Verifying Non-blocking Data Structures with Manual Memory Management

Sebastian Wolff

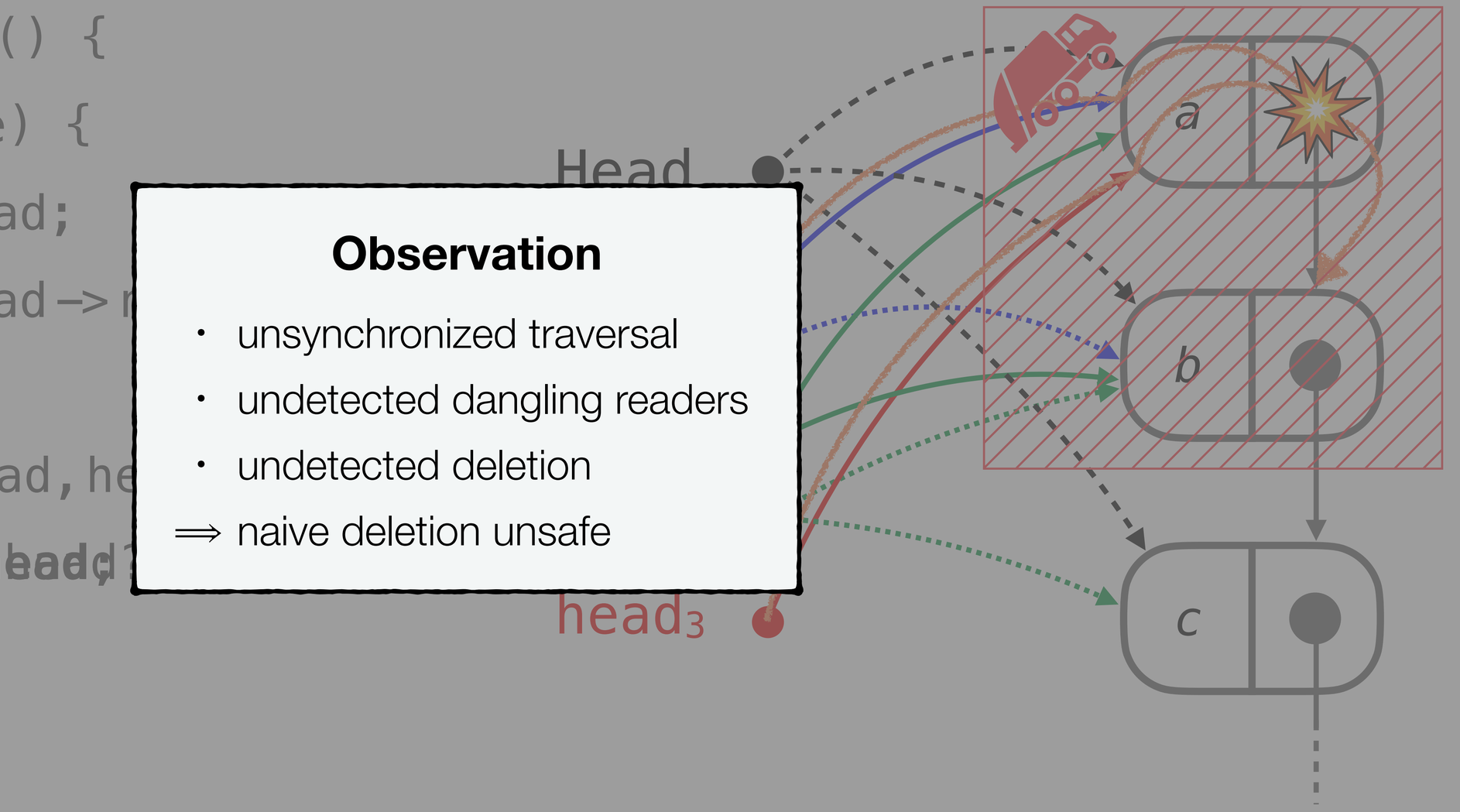
Non-blocking Queue (Michael&Scott)

```
void dequeue() {
  while (true) {
1 2 3 head = Head;
    next = head->n
    // ...
    if(CAS(Head,he
      delete head;
      return;
}}}
```

Observation

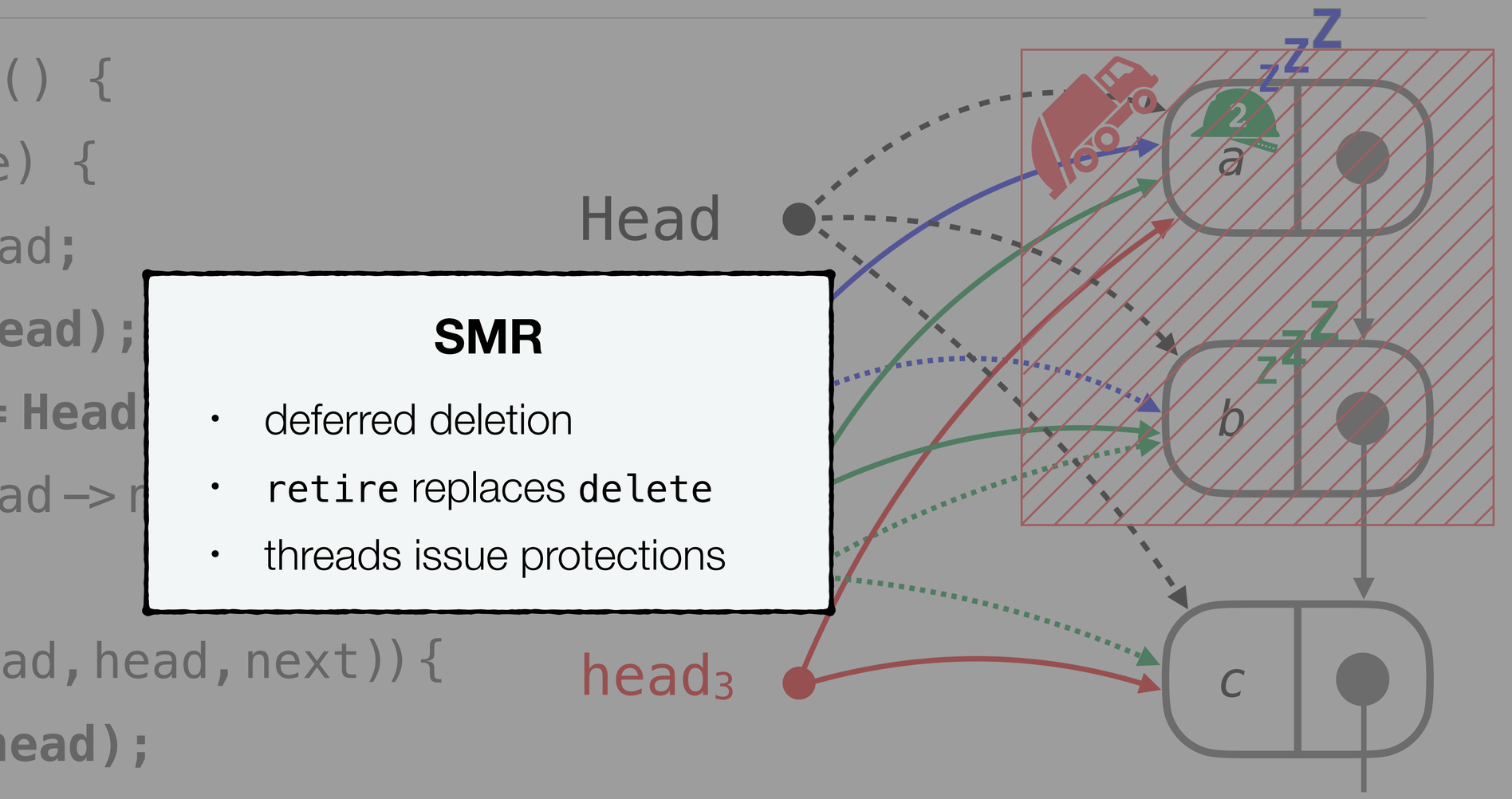
- unsynchronized traversal
- undetected dangling readers
- undetected deletion

$\Longrightarrow$ naive deletion unsafe



Safe Memory Reclamation (SMR)

```
void dequeue() {
  while (true) {
1 2 3 head = Head;
    protect(head);
    if(head != Head
    next = head->n
    // ...
    if(CAS(Head,head,next)){
      retire(head);
      return;
}}}
```



SMR

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

Non-blocking Queue (Michael&Scott)

```
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

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**

(non-blocking data structure)^2

6. **reclamation**

GC-techniques do not work (well)!

In particular: ownership reasoning

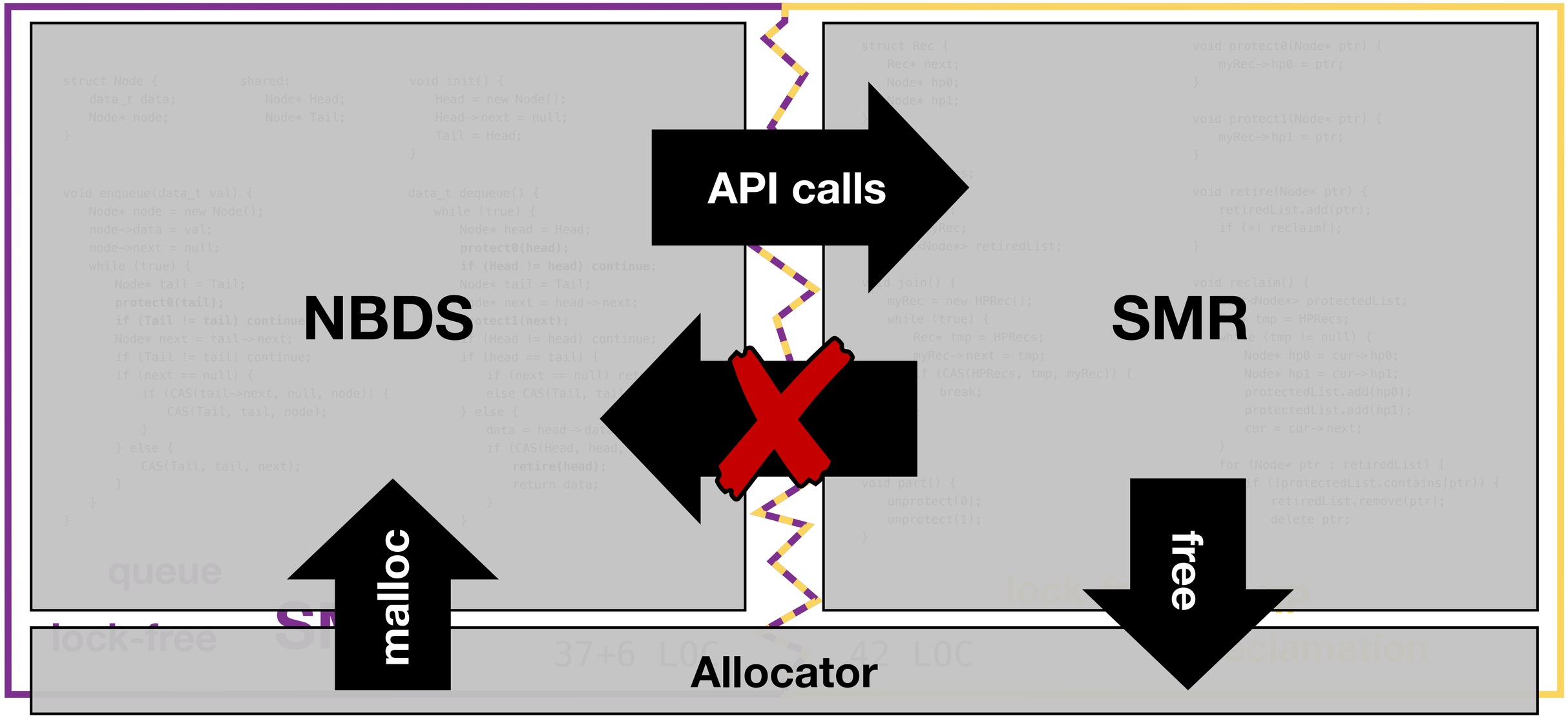
Our Approach

?

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

Contribution 1: Compositional Verification

Compositionality in Practice



Compositionality in Verification

API between **data structure** and **SMR impl**

- ➡ give a formal specification **SMR spec**
- ➡ **SMR spec** states *which&when* addresses are freed
- ➡ use new class of automata as **SMR spec**

Our Approach

SMR impl $\models$ SMR spec ?

Data structure + SMR spec $\models$ Property

Data structure + SMR impl $\models$ Property

[POPL'19]

with Roland Meyer

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.

Compositionality on an Example

```
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

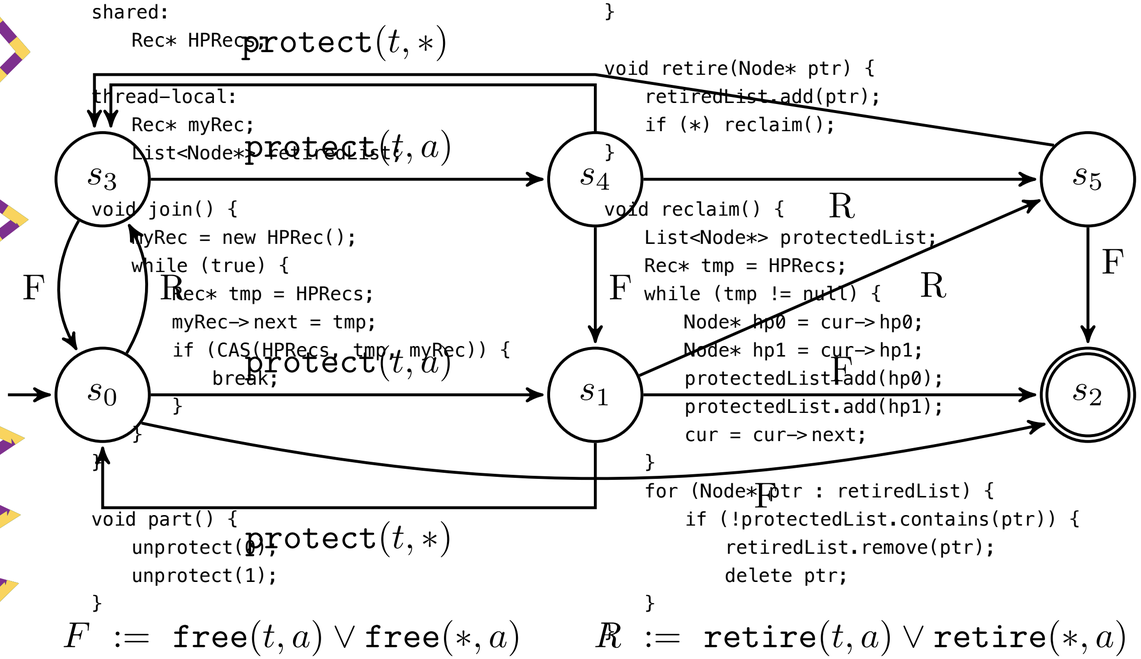
lock-free

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;
        }
    }
}
```



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

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

lock-free

automaton

42 LOC

HP

no reclamation

Experiments

- **SMR impl** ⊨ **SMR spec**:

SMR implementation	SMR spec size	Correctness
Hazard Pointers (HP)	3x5x5	**1.5s** ✓
Epoch-based Reclamation (EBR)	3x5	**11.2s** ✓

- **Data structure** + **SMR spec** ⊨ **Linearizability**

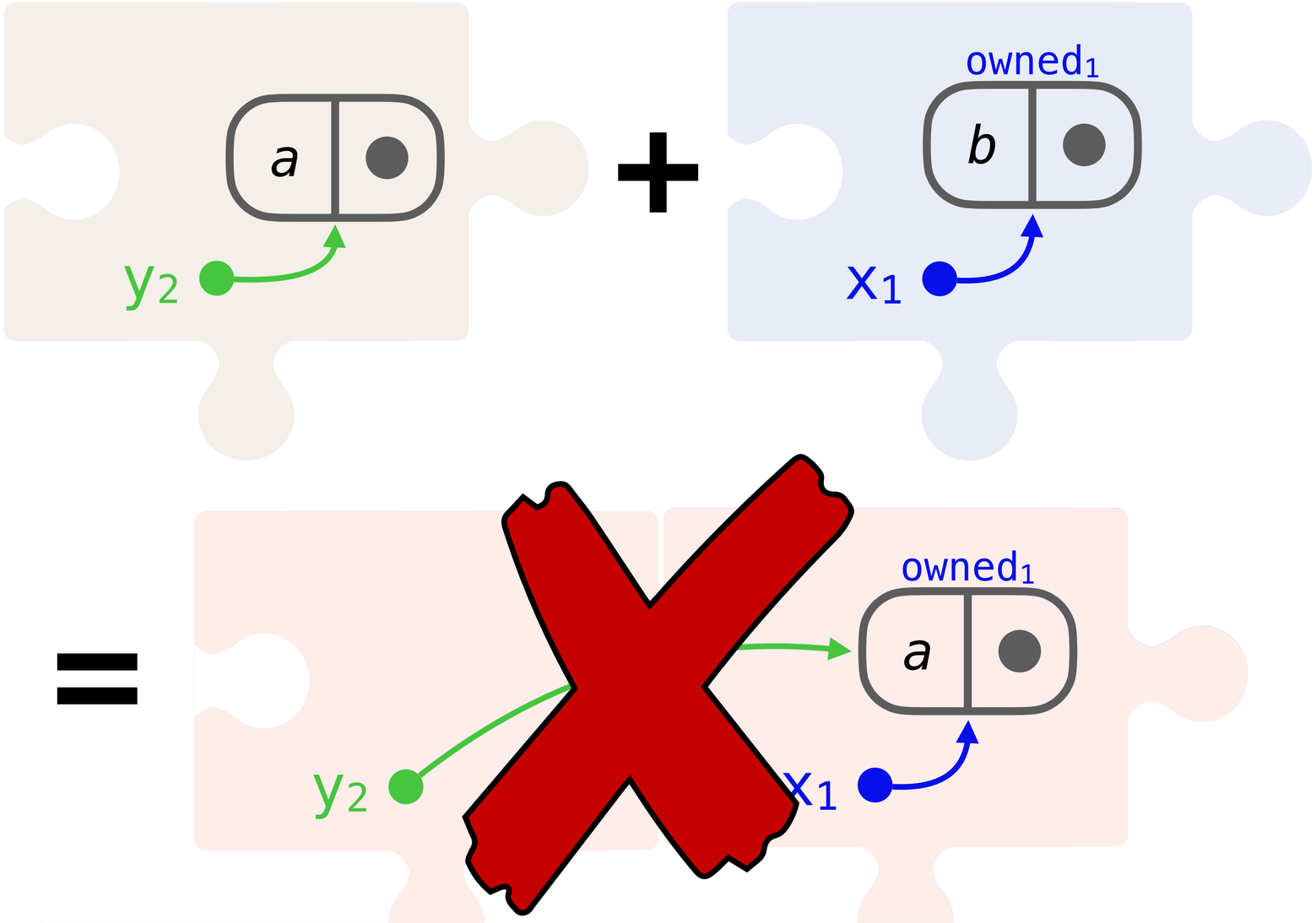
Infeasible: imprecision and state space explosion!

Contribution 2: Weak Ownership

Ownership Reasoning

Traditional:

- ➡ allocation grants ownership
- ➡ ownership grants exclusive access
- ➡ exclusivity crucial for verifier/tool precision
- ➡ unsound for SMR



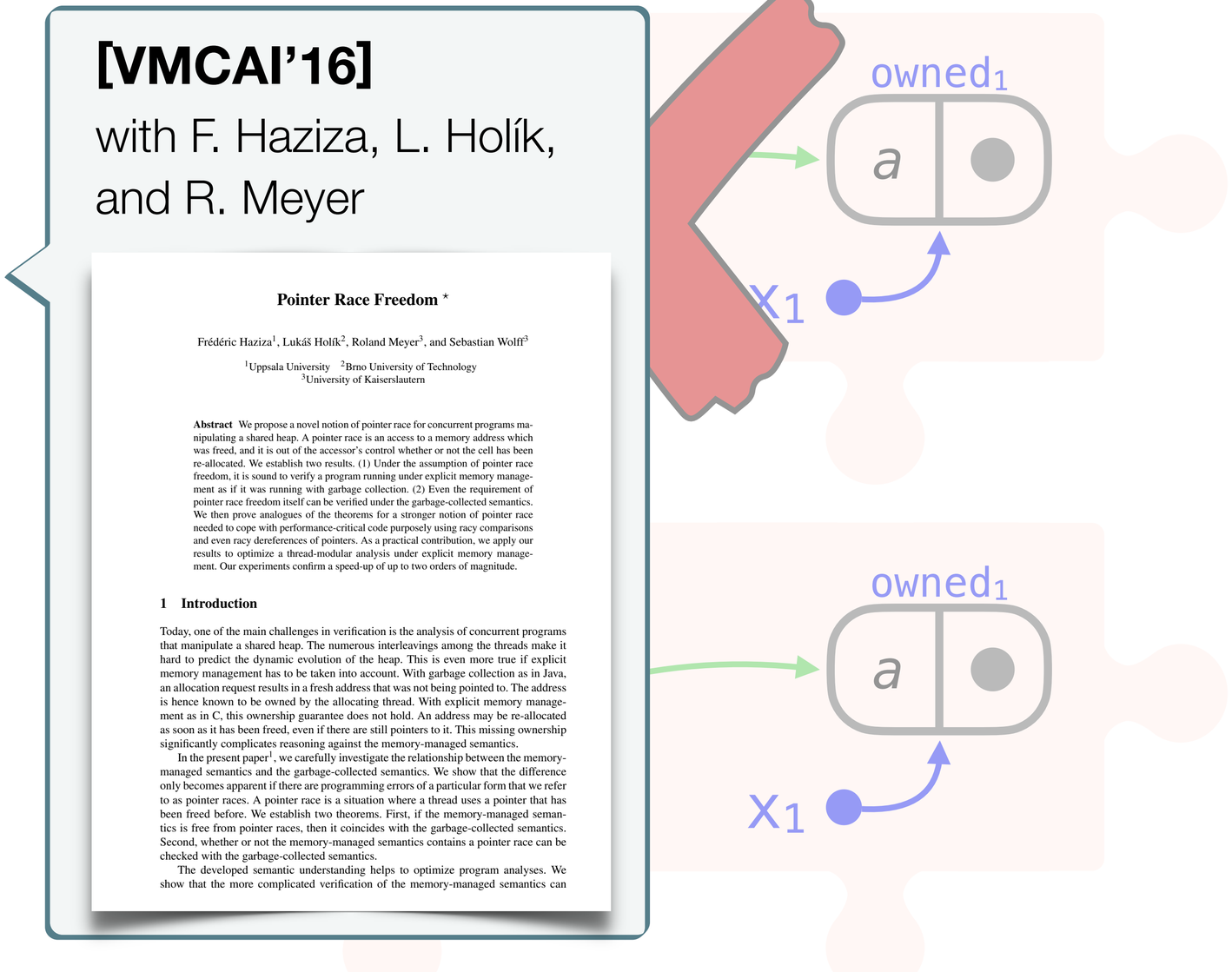
Ownership Reasoning

Weak:

- ➡ exclusivity only for non-dangling pointers
- ➡ sufficient precision
- ➡ sound for SMR

[VMCAI'16]

with F. Haziza, L. Holík, and R. Meyer



Pointer Race Freedom ⋆

Frédéric Haziza[1], Lukáš Holík[2], Roland Meyer[3], and Sebastian Wolff[3]

[1]Uppsala University [2]Brno University of Technology
[3]University of Kaiserslautern

Abstract We propose a novel notion of pointer race for concurrent programs manipulating a shared heap. A pointer race is an access to a memory address which was freed, and it is out of the accessor's control whether or not the cell has been re-allocated. We establish two results. (1) Under the assumption of pointer race freedom, it is sound to verify a program running under explicit memory management as if it was running with garbage collection. (2) Even the requirement of pointer race freedom itself can be verified under the garbage-collected semantics. We then prove analogues of the theorems for a stronger notion of pointer race needed to cope with performance-critical code purposely using racy comparisons and even racy dereferences of pointers. As a practical contribution, we apply our results to optimize a thread-modular analysis under explicit memory management. Our experiments confirm a speed-up of up to two orders of magnitude.

1 Introduction

Today, one of the main challenges in verification is the analysis of concurrent programs that manipulate a shared heap. The numerous interleavings among the threads make it hard to predict the dynamic evolution of the heap. This is even more true if explicit memory management has to be taken into account. With garbage collection as in Java, an allocation request results in a fresh address that was not being pointed to. The address is hence known to be owned by the allocating thread. With explicit memory management as in C, this ownership guarantee does not hold. An address may be re-allocated as soon as it has been freed, even if there are still pointers to it. This missing ownership significantly complicates reasoning against the memory-managed semantics.

In the present paper[1], we carefully investigate the relationship between the memory-managed semantics and the garbage-collected semantics. We show that the difference only becomes apparent if there are programming errors of a particular form that we refer to as pointer races. A pointer race is a situation where a thread uses a pointer that has been freed before. We establish two theorems. First, if the memory-managed semantics is free from pointer races, then it coincides with the garbage-collected semantics. Second, whether or not the memory-managed semantics contains a pointer race can be checked with the garbage-collected semantics.

The developed semantic understanding helps to optimize program analyses. We show that the more complicated verification of the memory-managed semantics can

Experiments

Data Structure with FL	Ownership	Linearizability
Treiber's stack	traditional	**0.06s** ✓
	weak	**2.4s** ✓
	none	**10m** ✓
Michael&Scott's queue	traditional	**2.5s** ✓
	weak	**3h** ✓
	none	– ✗

Impractical for more complex data structures / SMRs!

Contribution 3: ABA Freedom

ABAs

- Problem
  - ➡ impracticality due to state space explosion
  - ➡ state space explosion due to reallocations
- Observation
  - ➡ verification under GC instead of SMR is unsound
  - ➡ discrepancy manifests as ABA

State Space Reduction

- Theorem 1:

Verifying a data structure under garbage collection is sound,

if data structure + SMR spec it is free from ABAs.

- Theorem 2:

For an ABA check of data structure + SMR spec,

it is sound to restrict re-allocations to a single address.

(Side condition: **SMR spec** is invariant to re-allocations.)

Our Approach

SMR impl ⊨ SMR spec

~~Data structure + SMR spec ⊨ Property~~

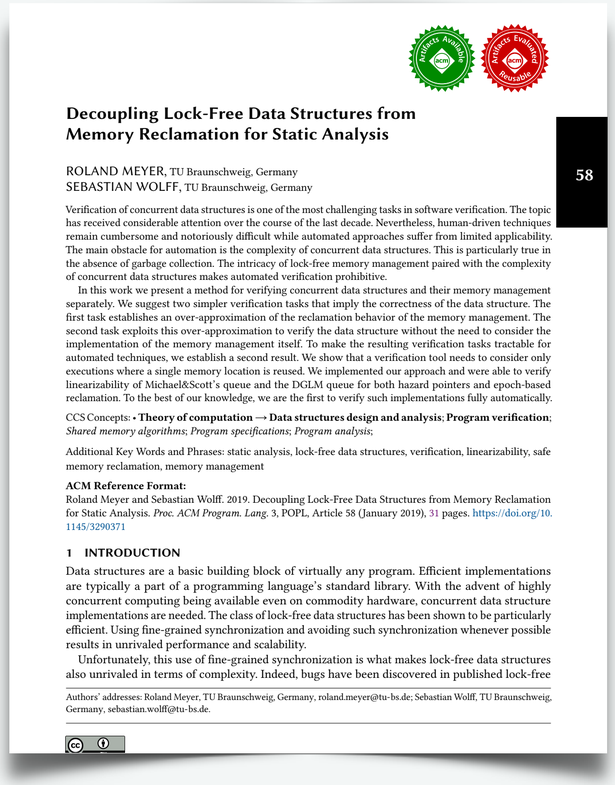
Data structure + Garbage Collection ⊨ Property

Data structure + SMR spec ⊨ ABA Freedom

Data structure + SMR impl ⊨ Property

[POPL'19]

with Roland Meyer



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.

Experiments

Data Structures	SMR	ABA Freedom	Linearizability
Treiber's stack	EBR	**16s** ✓	– ✓
	HP	**19s** ✓	– ✓
Michael&Scott's queue	EBR	**44m** ✓	– ✓
	HP	**120m** ✓	– ✓
DGLM queue	EBR	**63m** ✓	– ✓
	HP	**117m** ✓	– ✓

Reallocations remain the bottleneck!

Contribution 4: Pointer Race Freedom

Pointer Races

- Observation:

  In every ABA participates at least one dangling pointer.

- Pointer races
  - ➡ operations involving dangling pointers
  - ➡ operations that observe reclamation

- Pointer race freedom $\Longrightarrow$ no ABAs + memory safety

Pointer Race Check

Type system:

- ➡ to establish:

  Data structure + **SMR spec** ⊢ **Pointer Race Freedom**

- ➡ that is parametrized by the **SMR spec**

- ➡ externalizes shape analysis to a **GC** tool

External Shape Analysis

- Problem
  - ➡ danglingness may depend on shape invariant
  - ➡ type checks bad at finding shape invariants
- Solution
  - ➡ *“don’t try”*
  - ➡ annotate **data structure** with light-weight **annotations**
  - ➡ discharge **annotations** under **garbage collection**
  - ➡ guess&check finds annotations automatically

Our Approach

SMR impl ⊨ **SMR spec**

Data structure + **Garbage Collection** ⊨ **Property**

~~**Data structure** + **SMR spec** ⊢ **ABA Freedom**~~

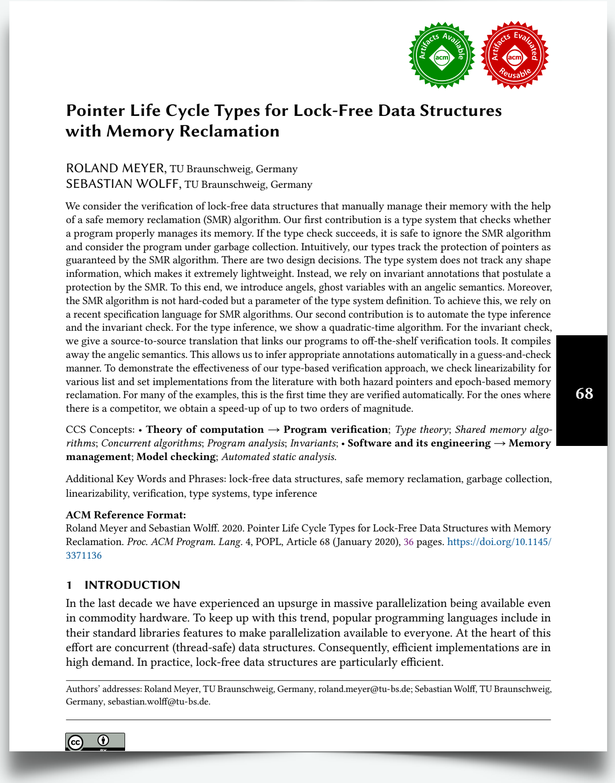
Annot. Data structure + **SMR spec** ⊢ **Pointer Race Freedom**

Data structure + **Garbage Collection** ⊨ **Annotations**

Data structure + **SMR impl** ⊨ **Property**

[POPL'20]

with Roland Meyer



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

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

We consider the verification of lock-free data structures that manually manage their memory with the help of a safe memory reclamation (SMR) algorithm. Our first contribution is a type system that checks whether a program properly manages its memory. If the type check succeeds, it is safe to ignore the SMR algorithm and consider the program under garbage collection. Intuitively, our types track the protection of pointers as guaranteed by the SMR algorithm. There are two design decisions. The type system does not track any shape information, which makes it extremely lightweight. Instead, we rely on invariant annotations that postulate a protection by the SMR. To this end, we introduce angels, ghost variables with an angelic semantics. Moreover, the SMR algorithm is not hard-coded but a parameter of the type system definition. To achieve this, we rely on a recent specification language for SMR algorithms. Our second contribution is to automate the type inference and the invariant check. For the type inference, we show a quadratic-time algorithm. For the invariant check, we give a source-to-source translation that links our programs to off-the-shelf verification tools. It compiles away the angelic semantics. This allows us to infer appropriate annotations automatically in a guess-and-check manner. To demonstrate the effectiveness of our type-based verification approach, we check linearizability for various list and set implementations from the literature with both hazard pointers and epoch-based memory reclamation. For many of the examples, this is the first time they are verified automatically. For the ones where there is a competitor, we obtain a speed-up of up to two orders of magnitude.

68

CCS Concepts: • **Theory of computation** → **Program verification**; *Type theory*; *Shared memory algorithms*; *Concurrent algorithms*; *Program analysis*; *Invariants*; • **Software and its engineering** → **Memory management**; **Model checking**; *Automated static analysis*.

Additional Key Words and Phrases: lock-free data structures, safe memory reclamation, garbage collection, linearizability, verification, type systems, type inference

ACM Reference Format:
Roland Meyer and Sebastian Wolff. 2020. Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation. *Proc. ACM Program. Lang.* 4, POPL, Article 68 (January 2020), 36 pages. https://doi.org/10.1145/3371136

1 INTRODUCTION

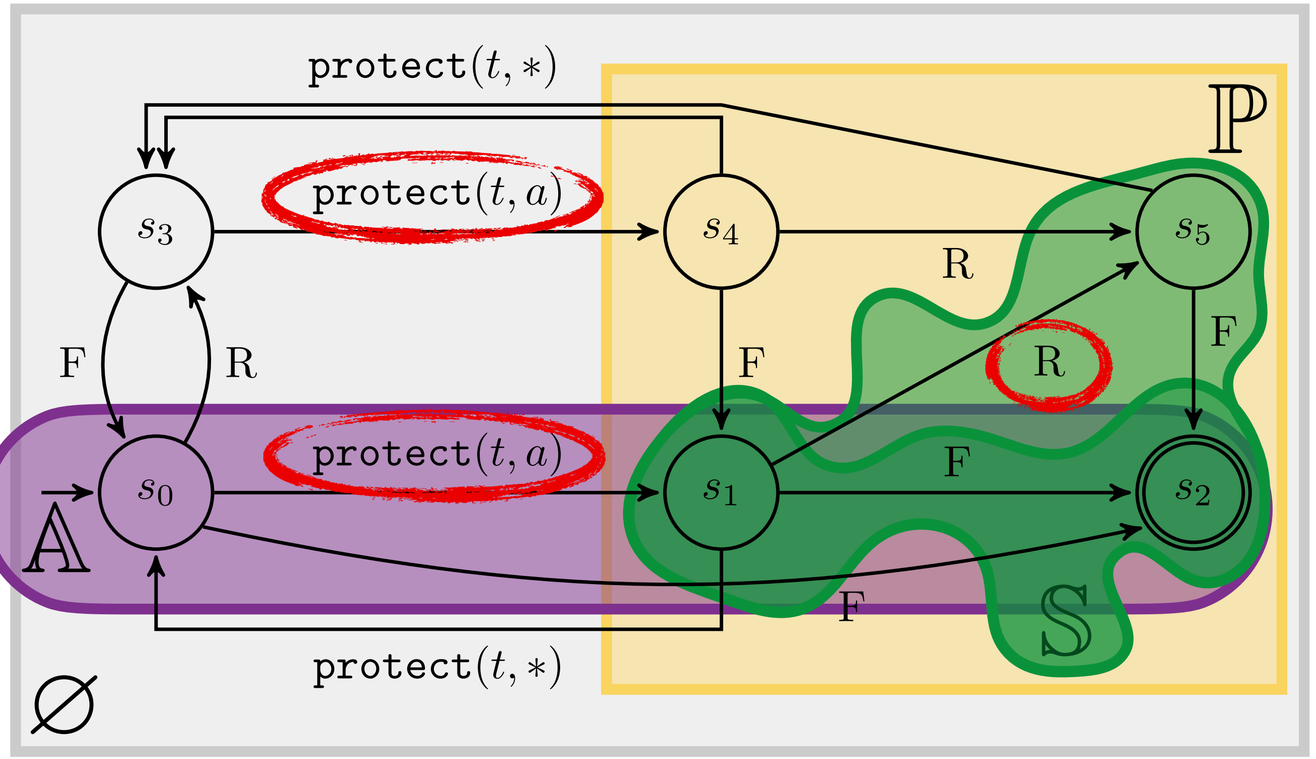
In the last decade we have experienced an upsurge in massive parallelization being available even in commodity hardware. To keep up with this trend, popular programming languages include in their standard libraries features to make parallelization available to everyone. At the heart of this effort are concurrent (thread-safe) data structures. Consequently, efficient implementations are in high demand. In practice, lock-free data structures are particularly efficient.

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

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**



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

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

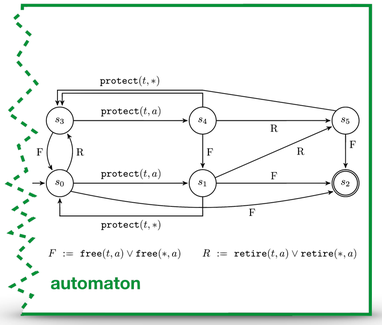
Experiments

Data Structure with HP	Types	Annot.	Lin.
Treiber's stack	0.7s ✓	12s ✓	1s ✓
Michael&Scott's queue	0.6s ✓	11s ✓	4s ✓
DGLM queue	0.6s ✓	1s ✗*	5s ✓
Vechev&Yahav's 2CAS 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

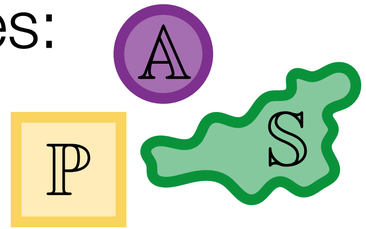
* imprecision in the back-end verifier

Final Approach

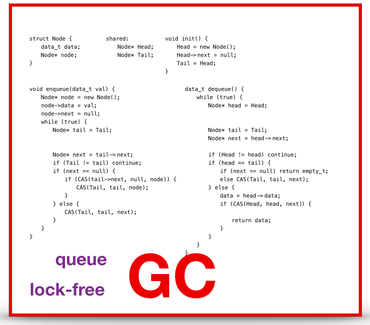
Thanks



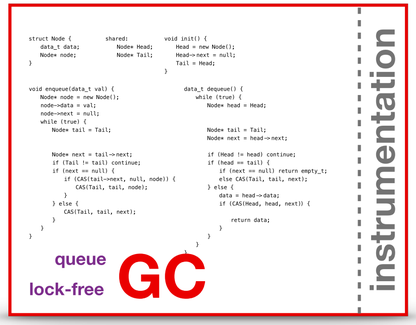
Types:



Type Check



Verification



Annotation Check

off-the-shelf verifier

Fin.