Modular Verification of a Non-Blocking Stack

Matthew Parkinson Richard Bornat Peter O'Hearn

Lecture for 15-818A4 Separation Logic
Henry DeYoung
April 25, 2011

Introduction

Concurrent Separation Logic

Guiding Principles:

- *Dijkstra:* Limit interference to rare moments of synchronization.
- *Separation Logic:* "Separate parts of a program that depend on separated resources can be dealt with independently."

An Approach: Use units of mutual exclusion to synchronize.

- Inefficient — Threads often must wait for exclusive access.
- Is there an effective treatment of *non-blocking concurrency*?

Non-Blocking Concurrency

Threads attempt to make concurrent changes to shared data structures, looping and trying again if an attempt fails.

- Algorithms rely on atomic hardware operations
  - Atomic reads and writes to single-word store (heap?) locations
  - `CAS` (compare-and-set) instruction

    $\texttt{CAS}(\ell, v_1, v_2)$: If $\ell \mapsto v_1$, then atomically mutate so $\ell \mapsto v_2$. Otherwise, skip.

Isn't this just more finely-grained mutual exclusion?

- *"An atomic access to the [shared] data structure is treated as a unit of mutual exclusion."*

Rely-Guarantee for Non-Blocking Concurrency

Each thread must specify

- properties which it *guarantees* to preserve, and
- properties which it *relies* on other threads to preserve.

Thread interference floods the proofs of clients!

- Every instruction in every thread must ensure the guarantees of its thread, because every specification must include those guarantees.

Can separation logic be used to contain thread interference?

Separation Logic and Non-Blocking Concurrency

- Shared data structure, equipped with an invariant
- Operations on data structure as procedures:
  - Non-blocking interaction through concurrent execution of procedures
  - Pre- and post-conditions of procedures do no involve invariant
  - Invariant is exposed within atomic operations in procedure bodies

"Even though the proofs [of the procedure bodies] may be horrible, their horrors are confined, and we can consider threads to be independent when outside those procedures."

Concurrent Separation Logic, "Simplified"

Concurrent Separation Logic, Simplified

$$\Gamma; I \vdash \{Q\}\ C\ \{R\}$$

- Procedure specifications $\{Q'\}\ f(\vec{x})\ \{R'\}$
- *Single* resource invariant for the shared data structure

$$\frac{\Gamma; I \vdash \{Q_1\}\ C_1\ \{R_1\} \qquad \Gamma; I \vdash \{Q_2\}\ C_2\ \{R_2\}}{\Gamma; I \vdash \{Q_1 \star Q_2\}\ C_1 \| C_2\ \{R_1 \star R_2\}}$$

Standard rule for parallel composition in concurrent separation logic.

The `atomic` Rule

$$\frac{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ C\ \{R \star I\}}{\Gamma; I \vdash \{Q\}\ \texttt{atomic\{}C\texttt{\}}\ \{R\}}$$

C must execute in mutual exclusion with `atomic{...}` commands!

- Rely on serialization properties of the hardware.
- C may invoke at most one single-word read or write in the shared data structure, or must be a `CAS` instruction.

Special case of the CCR rule when there is only one resource.

- It follows that `atomic{atomic{...}}` should not terminate.

Can one enforce the side condition more syntactically? E.g.,

$$\frac{\vdash \{Q \star I\}\ A\ \{R \star I\}}{\Gamma; I \vdash \{Q\}\ \texttt{atomic\{}A\texttt{\}}\ \{R\}}$$

where A is an atomic hardware operation.

The `module` Rule

$$\frac{\Gamma; I \vdash \{Q_f\}\ C_f\ \{R_f\} \qquad \Gamma, \{Q_f\} f(\vec{x_f}) \{R_f\}; \mathbf{emp} \vdash \{Q\}\ C\ \{R\}}{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ \texttt{module}\ f(\vec{x_f}) = C_f\ \texttt{in}\ C\ \{R \star I\}}$$

- Atomic instructions inside C_f can use the invariant — the rest of the program cannot.
- The procedure body itself is not executed atomically.
- Procedures cannot be recursive, apparently.
- "Extension to several procedures is obvious."

The `module` Rule

$$\frac{\begin{array}{l}\Gamma; I \vdash \{Q_f\}\ C_f\ \{R_f\} \\ \Gamma; I \vdash \{Q_g\}\ C_g\ \{R_g\} \\ \Gamma, \{Q_f\} f(\vec{x_f}) \{R_f\}, \{Q_g\} g(\vec{x_g}) \{R_g\}; \mathbf{emp} \vdash \{Q\}\ C\ \{R\}\end{array}}{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ \texttt{module}\ f(\vec{x_f}) = C_f\ \texttt{and}\ g(\vec{x_g}) = C_g\ \texttt{in}\ C\ \{R \star I\}}$$

- Atomic instructions inside C_f can use the invariant — the rest of the program cannot.
- The procedure body itself is not executed atomically.
- Procedures cannot be recursive, apparently.
- "Extension to several procedures is obvious."

The `module` Rule, cont.

"A more general approach: the `module` rule can be derived from

$$\frac{\Gamma; I_1 \star I_2 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I_1 \vdash \{Q \star I_2\}\ C\ \{R \star I_2\}} \quad \text{and} \quad \frac{\Gamma; I_1 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I_1 \star I_2 \vdash \{Q\}\ C\ \{R\}}\ ."$$

Define $\texttt{module}\ f(\vec{x_f}) = C_f\ \texttt{in}\ C$ as $\begin{array}{l}\texttt{resource}\ r\ \texttt{in} \\ \texttt{let}\ f(\vec{x_f}) = C_f\ \texttt{in}\ C\end{array}$

with $r \notin \texttt{res}(C)$.

$$\frac{\Gamma; \mathbf{emp} \star I \vdash \{Q_f\}\ C_f\ \{R_f\} \qquad \dfrac{\Gamma, \{Q_f\} C_f \{R_f\}; \mathbf{emp} \vdash \{Q\}\ C\ \{R\}}{\Gamma, \{Q_f\} C_f \{R_f\}; \mathbf{emp} \star I \vdash \{Q\}\ C\ \{R\}}}{\dfrac{\Gamma; \mathbf{emp} \star I \vdash \{Q\}\ \texttt{let}\ f(\vec{x_f}) = C_f\ \texttt{in}\ C\ \{R\}}{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ \texttt{resource}\ r\ \texttt{in}\ \texttt{let}\ f(\vec{x_f}) = C_f\ \texttt{in}\ C\ \{R \star I\}}}$$

The `module` Rule, cont.

"A more general approach: the `module` rule can be derived from

$$\frac{\Gamma; I_1 \star I_2 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I_1 \vdash \{Q \star I_2\}\ C\ \{R \star I_2\}} \quad \text{and} \quad \frac{\Gamma; I_1 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I_1 \star I_2 \vdash \{Q\}\ C\ \{R\}}."$$

"Note that these two rules make the standard frame rule derivable."

$$\frac{\dfrac{\Gamma; I_1 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I_1 \star I_2 \vdash \{Q\}\ C\ \{R\}}}{\Gamma; I_1 \vdash \{Q \star I_2\}\ C\ \{R \star I_2\}}$$

This seems odd because the bottom rule does not declare a resource.

The `CAS` Rule

$$\frac{\begin{array}{c}\Gamma; I \vdash \{(Q \star e \mapsto e_2) \wedge e' = e_1\}\ C_1\ \{R\} \\ \Gamma; I \vdash \{(Q \star e \mapsto e') \wedge e' \neq e_1\}\ C_2\ \{R\}\end{array}}{\Gamma; I \vdash \{Q \star e \mapsto e'\}\ \texttt{if}\ (\texttt{CAS}(e, e_1, e_2))\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{R\}}$$

$\texttt{CAS}(b; v, v_1, v_2)\{v'\} =$
  $\{v \mapsto v'\}$
  newvar v'' **in**
    $v'' := [v]$;
    $b := (v'' = v_1)$;
    if b **then**
      $[v] := v_2$
    else
      skip
  $\{(b \wedge v \mapsto v_2 \wedge v' = v_1) \vee$
   $(\neg b \wedge v \mapsto v' \wedge v' \neq v_1)\}$

$\{Q \star e \mapsto e'\}$
newvar b **in**
  $\texttt{CAS}(b; e, e_1, e_2)\{e'\}; \} \star Q$
  $\{(b \wedge (Q \star e \mapsto e_2) \wedge e' = e_1) \vee$
   $(\neg b \wedge (Q \star e \mapsto e') \wedge e' \neq e_1)\}$
  if b **then**
    $\{(Q \star e \mapsto e_2) \wedge e' = e_1\}$
    C_1
  else
    $\{(Q \star e \mapsto e') \wedge e' \neq e_1\}$
    C_2
$\{R\}$

Heap Cell Permissions

Three types of permissions:

1. Read-only permissions
2. Write permissions
3. Existence permissions

This paper requires only that the invariant shares permissions with each thread — not that threads share permissions with each other.

Read-Only Permissions:

$$E \mapsto F \iff E \overset{r}{\mapsto} F \star E \overset{r}{\mapsto} F$$

A read permission allows a heap cell to be read.

$$E \overset{r}{\mapsto} F \star E \overset{r}{\mapsto} F' \Rightarrow F = F'$$

Heap Cell Permissions

Three types of permissions:

1. Read-only permissions
2. Write permissions
3. Existence permissions

This paper requires only that the invariant shares permissions with each thread — not that threads share permissions with each other.

Write and Existence Permissions:

$$E \mapsto F \iff E \overset{w}{\mapsto} F \star E \overset{e}{\mapsto} -$$

A write permission allows mutation of heap cells. An existence permission allows nothing, but ensures existence.

$$E \overset{e}{\mapsto} - \star E' \overset{e}{\mapsto} - \Rightarrow E \neq E'$$

Correctness of Michael's Algorithm

Case Study: Michael's Algorithm

Michael's algorithm implements a shared stack.

- Think storage allocator for single-cell heap records.
  - Stack $\approx$ freelist, `pop()` $\approx$ `malloc()`, and `push(x)` $\approx$ `free(x)`
- `push` can fail because of interference from other threads.

Specifications of `pop()` and `push(`x`)`:

$$\{\mathbf{emp}\}\ \texttt{pop()}\ \{(ret \mapsto -) \vee (ret = \mathbf{nil} \wedge \mathbf{emp})\}$$
$$\{x \mapsto -\}\ \texttt{push}(x)\ \{(ret \wedge \mathbf{emp}) \vee (\neg ret \wedge x \mapsto -)\}$$

A Client of the Shared Stack

Interference between threads is confined to pop and push.

`alloc()` of memory manager:

- Uses (slower) system allocator if the stack is empty.
- Proof does not need to (directly) account for thread interference.

```
{emp}
alloc() {
   local y;
   {emp}
   y=pop();
   {(y = nil ∧ emp) ∨ y ↦ −}
   if (y==nil) {
      {emp}
      y=new();
      {y ↦ −}
   }
   {y ↦ −}
   return y;
}
{ret ↦ −}
```


push for a Shared Stack?

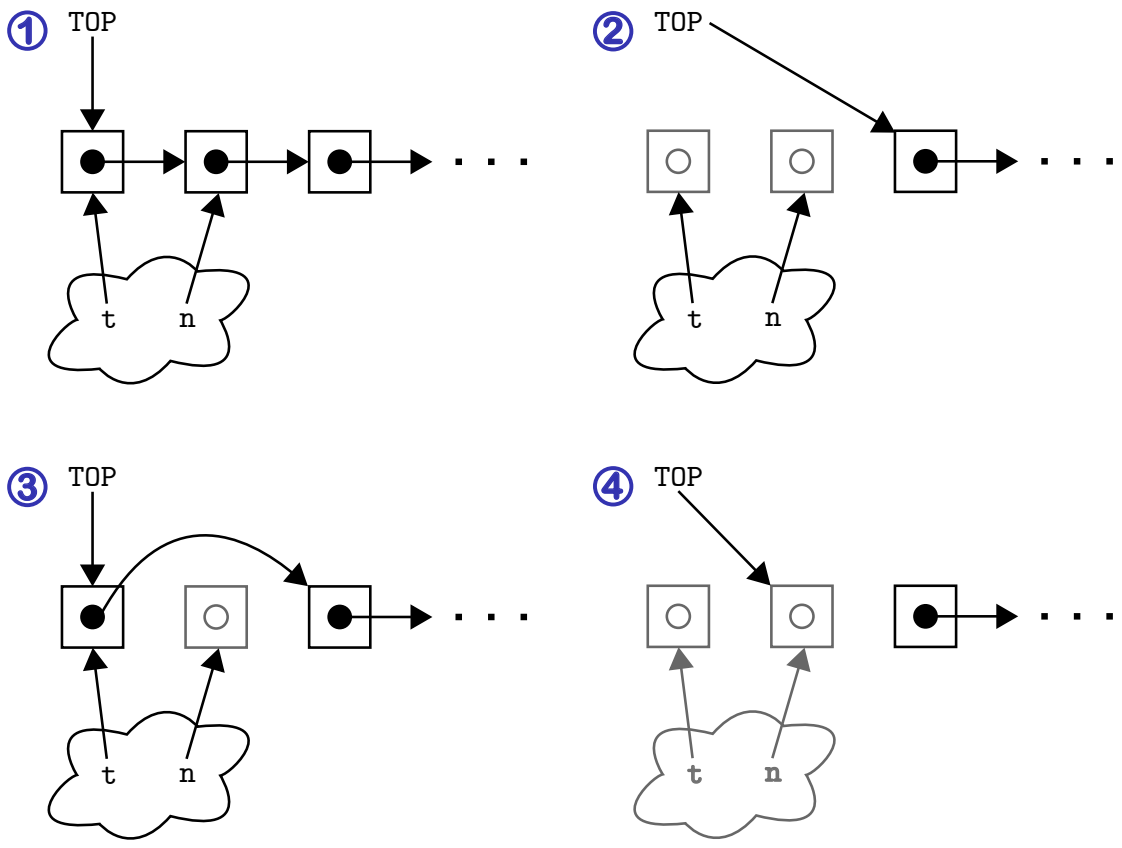
```
   {b ↦ −}
1  push(b) {
2    local t,n;
3    while(true) {
4      t = TOP;
5      b->tl = t;
6      if (CAS(&TOP, t, b))    // If TOP still matches b->tl,
7        break;                // then atomically set TOP to b.
8    }                         // Else, try pushing again.
9    return true;
10 }
   {ret ∧ emp}
```

pop for a Shared Stack?

```
1   pop() {
2     local t,n;
3     while(true) {
4       t = TOP;
5       if (t == nil)
6         break;
7       n = t->tl;
8       if (CAS(&TOP, t, n))    // If TOP still matches t, then
9         break;                // atomically set TOP to t->tl.
10    }                         // Else, try popping again.
11    return t;
12  }
```

pop() is broken by the ABA problem!

The ABA Problem



Fixing the ABA Problem

Solution: Add a global array H of 'hazard pointers'.

```
1  pop() {
2    local t,n;
3    while(true) {
4      t = TOP;
5      if (t == nil)
6        break;
7      H[tid] = t;
8      if (t != TOP) continue;
9      n = t->tl;
10     if (CAS(&TOP, t, n))
11       break;
12   }
13   H[tid] = nil;
14   return t;
15 }
```

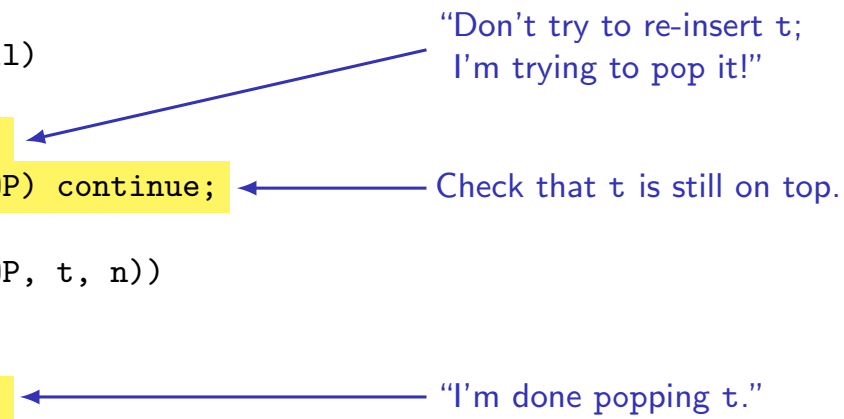
```
1  push(b) {
2    local t,n;
3    for (n=0; n<=THREADS; n++)
4      if (H[n] == b)
5        return false;
6    while(true) {
7      t = TOP;
8      b->tl = t;
9      if (CAS(&TOP, t, b))
10       break;
11   }
12   return true;
13 }
```

(†) Between execution of `H[tid] = t;` and assignment of another value to `H[tid]`, the cell pointed to by `t` will not be removed from the stack and subsequently re-inserted.

Fixing the ABA Problem

Solution: Add a global array H of 'hazard pointers'.

```
1  pop() {
2    local t,n;
3    while(true) {
4      t = TOP;
5      if (t == nil)
6        break;
7      H[tid] = t;
8      if (t != TOP) continue;
9      n = t->tl;
10     if (CAS(&TOP, t, n))
11       break;
12   }
13   H[tid] = nil;
14   return t;
15 }
```



(†) Between execution of `H[tid] = t;` and assignment of another value to `H[tid]`, the cell pointed to by `t` will not be removed from the stack and subsequently re-inserted.

Fixing the ABA Problem

Solution: Add a global array `H` of 'hazard pointers'.

Check that no other thread is trying to pop record b.

```
1  push(b) {
2    local t,n;
3    for (n=0; n<=THREADS; n++)
4      if (H[n] == b)
5        return false;
6    while(true) {
7      t = TOP;
8      b->tl = t;
9      if (CAS(&TOP, t, b))
10       break;
11   }
12   return true;
13 }
```

(†) Between execution of `H[tid] = t;` and assignment of another value to `H[tid]`, the cell pointed to by `t` will not be removed from the stack and subsequently re-inserted.

Fixing the ABA Problem

Solution: Add a global array `H` of 'hazard pointers'.

```
1  pop() {
2    local t,n;
3    while(true) {
4      t = TOP;
5      if (t == nil)
6        break;
7      H[tid] = t;
8      if (t != TOP) continue;
9      n = t->tl;
10     if (CAS(&TOP, t, n))
11       break;
12   }
13   H[tid] = nil;
14   return t;
15 }
```

```
1  push(b) {
2    local t,n;
3    for (n=0; n<=THREADS; n++)
4      if (H[n] == b)
5        return false;
6    while(true) {
7      t = TOP;
8      b->tl = t;
9      if (CAS(&TOP, t, b))
10       break;
11   }
12   return true;
13 }
```

(‡) Once a cell is inserted into the stack, its `tl` value will not be altered until it is removed.

Auxiliary State to Express Key Properties

Q: How can we express the (†) and (‡) properties?

A: Use auxiliary state to track the status of hazard pointers.

- A hazard pointer can be in one of four states:

Unset	`H[tid]` = **nil**.
Req	`H[tid]` = b and b has not been observed in the stack.
Tail(k)	`H[tid]` = b and b has been observed in the stack with tail k.
Left	`H[tid]` = b and b has been observed in the stack, but is now known not to be in the stack.

- If a hazard pointer is *Tail*(k) or *Left*, it shouldn't be `pushed`.

Adding Auxiliary Assignments to pop

```
1  pop() {
2    local t,n;
3    while(true) {
4      atomic{ t = TOP; }
5      if (t == nil) break;
6      atomic{ H[tid] = t; H'[tid] = Req; }
7      atomic{ if (TOP != t) continue;
8              else H'[tid] = Tail(t->tl); }
9      atomic{ n = t->tl;
10             if (H'[tid] != Tail(n))
11               H'[tid] = Left; }
12     atomic{ if (CAS(&TOP, t, n)) break; }
13   }
14   atomic{ H[tid] = nil; H'[tid] = Unset; }
15   return t;
16 }
```

`t` not yet observed in stack.

`H[tid]` is observed in stack.

If `t->tl` changed, then `t` was popped (‡). So, `t` can't have been re-inserted (†).

`H[tid]` is no longer set.

Over-cautious: Test `TOP == t` as surrogate for '`t` is somewhere in the stack'.

Adding Auxiliary Assignments to push

```
1  push(b) {
2    local t,n;
3    for (n=0; n<=THREADS; n++) {
4      atomic{
5        if (H[n] == b) { G[tid] = Unset; return false; }
6        G[tid] = NotHaz(b,{0,...,n});
7      }
8    }
9    while(true) {
10     atomic{ t = TOP; }
11     b->tl = t;
12     atomic{
13       if (CAS(&TOP, t, b)) { G[tid] = Unset; break; }
14     }
15   }
16   return true;
17 }
```

"I give up on pushing b."

"I'm trying to push b, and it's safe for 0, . . . , n."

"I'm done pushing b."

What if hazard pointers change after we've checked them?

- Can't happen — A cell being pushed is not in the stack. So, during the push, no other thread can set its hazard pointer to that cell's address.

Building Up the Invariant

Q: How does the stack constrain the hazard-pointer array h and hazard-status array h'?

A:

- If y is in the stack and is hazard-pointered for thread i, then its status is either $Tail(z)$ or Req — it cannot be $Left$ or $Unset$.

$$\begin{aligned}\mathsf{RNode}(y,z,h,h') \stackrel{\text{def}}{=}\ & y \mapsto z \wedge \\ & \forall i \in T.\ \big((h(i) = y) \Rightarrow \\ & \qquad (h'(i) = Tail(z) \vee h'(i) = Req)\big)\end{aligned}$$

- Apply this restriction to every element of the stack:

$$\begin{aligned}\mathsf{RList}(y,h,h') \stackrel{\text{def}}{=}\ & (y = \mathbf{nil} \wedge \mathbf{emp}) \vee \\ & \exists z.\ \big(\mathsf{RNode}(y,z,h,h') \star \mathsf{RList}(z,h,h')\big)\end{aligned}$$

Building Up the Invariant, cont.

Q: How does G restrict the hazard-pointer and -status arrays?

A:

$$\mathsf{HCons}(h, h') \stackrel{\text{def}}{=} \bigodot_{i \in T} \begin{pmatrix} \mathtt{G}[i] \overset{r}{\mapsto} \mathit{Unset} \vee \\ \exists b, S.\ \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\mapsto} \mathit{NotHaz}(b, S) \star b \overset{e}{\mapsto} -) \\ \wedge\, \forall j \in S.\ (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

For each thread i, either:

- thread i is not pushing anything; or
- thread i is pushing some b and:
  - b is not a confirmed hazard for any threads checked so far in the for-loop, and
  - b is not in the stack and not being pushed by another thread.

Building Up the Invariant, cont.

Q: How does G restrict the hazard-pointer and -status arrays?

A:

$$\mathsf{HCons}(h, h') \stackrel{\text{def}}{=} \bigodot_{i \in T} \begin{pmatrix} \mathtt{G}[i] \overset{r}{\mapsto} \mathit{Unset} \vee \\ \exists b, S.\ \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\mapsto} \mathit{NotHaz}(b, S) \star b \overset{e}{\mapsto} -) \\ \wedge\, \forall j \in S.\ (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

- If thread i is pushing some b, the thread gives $b \overset{e}{\mapsto} -$ to the invariant and only keeps $b \overset{w}{\mapsto} -$ for itself. Thus, the thread loses total permission and it cannot begin to push a cell and then pass the cell to another thread that finishes pushing b.
- The invariant provides only read permissions for G.
  Thread i will have the other read permission for $\mathtt{G}[i]$.

Finally, the Invariant

$$\mathsf{Inv} \stackrel{\mathsf{def}}{=} \exists h, h'. \begin{pmatrix} \left(\odot_{j \in T}.\, \mathtt{H}[j] \stackrel{r}{\mapsto} h(j) \star \mathtt{H'}[j] \stackrel{r}{\mapsto} h'(j)\right) \star \\ \mathsf{HCons}(h, h') \star \mathsf{RList}(\mathtt{TOP}, h, h') \star \\ \forall i \in T.\, \left(h(i) = \mathbf{nil} \Leftrightarrow h'(i) = \mathit{Unset}\right) \end{pmatrix}$$

- The arrays `H` and `H'` represent h and h', respectively.
- The invariant provides only read permissions for `H` and `H'`. Thread j will have the other read permission for $\mathtt{H}[j]$ and $\mathtt{H'}[j]$.
- h and h' are well-behaved with respect to the array `G`.
- h and h' are well-behaved with respect to the stack.
- h and h' are consistent for unset hazard pointers.

Should this instead be $\wedge$?

Some Details of the Proof

Some Details of pop

$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
`atomic{`
  $\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req} \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
  `if (TOP != t)`
    $\{\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} - \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} - \star \mathsf{Inv}\}$
    `continue;`
  `else`
    $\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{TOP} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req} \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil} \wedge \mathtt{TOP} = \mathtt{t}\}$
    `H'[tid]=`Tail`(t->tl);`
    $\{\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-) \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
`}`
$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-)) \wedge \mathtt{t} \neq \mathbf{nil}\}$

Recall

$$\mathsf{Inv} \overset{\text{def}}{=} \exists h, h'. \begin{pmatrix} (\odot_{j \in T}. \mathtt{H}[j] \overset{r}{\mapsto} h(j) \star \mathtt{H'}[j] \overset{r}{\mapsto} h'(j)) \star \\ \mathsf{HCons}(h, h') \star \mathsf{RList}(\mathtt{TOP}, h, h') \star \\ \forall i \in T. \ (h(i) = \mathbf{nil} \Leftrightarrow h'(i) = \mathit{Unset}) \end{pmatrix}$$

It suffices to show:

$$h(\mathtt{tid}) = \mathtt{t} \wedge \mathsf{RNode}(\mathtt{t}, n, h, h') \Rightarrow \mathsf{RNode}(\mathtt{t}, n, h, h'[\mathtt{tid} \mapsto \mathit{Tail}(n)])$$

Some Details of pop

$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
`atomic{`
  $\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req} \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
  `if (TOP != t)`
    $\{\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} - \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} - \star \mathsf{Inv}\}$
    `continue;`
  `else`
    $\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{TOP} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Req} \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil} \wedge \mathtt{TOP} = \mathtt{t}\}$
    `H'[tid]=`Tail`(t->tl);`
    $\{\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-) \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
`}`
$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-)) \wedge \mathtt{t} \neq \mathbf{nil}\}$

Recall

$$\mathsf{Inv} \overset{\text{def}}{=} \exists h, h'. \begin{pmatrix} (\odot_{j \in T}. \mathtt{H}[j] \overset{r}{\mapsto} h(j) \star \mathtt{H'}[j] \overset{r}{\mapsto} h'(j)) \star \\ \mathsf{HCons}(h, h') \star \mathsf{RList}(\mathtt{TOP}, h, h') \star \\ \forall i \in T. \ (h(i) = \mathbf{nil} \Leftrightarrow h'(i) = \mathit{Unset}) \end{pmatrix}$$

It suffices to show:

$$\begin{aligned} h(\mathtt{tid}) = \mathtt{t} \wedge \mathsf{HCons}(h, h') \star \mathtt{t} \mapsto n \\ \Rightarrow \mathsf{HCons}(h, h'[\mathtt{tid} \mapsto \mathit{Tail}(n)]) \star \mathtt{t} \mapsto n \end{aligned}$$

Some Details of pop, cont.

We need to show:

$$h(\texttt{tid}) = \texttt{t} \wedge \mathsf{HCons}(h, h') \star \texttt{t} \mapsto n \\ \Rightarrow \mathsf{HCons}(h, h'[\texttt{tid} \mapsto \mathit{Tail}(n)]) \star \texttt{t} \mapsto n$$

Recall

$$\mathsf{HCons}(h, h') \stackrel{\text{def}}{=} \bigodot_{i \in T} \begin{pmatrix} \texttt{G}[i] \stackrel{r}{\mapsto} \mathit{Unset} \vee \\ \exists b, S.\ \begin{pmatrix} (\texttt{G}[i] \stackrel{r}{\mapsto} \mathit{NotHaz}(b, S) \star b \stackrel{e}{\mapsto} -) \\ \wedge\, \forall j \in S.\ (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

For each thread i, there are two cases:

1. If $\texttt{G}[i] = \mathit{Unset}$, then setting $h'(\texttt{tid}) = \mathit{Tail}(n)$ is safe for i.
2. Otherwise, we have $\texttt{t} \stackrel{w}{\mapsto} n \star \texttt{t} \stackrel{e}{\mapsto} - \star b \stackrel{e}{\mapsto} - \Rightarrow \texttt{t} \neq b$, and setting $h'(\texttt{tid}) = \mathit{Tail}(n)$ is safe for i.

More Details of pop

$\{(\texttt{H[tid]} \stackrel{r}{\mapsto} \texttt{t} \star \texttt{H'[tid]} \stackrel{r}{\mapsto} \mathit{Tail}(-)) \wedge \texttt{t} \neq \textbf{nil}\}$

```
atomic{
```

$\{(\texttt{H[tid]} \stackrel{r}{\mapsto} \texttt{t} \star \texttt{H'[tid]} \stackrel{r}{\mapsto} \mathit{Tail}(-) \star \mathsf{Inv}) \wedge \texttt{t} \neq \textbf{nil}\}$

```
  n = t->tl;
  if (H'[tid] != Tail(n))
    H'[tid] = Left;
}
```

$\{\big(\texttt{H[tid]} \stackrel{r}{\mapsto} \texttt{t} \star (\texttt{H'[tid]} \stackrel{r}{\mapsto} \mathit{Tail}(\texttt{n}) \vee \texttt{H'[tid]} \stackrel{r}{\mapsto} \mathit{Left})\big) \wedge \texttt{t} \neq \textbf{nil}\}$

Case: If t is in the stack, i.e., $\mathsf{RList}(\texttt{TOP}, h, h') \Rightarrow \texttt{t} \mapsto - \star \textbf{true}$, then Inv gives us total permission for t.

Moreover, the test `H'[tid] != Tail(n)` will fail because

$$h(\texttt{tid}) = \texttt{t} \wedge h'(\texttt{tid}) = \mathit{Tail}(a) \\ \wedge\, \mathsf{RList}(\texttt{TOP}, h, h') \wedge (\texttt{t} \mapsto x \star \textbf{true}) \\ \Rightarrow x = a$$

because $(p \star q) \wedge e \hookrightarrow e' \Rightarrow ((p \wedge e \hookrightarrow e') \star q) \vee (p \star (q \wedge e \hookrightarrow e'))$.

More Details of pop

$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-)) \wedge \mathtt{t} \neq \mathbf{nil}\}$

```
atomic{
```
$\{(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(-) \star \mathsf{Inv}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
```
  n = t->tl;
  if (H'[tid] != Tail(n))
    H'[tid] = Left;
}
```
$\{\big(\mathtt{H}[\mathtt{tid}] \overset{r}{\mapsto} \mathtt{t} \star (\mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \vee \mathtt{H'}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Left})\big) \wedge \mathtt{t} \neq \mathbf{nil}\}$

Case: Otherwise, $\mathsf{RList}(\mathtt{TOP}, h, h') \Rightarrow \neg(\mathtt{t} \mapsto - \star \mathbf{true})$, and we do not have any permissions for `t`.

- Therefore, we must use a racy read:

$$\overline{\{\mathbf{emp}\}\ \mathtt{n = [t]}\ \{\mathbf{emp}\}}$$

  This reflects the algorithm's optimism in reading `t->tl`.
- But, how do we justify setting $h'(\mathtt{tid}) = \mathit{Left}$?

More Details of pop, cont.

We may set the status to *Left* because

- it follows from distributivity of $\wedge\, \neg(\mathtt{t} \mapsto - \star \mathbf{true})$ over $\star$ that

$$h(\mathtt{tid}) = \mathtt{t} \wedge \mathsf{RList}(\mathtt{TOP}, h, h') \wedge \neg(\mathtt{t} \mapsto - \star \mathbf{true}) \Rightarrow \mathsf{RList}(\mathtt{TOP}, h, h'[\mathtt{tid} \mapsto \mathit{Left}])$$

- and because

$$h(\mathtt{tid}) = \mathtt{t} \wedge h'(\mathtt{tid}) = \mathit{Tail}(-) \wedge \mathsf{HCons}(h, h') \Rightarrow \mathsf{HCons}(h, h'[\mathtt{tid} \mapsto \mathit{Left}])$$

  Recall

$$\mathsf{HCons}(h, h') \overset{\text{def}}{=} \bigodot_{i \in T} \begin{pmatrix} \mathtt{G}[i] \overset{r}{\mapsto} \mathit{Unset} \vee \\ \exists b, S.\ \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\mapsto} \mathit{NotHaz}(b, S) \star b \overset{e}{\mapsto} -) \\ \wedge\, \forall j \in S.\ (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

Still More Details of pop

$\{\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star (\mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \vee \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
atomic{
  $\{(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil}) \vee$
   $(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left} \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil})\}$
  if (CAS(&TOP, t, n))
    $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star (\mathtt{t} \doteq \mathbf{nil} \vee \mathtt{t} \mapsto -) \star \mathsf{Inv}\}$
    break;
  $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star \mathsf{Inv}\}$
}
$\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} -\}$

Case: Status is *Left*.

- The CAS will fail because:

$$\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left} \star \mathsf{Inv} \Rightarrow \mathtt{TOP} \neq \mathtt{t}$$

The postcondition follows immediately.

Still More Details of pop

$\{\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star (\mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \vee \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
atomic{
  $\{(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil}) \vee$
   $(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left} \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil})\}$
  if (CAS(&TOP, t, n))
    $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star (\mathtt{t} \doteq \mathbf{nil} \vee \mathtt{t} \mapsto -) \star \mathsf{Inv}\}$
    break;
  $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star \mathsf{Inv}\}$
}
$\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} -\}$

Case: Status is $\mathit{Tail}(\mathtt{n})$ and $\mathtt{TOP} \neq \mathtt{t}$.

- The CAS will fail, and the postcondition follows immediately.

Still More Details of pop

$\{\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star (\mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \vee \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left}) \wedge \mathtt{t} \neq \mathbf{nil}\}$
atomic{
  $\{(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Tail}(\mathtt{n}) \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil}) \vee$
   $(\mathtt{H[tid]} \overset{r}{\mapsto} \mathtt{t} \star \mathtt{H'[tid]} \overset{r}{\mapsto} \mathit{Left} \star \mathsf{Inv} \wedge \mathtt{t} \neq \mathbf{nil})\}$
  if (CAS(&TOP, t, n))
    $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star (\mathtt{t} \doteq \mathbf{nil} \vee \mathtt{t} \mapsto -) \star \mathsf{Inv}\}$
    break;
  $\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} - \star \mathsf{Inv}\}$
}
$\{\mathtt{H[tid]} \overset{r}{\mapsto} - \star \mathtt{H'[tid]} \overset{r}{\mapsto} -\}$

Case: Status is $\mathit{Tail}(\mathtt{n})$ and $\mathtt{TOP} = \mathtt{t}$.

- The postcondition follows from

$$h(\mathtt{tid}) = \mathtt{t} \wedge h'(\mathtt{tid}) = \mathit{Tail}(\mathtt{n}) \wedge \mathsf{RList}(\mathtt{t}, h, h') \wedge \mathtt{t} \neq \mathbf{nil} \Rightarrow \mathtt{t} \mapsto \mathtt{n} \star \mathsf{RList}(\mathtt{n}, h, h')$$

Some Details of push

$\{\mathtt{G[tid]} \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} \mathtt{t}\}$
atomic{
  $\{\mathtt{G[tid]} \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} \mathtt{t} \star \mathsf{Inv}\}$
  if (CAS(&TOP, t, b)) {
    G[tid] = *Unset*;
    $\{\mathtt{G[tid]} \overset{r}{\mapsto} \mathit{Unset} \star \mathsf{Inv}\}$
    break;
  }
  $\{\mathtt{G[tid]} \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} - \star \mathsf{Inv}\}$
}
$\{\mathtt{G[tid]} \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} -\}$

- If the CAS succeeds, then b is not a confirmed hazard because

$$\mathsf{HCons}(h, h') \star \mathtt{G[tid]} \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \Rightarrow \forall i \in T.\ (h(i) = \mathtt{b} \Rightarrow h'(i) = \mathit{Req}).$$

Recall that

$$\mathsf{HCons}(h, h') \overset{\text{def}}{=} \bigodot_{i \in T} \begin{pmatrix} \mathtt{G}[i] \overset{r}{\mapsto} \mathit{Unset} \vee \\ \exists b, S.\ \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\mapsto} \mathit{NotHaz}(b, S) \star b \overset{e}{\mapsto} -) \\ \wedge \forall j \in S.\ (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

Some Details of push

$\{\mathtt{G}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} \mathtt{t}\}$
`atomic{`
  $\{\mathtt{G}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} \mathtt{t} \star \mathsf{Inv}\}$
  `if (CAS(&TOP, t, b)) {`
    `G[tid] =` *Unset*`;`
    $\{\mathtt{G}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{Unset} \star \mathsf{Inv}\}$
    `break;`
  `}`
  $\{\mathtt{G}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} - \star \mathsf{Inv}\}$
`}`
$\{\mathtt{G}[\mathtt{tid}] \overset{r}{\mapsto} \mathit{NotHaz}(\mathtt{b}, T) \star \mathtt{b} \overset{w}{\mapsto} -\}$

- Therefore, it is safe to add b to the stack:

$$\mathtt{b} \mapsto \mathtt{t} \star \mathsf{RList}(\mathtt{t}, h, h') \star \left(\forall i \in T.\ (h(i) = \mathtt{b} \Rightarrow h'(i) = \mathit{Req})\right) \Rightarrow \mathsf{RList}(\mathtt{b}, h, h')$$

Recall that

$$\mathsf{RNode}(\mathtt{b}, \mathtt{t}, h, h') \overset{\text{def}}{=} \mathtt{b} \mapsto \mathtt{t} \wedge \forall i \in T.\ \left((h(i) = \mathtt{b}) \Rightarrow (h'(i) = \mathit{Tail}(\mathtt{t}) \vee h'(i) = \mathit{Req})\right)$$

Conclusion

Conclusion

Key Points:

- Formally contained inter-thread interference to operations on shared, non-blocking data structure.
- "Even though the proofs [of the procedure bodies] may be horrible, their horrors are confined."

Future Work:

- Liveness properties — lock/wait-freedom?
- Marriage of separation logic and rely/guarantee or temporal logic?