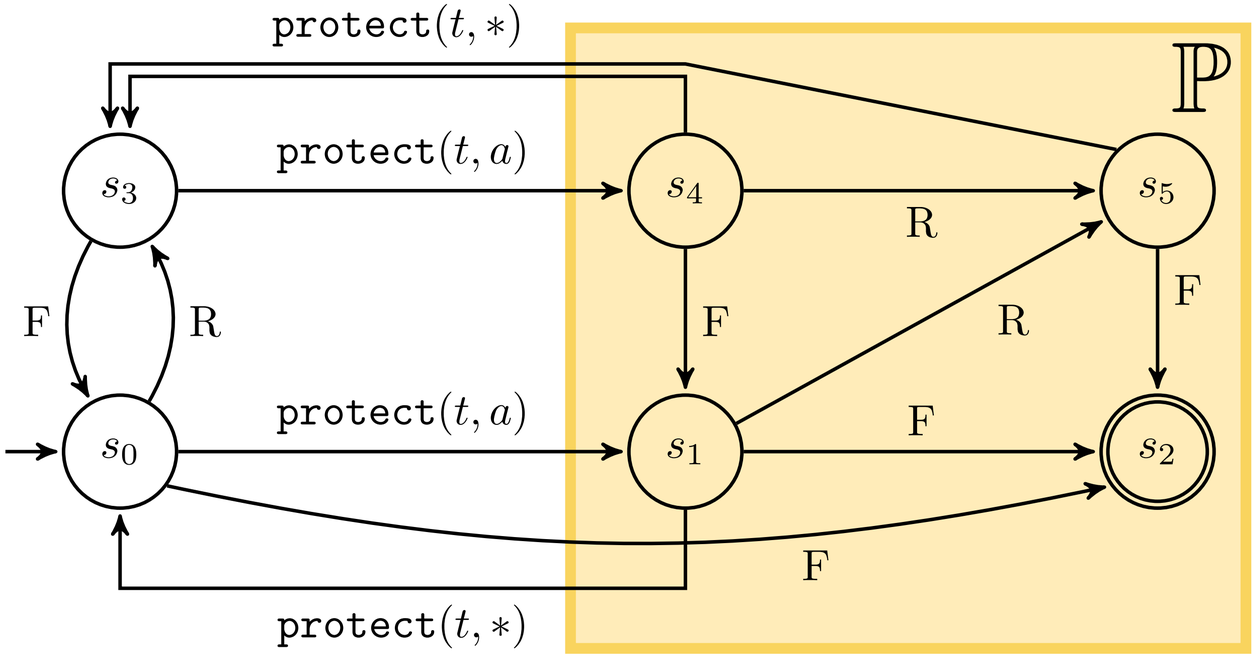
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);


  } // end atomic


  // ...
  Node* next = head->next;
```

Type for **head**



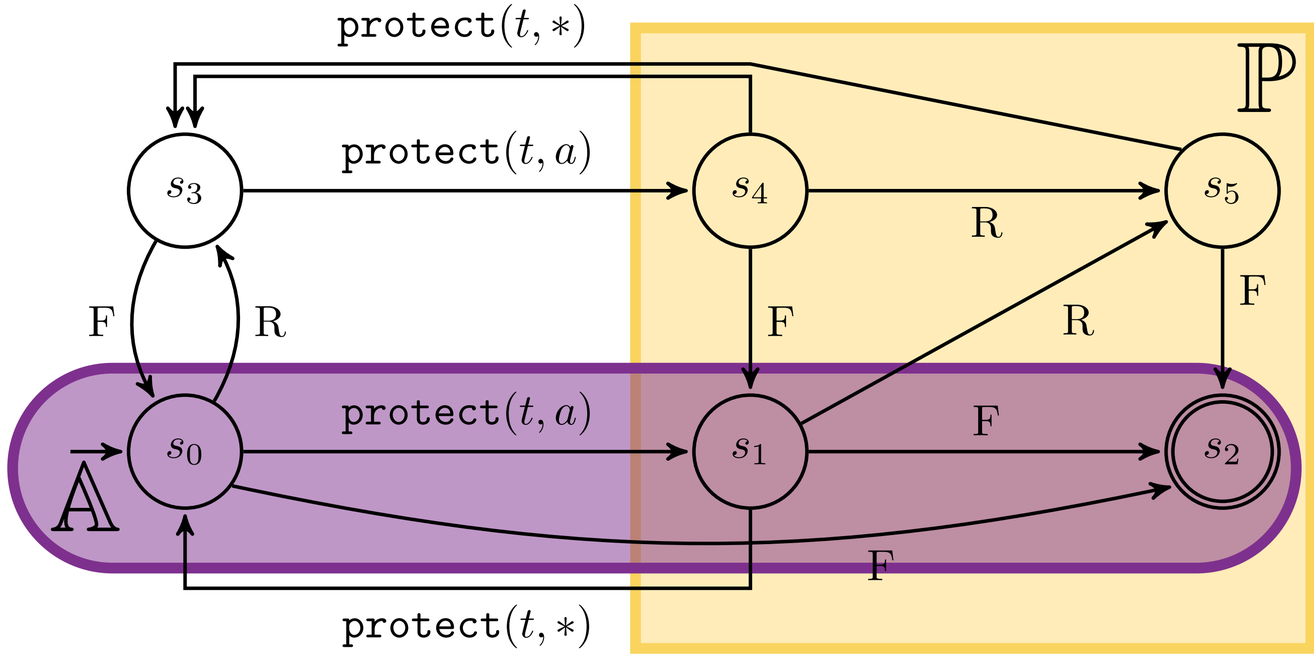
$$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$$

$$R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }


  } // end atomic


  // ...
  Node* next = head->next;
```

Type for **head**



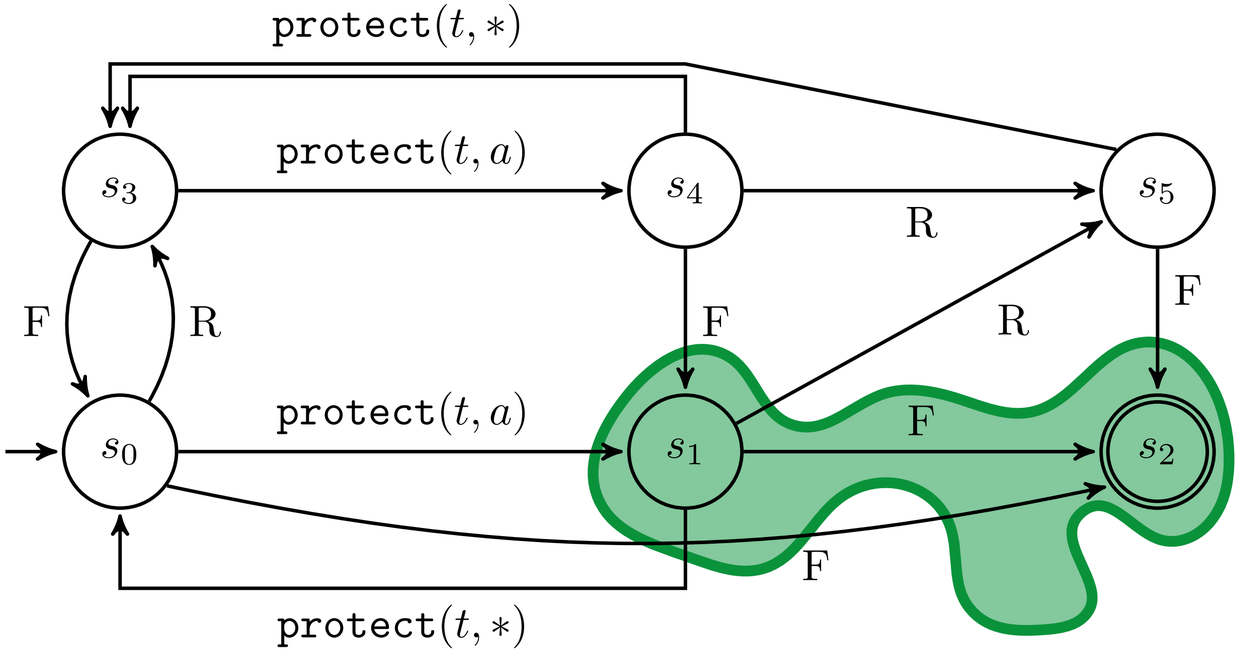
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }


  } // end atomic


  // ...
  Node* next = head->next;
```

Type for **head**



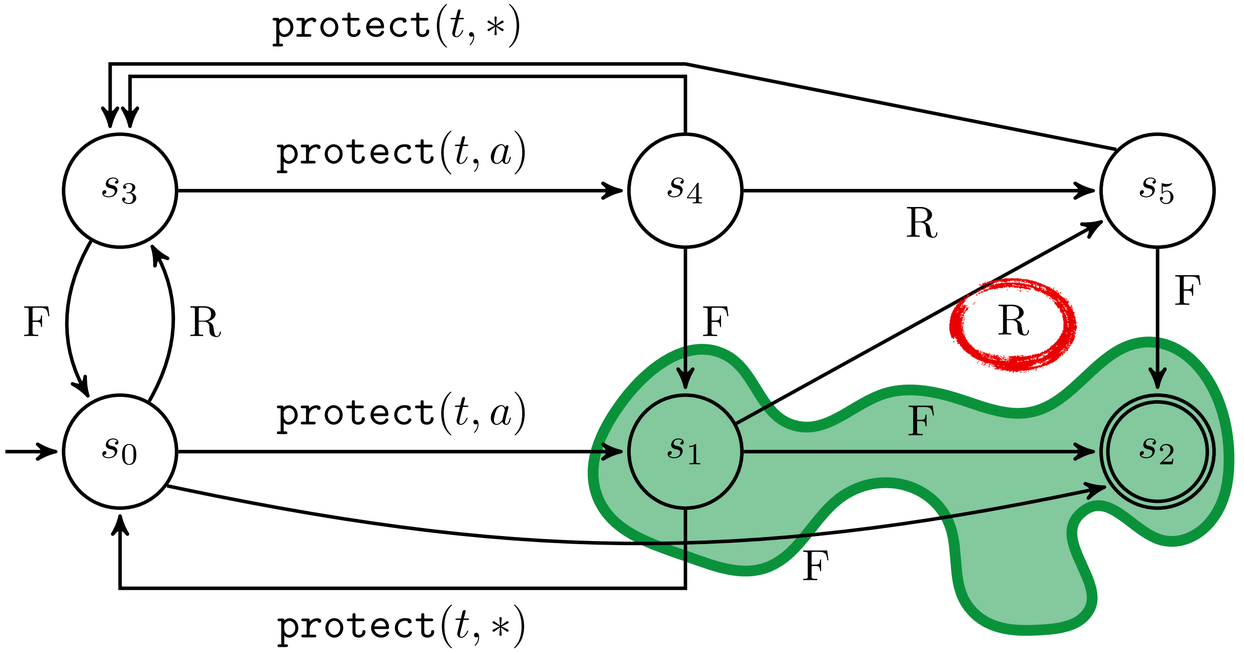
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }


  } // end atomic


  // ...
  Node* next = head->next;
```

Type for head



$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

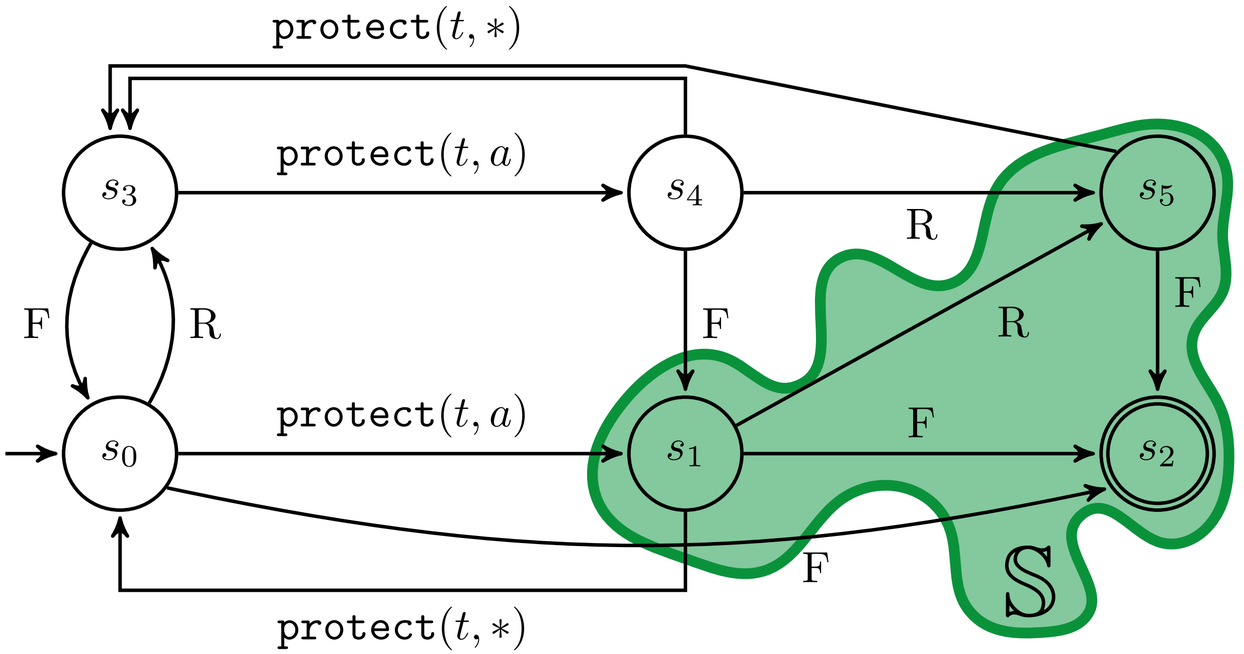
$\{ \texttt{Head}:\mathbb{A},\ \texttt{head}:\mathbb{P} \}$

```
        assume(head == Head);
```

$\{ \texttt{Head}:\mathbb{P} \wedge \mathbb{A},\ \texttt{head}:\mathbb{P} \wedge \mathbb{A} \}$

```
    } // end atomic


    // ...
    Node* next = head->next;
```

Type for **head**



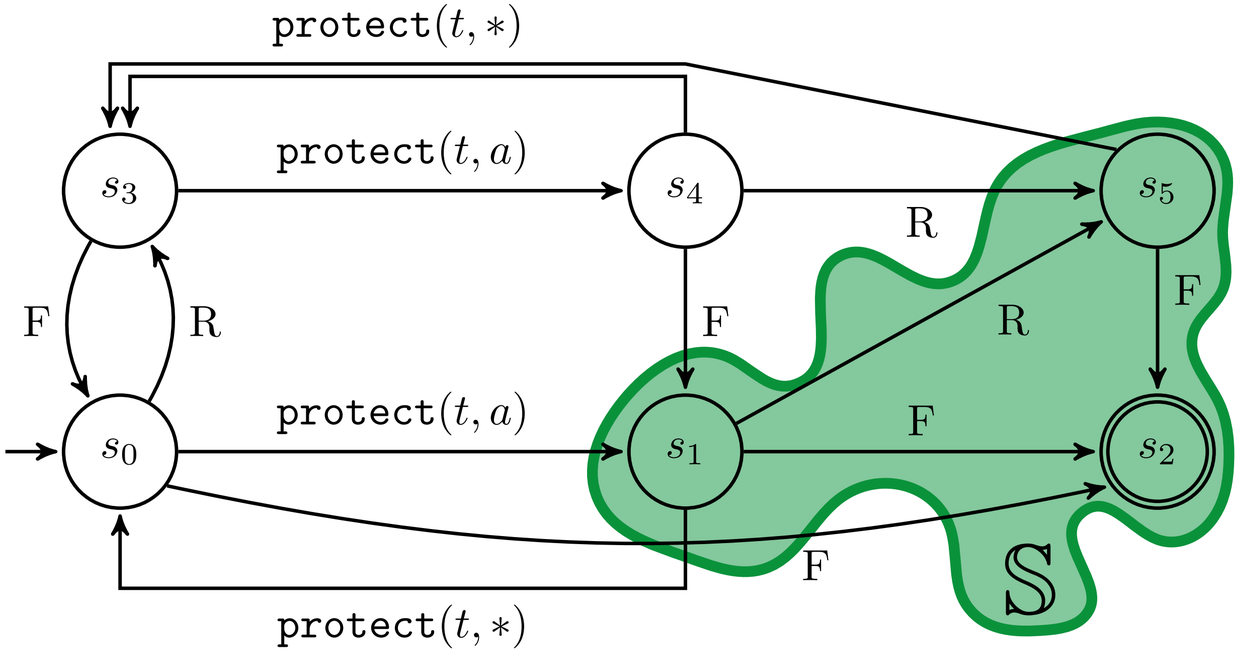
$$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$$

$$R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }
{ Head:𝕊, head:𝕊 }
   } // end atomic


   // ...
   Node* next = head->next;
```

Type for **head**



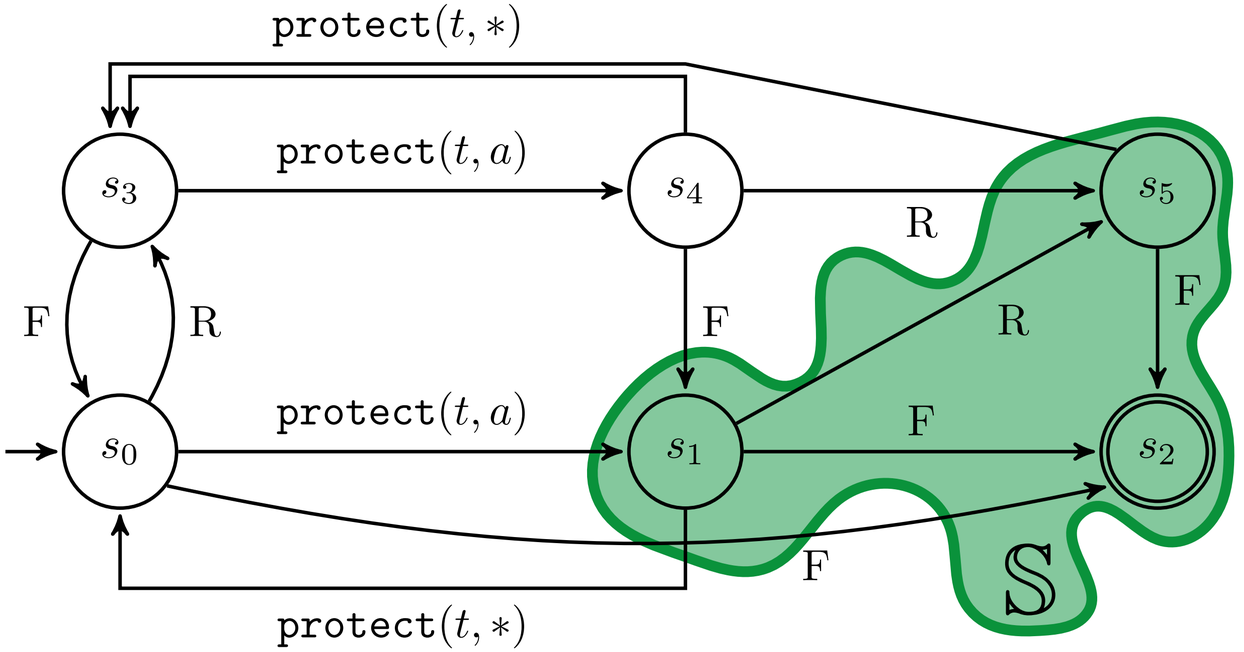
$$F := \texttt{free}(t, a) \vee \texttt{free}(*, a)$$

$$R := \texttt{retire}(t, a) \vee \texttt{retire}(*, a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }
{ Head:𝕊, head:𝕊 }
   } // end atomic
{ Head:∅, head:𝕊 }
   // ...
   Node* next = head->next;
```

Type for head



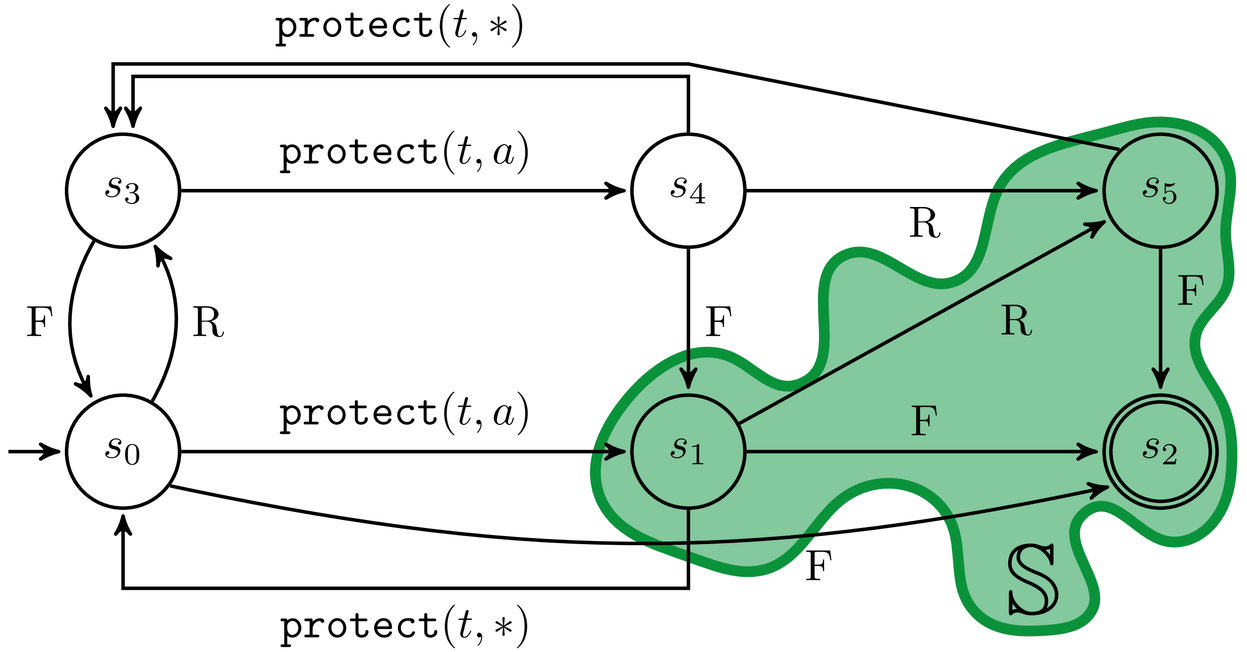
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }
{ Head:𝕊, head:𝕊 }
   } // end atomic
{ Head:∅, head:𝕊 }
   // ...
   Node* next = head->next;
```


Type for head



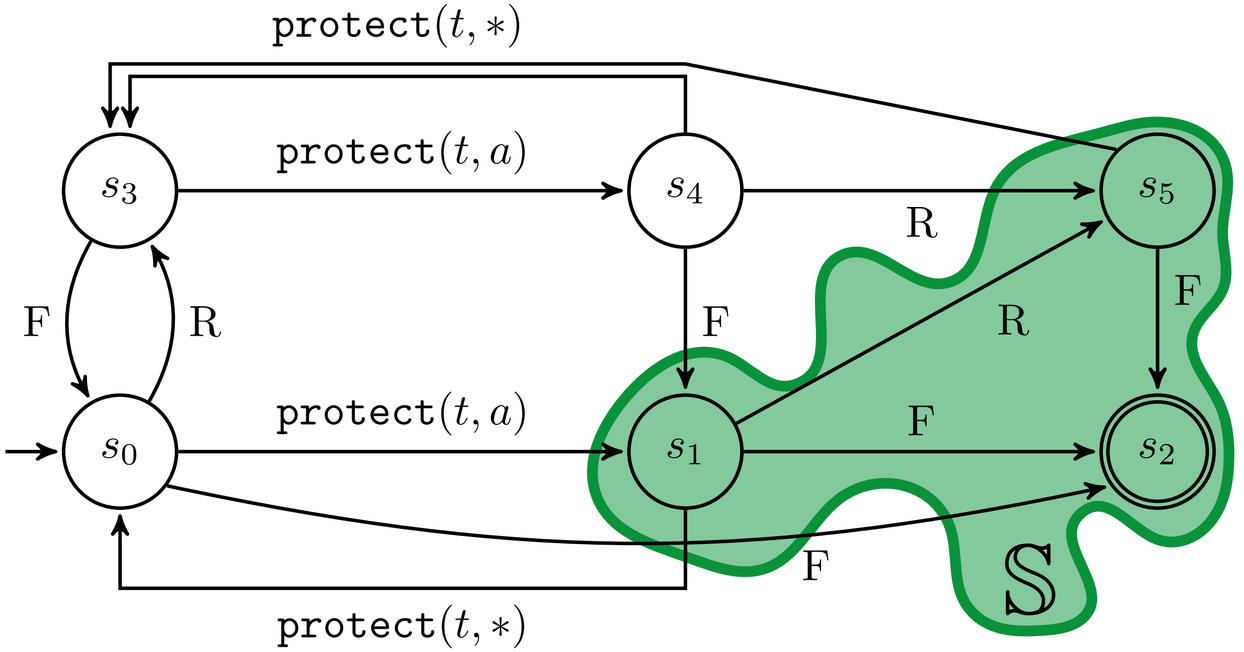
$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Example

```
{ Head:𝔸, head:ℙ }
      assume(head == Head);
{ Head:ℙ ∧ 𝔸, head:ℙ ∧ 𝔸 }
{ Head:𝕊, head:𝕊 }
   } // end atomic
{ Head:∅, head:𝕊 }
   // ...
   Node* next = head->next;
{ Head:∅, head:𝕊, next:∅ }
```


Type for head



$$F := \texttt{free}(t,a) \vee \texttt{free}(*,a)$$

$$R := \texttt{retire}(t,a) \vee \texttt{retire}(*,a)$$

Obtaining $\mathbb{A}$

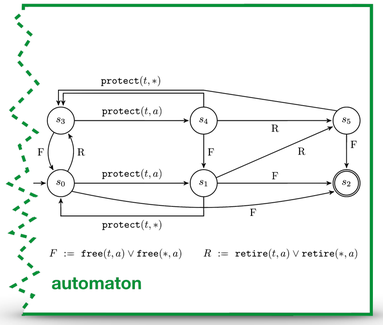
- Rely on `@active` annotations
- Discharge correctness of annotations automatically
  - ➡ instrumentation of DS code
  - ➡ can be done under GC!

$\Longrightarrow$ Guess-and-check approach for finding annotations

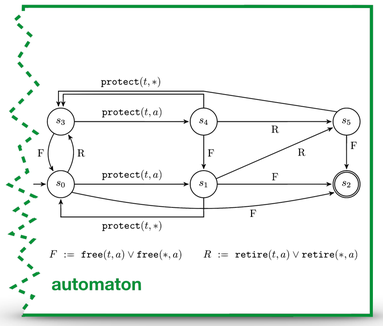
  - ➡ based on failed type check

Overall Approach

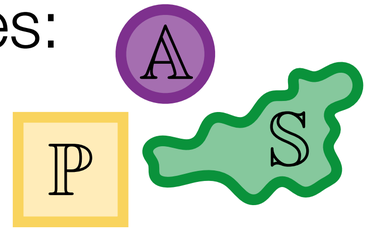
Overall Approach



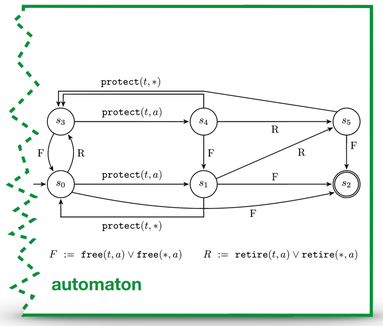
Overall Approach



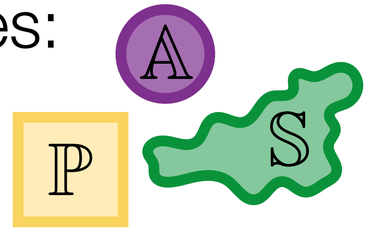
Types:



Overall Approach

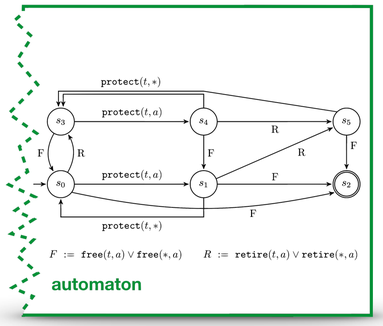


Types:

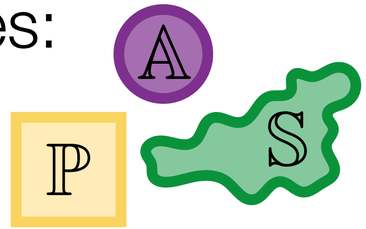


Type Check

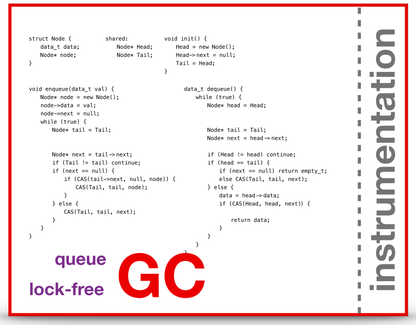
Overall Approach



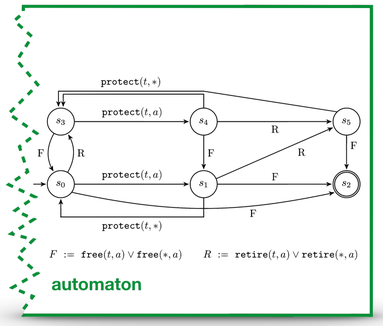
Types:



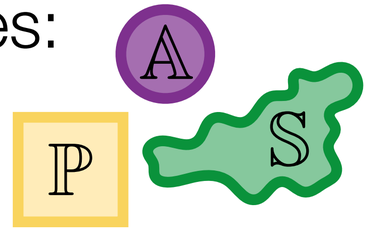
Type Check



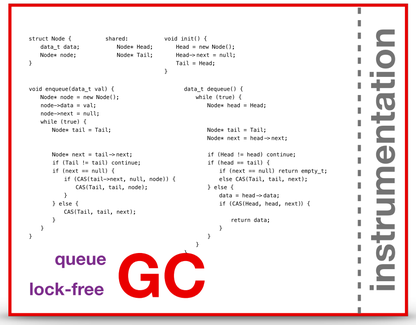
Overall Approach



Types:

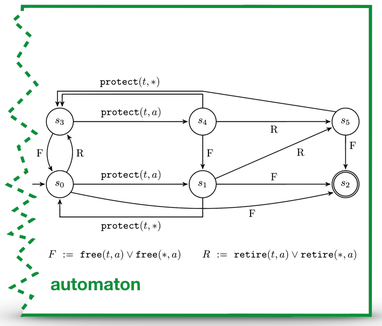


Type Check

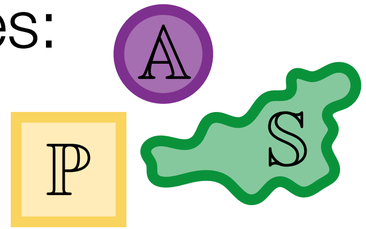


Annotation Check

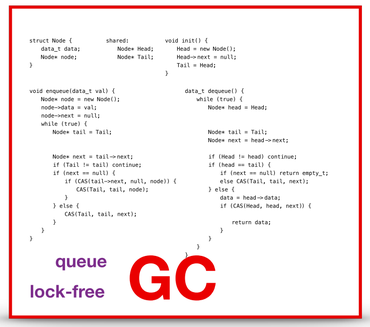
Overall Approach

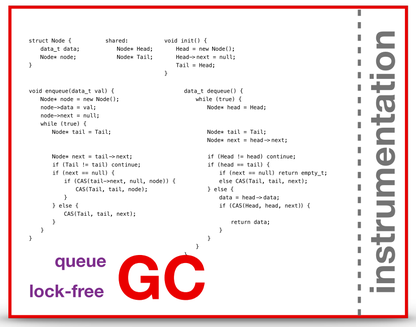


Types:



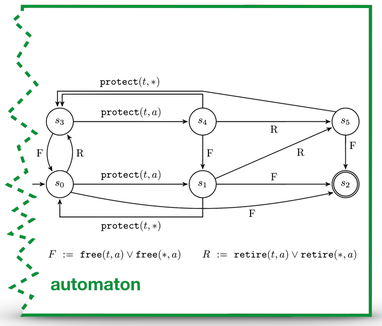
Type Check



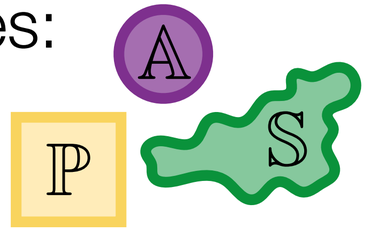


Annotation Check

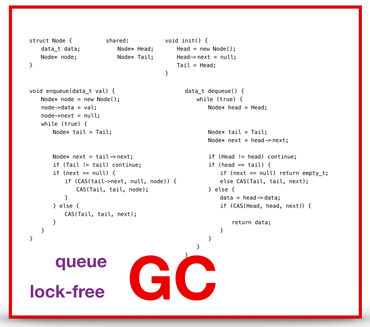
Overall Approach



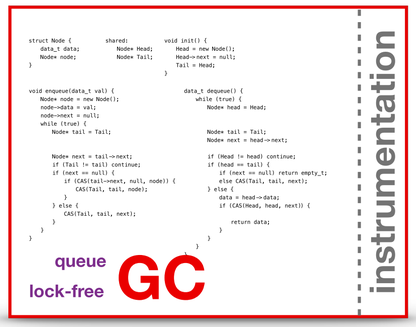
Types:



Type Check

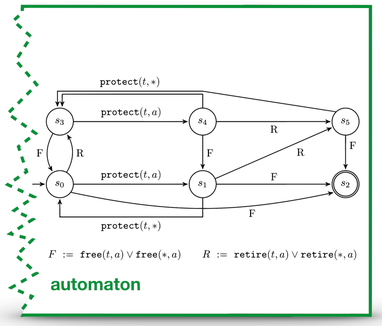


Verification

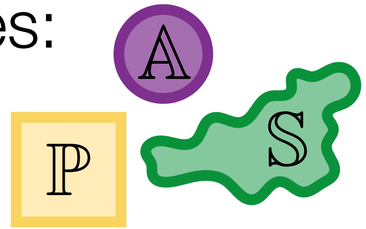


Annotation Check

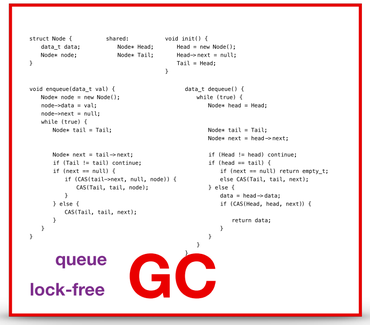
Overall Approach



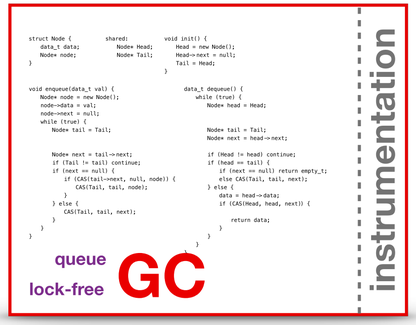
Types:



Type Check



Verification



Annotation Check

off-the-shelf verifier

Experiments

Data Structure	Types	Annot.	Verif.
Treiber's stack	0.7s ✓	12s ✓	1s ✓
Michael&Scott's queue	0.6s ✓	11s ✓	4s ✓
DGLM queue	0.6s ✓	1s ✗*	5s ✓
Vechev&Yahav's DCAS set	1.2s ✓	13s ✓	98s ✓
Vechev&Yahav's CAS set	1.2s ✓	3.5h ✓	42m ✓
ORVYY set	1.2s ✓	3.2h ✓	47m ✓
Michael's set	1.2s ✓	90s ✗*	t/o

* imprecision in the back-end verifier

Fin.