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

Roland Meyer and Sebastian Wolff
TU Braunschweig, Germany
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if(CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if(CAS(Head, head, next)){
            // leak head?
            return;
        }
    }
}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

```c
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
```

---

**Diagram:**

- **Head**
  - `head1` (previous head)
  - `head2` (current head)
  - `headx` (new head)

- **Elements:**
  - `a`
  - `b`
  - `c`

- **Connections:**
  - `head1` to `head2`
  - `head2` to `headx`
  - `next1` to `next2`
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

```c
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if(CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
```

Diagram:
- Head
- a
- b
- c
- head₁
- head₂
- next₁
- next₂
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            // leak head?
            return;
        }
    }
}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            delete head;
            return;
        }
    }
}
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            delete head;
            return;
        }
    }
}
Lock-free Queue (Michael&Scott)

```cpp
void dequeue() {
    while (true) {
        head = Head;
        next = head->next;
        // ... 
        if(CAS(Head, head, next)){
            delete head;
            return;
        }
    }
}
```
Reclamation

- Lock-free data structures (LFDS)
  - unsynchronized traversal
  - threads cannot detect whether a dereference is *safe*

- Safe memory reclamation (SMR)
  - defers deletion until it is safe
  - controlled by LFDS
  - various sophisticated techniques exist
Lock-free Queue (Michael&Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        // protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
```c
data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
```
Lock-free Queue (Michael&Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...  
        if(CAS(Head, head, next)){
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael&Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if(CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael & Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head -> next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
Lock-free Queue (Michael&Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        } 
    }
}
Lock-free Queue (Michael&Scott)

data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ...
        if(CAS(Head, head, next)){
            retire(head);
            return;
        }
    }
}
data_t dequeue() {
    while (true) {
        head = Head;
        protect(head);
        next = head->next;
        // ... 
        if (CAS(Head, head, next)) {
            retire(head);
            return;
        }
    }
}
State-of-the-art Verification of Data Structures

- Pen&paper, mechanized/tool-supported
  - require deep understanding of proof technique, LFDS, and SMR
  - few works consider reclamation

- Automated (model-checking)
  - only done for GC
  - or custom semantics (allowing accesses of deleted memory)
  - no works consider SMR
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 {
            data = head->data;
            if (CAS(Head, head, next)) {
                return data;
            }
        }
    }
}

46 LOC
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;
    protect(tail, 0);
    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);
        break;
      }
    } else {
      CAS(Tail, tail, next);
    }
  }
}

data_t dequeue() {
  while (true) {
    Node* head = Head;
    protect(head, 0);
    if (Head != head) continue;
    Node* tail = Tail;
    Node* next = head->next;
    protect(next, 1);
    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;
      }
    }
  }
}
struct Node {
    data_t data;
    Node* next;
    Node* tail;
} node

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

data_t enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        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;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        if (next == null) {
            if (CAS(head->next, null, tail)) {
                return empty_t;
            }
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}

void protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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

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

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

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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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

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

46+6 LOC +52 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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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;
        }
    }
}

++52 LOC

Verification LFDS+SMR

It is a second lock-free data structure!
Verification LFDS+SMR

It is a second lock-free data structure!
Verification LFDS+SMR

struct Node {
    data_t data;
    Node* next;
    Node* hp0;
    Node* hp1;
}

void init() {
    Node* node = new Node();
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                return;
            }
        } else {
            protect(next, 1);
            if (Tail != tail) continue;
            Node* head = Tail;
            protect(head, 0);
            if (Head != head) continue;
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        }
    }
}

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                return;
            }
        } else {
            protect(next, 1);
            if (Tail != tail) continue;
            Node* head = Tail;
            protect(head, 0);
            if (Head != head) continue;
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        }
    }
}

void dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        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;
            }
            retire(head);
        }
    }
}

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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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

It is a second lock-free data structure!
Verifying LFDS+SMR

**LFDS**

```c
struct Node {
  int data;
  Node* next;
};

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

void enqueue(int val) {
  Node* node = new Node();
  node->data = val;
  node->next = null;
  while (true) {
    Node* tail = Tail;
    protect(tail, 0);
    if (Tail != tail) continue;
    Node* next = tail->next;
    if (next == null) {
      if (CAS(tail->next, null, node)) {
        CAS(Tail, tail, node);
      } else {
        return;
      }
    } else {
      CAS(tail, tail, next);
    }
  }
}
```

**SMR**

```c
struct Node {
  int data;
  Node* next;
};

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

int dequeue() {
  while (true) {
    Node* head = Head;
    protect(head, 0);
    if (Head != head) continue;
    Node* tail = Tail;
    Node* next = head->next;
    if (next == null) {
      if (CAS(head->next, null, head)) {
        return empty;
      } else {
        data = head->data;
        if (CAS(head, head, next)) {
          return data;
        }
      }
    } else {
      CAS(head, head, next);
    }
  }
}
```

It is a second lock-free data structure!
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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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;
        }
    }
}

46+6 LOC

Verification LFDS+SMR

It is a second lock-free data structure!

API calls

SMR

Allocator

free

free

Allocator
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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

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;
        }
    }
}

+52 LOC
Verification LFDS+SMR

46+6 LOC

46+6 LOC

It is a second lock-free data structure!
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 protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (!protectedList.contains(ptr)) {
        retiredList.remove(ptr);
        delete ptr;
    }
}

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;
        }
    }
}

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        protect(next, 1);
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
                return;
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        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;
            }
        }
    }
}

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

void free(Node* ptr) {
    free(ptr);
}
Contribution 1: Compositional Verification for LFDS + SMR
Compositional Verification

- API between LFDS and SMR
  - give a formal specification SPEC
      ➡ SPEC states \textit{which\&when} addresses are freed

- \textbf{Compositional Verification}
  1) verify SMR against SPEC
  2) verify LFDS, using SPEC to over-approximate SMR
SPEC Example

- Hazard pointers:
  
  a retired node is not reclaimed if it has been protected continuously since before the retire

- Programmers rely on this guarantee, not on the actual implementation

- Formalized:

  $O_{HP}(t, a)$

  ![Diagram]

  - $s_1$: invocation $\text{protect}(t, a)$
  - $s_2$: response $\text{protect}(t, a)$
  - $s_3$: invocation $\text{retire}(*, a)$
  - $s_4$: invocation $\text{free}(*, a)$
  - $s_5$: invocation $\text{unprotect}(t)$
### Experiments

- **SMR against SPEC:**

| SMR implementation                  | SPEC size | Time   | Correct? |
|-------------------------------------|-----------|--------|----------|
| Hazard Pointers (HP)                | 3x5x5     | 1.5s   | yes      |
| Epoch-based Reclamation (EBR)       | 3x5       | 11.2s  | yes      |
Experiments

- SMR against SPEC:

| SMR implementation       | SPEC size | Time  | Correct? |
|--------------------------|-----------|-------|----------|
| Hazard Pointers (HP)     | 3x5x5     | 1.5s  | yes      |
| Epoch-based Reclamation (EBR) | 3x5      | 11.2s | yes      |

- Linearizability of LFDS+SPEC

Infeasible: severe state space explosion due to re-allocations!
Contribution 2: State Space Reduction
State Space Reduction

• Theorem:

  For verification, it is sound to restrict re-allocations to a single address

• Two requirements:
  1) SPEC invariant to re-allocations
  2) LFDS free from ABAs
State Space Reduction

• Theorem:

   For verification, it is sound to restrict re-allocations to a single address

• Two requirements:

   1) SPEC invariant to re-allocations  ➞  check on SPEC automaton

   2) LFDS free from ABAs  ➞  check on reduced (!) LFDS state space
## Experiments cont.

| LFDS                  | SPEC     | Time | Linearizable? |
|-----------------------|----------|------|---------------|
| Michael&Scott's queue | NoReclaim | 7m   | yes           |
| Michael&Scott's queue | EBR      | 44m  | yes           |
| Michael&Scott's queue | HP       | 120m | yes           |
| Treiber's stack       | EBR      | 16s  | yes           |
| Treiber's stack       | HP       | 19s  | yes           |
| DGLM queue            | EBR      | 63m  | yes*          |
| DGLM queue            | HP       | 117m | yes*          |

* with hint for heap abstraction
Fin.

Questions?