Proving Logical Atomicity using Lock Invariants

Roshan Sharma Shengyi Wang Alexander Oey
Anastasiia Evdokimova Lennart Beringer William Mansky

July 31, 2022

What We Have Done

- We use lock invariants to prove logically atomic specifications for concurrent data structures.

What We Have Done

- We use lock invariants to prove logically atomic specifications for concurrent data structures.
- The proofs are significantly more complicated than those that use TaDA-style lock specifications.

What We Have Done

- We use lock invariants to prove logically atomic specifications for concurrent data structures.
- The proofs are significantly more complicated than those that use TaDA-style lock specifications.
- This is the first foundational verification of a C implementation against logically atomic specifications.

Two Styles of Lock Specification

Lock Invariant Style [Gotsman et al., 2007; Hobor et al., 2008]

$$\{\ell \boxdot\!\!\rightarrow R\}\ \texttt{acquire}(\ell)\ \{R * \ell \boxdot\!\!\rightarrow R\}$$

$$\{R * \ell \boxdot\!\!\rightarrow R\}\ \texttt{release}(\ell)\ \{\ell \boxdot\!\!\rightarrow R\}$$

Two Styles of Lock Specification

Lock Invariant Style [Gotsman et al., 2007; Hobor et al., 2008]

$$\{\ell \boxdot\!\!\to R\}\ \texttt{acquire}(\ell)\ \{R * \ell \boxdot\!\!\to R\}$$

$$\{R * \ell \boxdot\!\!\to R\}\ \texttt{release}(\ell)\ \{\ell \boxdot\!\!\to R\}$$

TaDA Style [da Rocha Pinto et al., 2014]

$$\langle b.\ (\mathsf{L}(\ell) \wedge \neg b) \vee (\mathsf{U}(\ell) \wedge b)\rangle\ \texttt{acquire}(\ell)\ \langle \mathsf{L}(\ell) \wedge b\rangle$$

$$\langle \mathsf{L}(\ell)\rangle\ \texttt{release}(\ell)\ \langle \mathsf{U}(\ell)\rangle$$

Logically Atomic Specifications

Atomic Triple

$$\langle a.\ P_l \mid P_p(a) \rangle\ c\ \langle Q_l \mid Q_p(a) \rangle$$

Logically Atomic Specifications

Atomic Triple

$$\langle a.\ P_l \mid P_p(a) \rangle\ c\ \langle Q_l \mid Q_p(a) \rangle$$

Example: a specification for push

$$\langle s.\ \mathsf{is\_stack}_g\ p \mid \mathsf{stack}_g\ s \rangle\ \mathsf{push}(v)\ \langle \mathsf{is\_stack}_g\ p \mid \mathsf{stack}_g\ (v :: s) \rangle$$

Logically Atomic Specifications

Atomic Triple

$$\langle a.\ P_l \mid P_p(a)\rangle\ c\ \langle Q_l \mid Q_p(a)\rangle$$

Example: a specification for `push`

$$\langle s.\ \mathsf{is\_stack}_g\ p \mid \mathsf{stack}_g\ s\rangle\ \mathsf{push}(v)\ \langle \mathsf{is\_stack}_g\ p \mid \mathsf{stack}_g\ (v :: s)\rangle$$

Logically Atomic Specifications

Atomic Triple

$$\langle a.\ P_l \mid P_p(a)\rangle\ c\ \langle Q_l \mid Q_p(a)\rangle$$

Example: a specification for push

$$\langle s.\ \mathsf{is\_stack}\ p \mid \mathsf{stack}\ s\rangle\ \mathsf{push}(v)\ \langle \mathsf{is\_stack}\ p \mid \mathsf{stack}\ (v :: s)\rangle$$

VST and Iris

Verified Software Toolchain

Iris

Concurrent Binary Search Tree (BST) as an Example

Atomic Specifications

$$\langle m.\,\mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ m\rangle\ \mathtt{insert}(p, k, v)\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ (m[k \mapsto v])\rangle$$

$$\langle m.\,\mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ m\rangle\ \mathtt{lookup}(p, k)\ \langle v.\,\mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ m \wedge m(k) = v\rangle$$

$$\langle m.\,\mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ m\rangle\ \mathtt{delete}(p, k)\ \langle \mathsf{bst\_ref}\ p \mid \mathsf{bst\_abs}\ (m[k \mapsto \_])\rangle$$

Coarse-Grained Locking

Definitions of Data Structure Assertions

$$R_{cg} \triangleq \exists t.\ \mathsf{BST}\ p\ t * \mathsf{ghost\_bst}_{.5}\ t$$

$$\mathsf{bst\_ref}\ l \triangleq l \mapsto_{\pi} R_{cg}$$

$$\mathsf{bst\_abs}\ t \triangleq \mathsf{ghost\_bst}_{.5}\ t$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_{\pi} R_{cg} \mid \text{ghost\_bst}_{.5}\, t \rangle$$

```
acquire(l);
```

```
seq_insert(p, x, v);
```

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_{\pi} R_{cg} \mid \text{ghost\_bst}_{.5}\, t[x \mapsto v] \rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \mathsf{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);

seq_insert(p, x, v);

release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\to_{\pi} R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\to_{\pi} (\exists t.\ \mathsf{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);
```

$$\langle \mathsf{BST}\ p\ t_1 * \text{ghost\_bst}_{.5} t_1 * l \boxdot\!\!\to_{\pi} R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
seq_insert(p, x, v);
```

```
release(l);
```

$$\langle l \boxdot\!\!\to_{\pi} R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \text{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);
```

$$\langle \text{BST}\ p\ t_1 * \text{ghost\_bst}_{.5}\, t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\{\text{BST}\ p\ t_1\}$$

```
seq_insert(p, x, v);
```

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t \rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \mathsf{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t \rangle$$

```
acquire(l);
```

$$\langle \mathsf{BST}\ p\ t_1 * \text{ghost\_bst}_{.5} t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t \rangle$$

$$\{\mathsf{BST}\ p\ t_1\}$$

```
seq_insert(p, x, v);
```

$$\{\mathsf{BST}\ p\ t_1[x \mapsto v]\}$$

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v] \rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \text{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);
```

$$\langle \text{BST}\ p\ t_1 * \text{ghost\_bst}_{.5} t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\{\text{BST}\ p\ t_1\}$$

```
seq_insert(p, x, v);
```

$$\{\text{BST}\ p\ t_1[x \mapsto v]\}$$

$$\langle \text{BST}\ p\ t_1[x \mapsto v] * \text{ghost\_bst}_{.5} t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \text{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);
```

$$\langle \text{BST}\ p\ t_1 * \text{ghost\_bst}_{.5}\, t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\{\text{BST}\ p\ t_1\}$$

```
seq_insert(p, x, v);
```

$$\{\text{BST}\ p\ t_1[x \mapsto v]\}$$

$$\langle \text{BST}\ p\ t_1[x \mapsto v] * \text{ghost\_bst}_{.5}\, t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle \text{BST}\ p\ t[x \mapsto v] * \text{ghost\_bst}_{.5}\, t[x \mapsto v] * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Sketch Proof of Coarse-Grained Insertion of BST

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle l \boxdot\!\!\rightarrow_\pi (\exists t.\ \text{BST}\ p\ t * \text{ghost\_bst}_{.5}\ t) \mid \text{ghost\_bst}_{.5}\, t\rangle$$

```
acquire(l);
```

$$\langle \text{BST}\ p\ t_1 * \text{ghost\_bst}_{.5}\, t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\{\text{BST}\ p\ t_1\}$$

```
seq_insert(p, x, v);
```

$$\{\text{BST}\ p\ t_1[x \mapsto v]\}$$

$$\langle \text{BST}\ p\ t_1[x \mapsto v] * \text{ghost\_bst}_{.5}\, t_1 * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\, t\rangle$$

$$\langle \text{BST}\ p\ t[x \mapsto v] * \text{ghost\_bst}_{.5}\, t[x \mapsto v] * l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

```
release(l);
```

$$\langle l \boxdot\!\!\rightarrow_\pi R_{cg} \mid \text{ghost\_bst}_{.5}\ t[x \mapsto v]\rangle$$

Fine-Grained Locking: General Idea

- There are multiple locks on different pieces of the data structure

Fine-Grained Locking: General Idea

- There are multiple locks on different pieces of the data structure
- A ghost state represents the abstract state of a locked section alone

Fine-Grained Locking: General Idea

- There are multiple locks on different pieces of the data structure
- A ghost state represents the abstract state of a locked section alone
- The abstract state of the data structure is derived from the composition of the states of the locked components

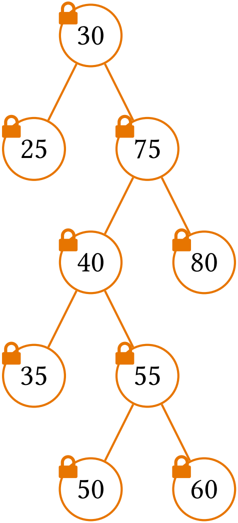
Fine-Grained Locking: General Idea

- There are multiple locks on different pieces of the data structure
- A ghost state represents the abstract state of a locked section alone
- The abstract state of the data structure is derived from the composition of the states of the locked components
- Each locked component c has a piece of ghost state ghost_c that is split between the lock invariant and the top-level abstract state

Fine-Grained Locking: General Idea

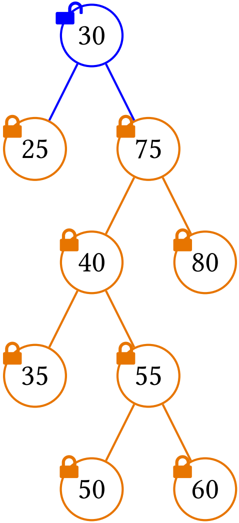
- There are multiple locks on different pieces of the data structure
- A ghost state represents the abstract state of a locked section alone
- The abstract state of the data structure is derived from the composition of the states of the locked components
- Each locked component c has a piece of ghost state ghost_c that is split between the lock invariant and the top-level abstract state
- The lock invariant for each component must also carefully account for the ownership of both that component's lock and the locks of related nodes

Hand-over-Hand Locking: BST Example



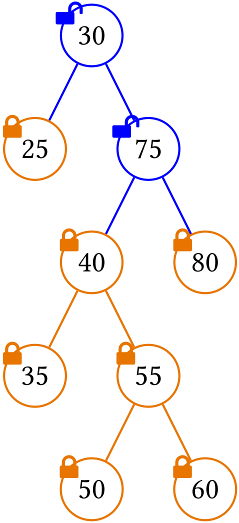

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



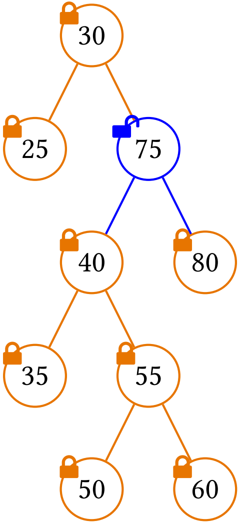

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



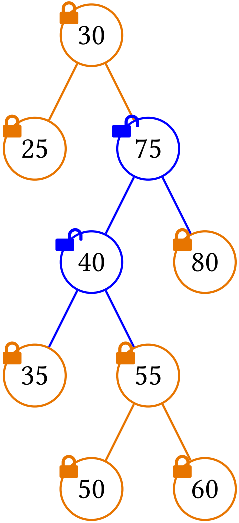

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



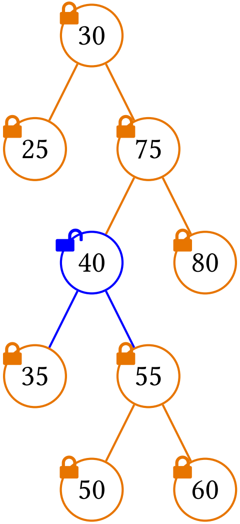

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



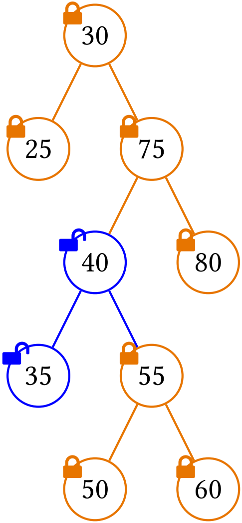

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



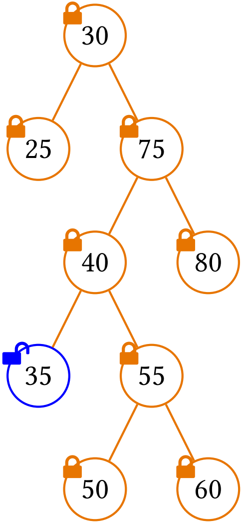

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



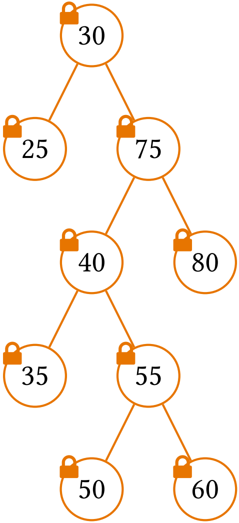

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example



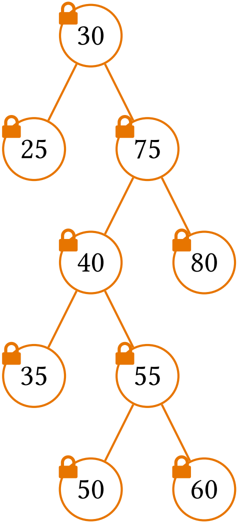

```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

Hand-over-Hand Locking: BST Example

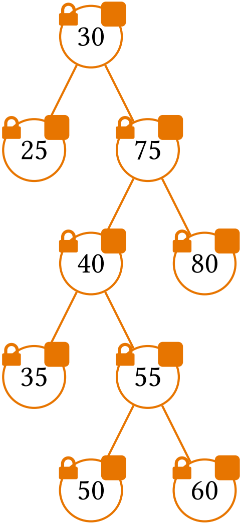



```
void *lookup (treebox t, int x) {
  acquire(l); p = tgt->t;
  while (p != NULL) {
    int y = p->key;
    if (x < y) {
      tgt = p->left; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else if (y < x) {
      tgt=p->right; void *l_old = l;
      l = tgt->lock; acquire(l);
      p=tgt->t; release(l_old);
    } else {
      v = p->value; release(l); return v;
    } ... }
```

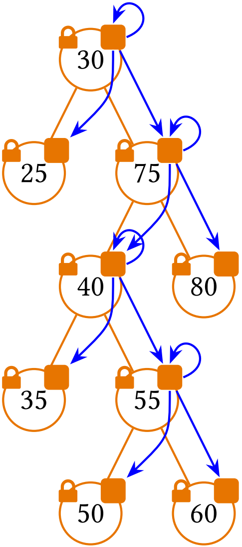
Recursive Lock Invariant



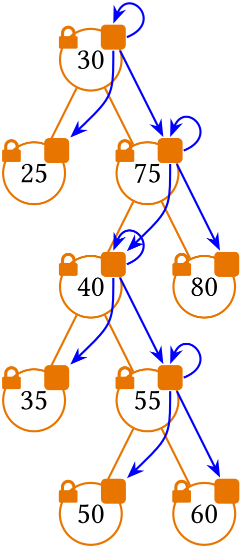
Recursive Lock Invariant



Recursive Lock Invariant



Recursive Lock Invariant



Lock Invariant: $\blacksquare_g(p)$

$\blacksquare_g(p) \triangleq \exists i\, j\, c.\ \text{👻}^{.5}_g((i,j),c) *$

$\quad p.\texttt{lock} \boxdot\!\!\longmapsto_{\pi_1} \blacksquare_g *$

$\quad \text{match } c \text{ with}$

$\quad |\ \text{None} \Rightarrow p.\texttt{t} = \texttt{NULL}$

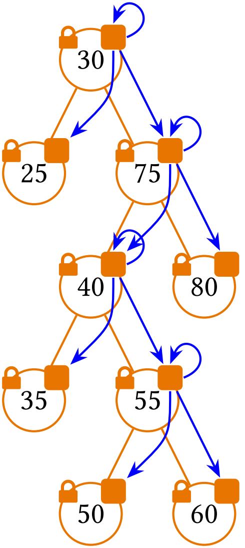
$\quad |\ \text{Some } (k, v, g_l, g_r) \Rightarrow k \in (i,j) \land$

$\qquad \exists p_l, p_r.\ p.\texttt{t} \mapsto (k, v, p_l, p_r) *$

$\qquad p_l.\texttt{lock} \boxdot\!\!\longmapsto_{\pi_2} \blacksquare_{g_l} *$

$\qquad p_r.\texttt{lock} \boxdot\!\!\longmapsto_{\pi_2} \blacksquare_{g_r}$

Recursive Lock Invariant



Lock Invariant: $\blacksquare_g(p)$

$$\blacksquare_g(p) \triangleq \exists i\, j\, c.\ 👻^{.5}_{g}\ ((i, j), c) *$$

$$p.\mathtt{lock} \mapsto_{\pi_1} \blacksquare_g *$$

$$\text{match } c \text{ with}$$

$$\mid \text{None} \Rightarrow p.\mathtt{t} = \mathtt{NULL}$$

$$\mid \text{Some } (k, v, g_l, g_r) \Rightarrow k \in (i, j) \wedge$$

$$\exists p_l, p_r.\ p.\mathtt{t} \mapsto (k, v, p_l, p_r) *$$

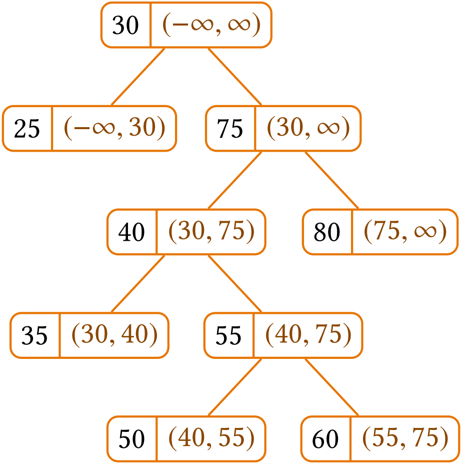
$$p_l.\mathtt{lock} \mapsto_{\pi_2} \blacksquare_{g_l} *$$

$$p_r.\mathtt{lock} \mapsto_{\pi_2} \blacksquare_{g_r}$$

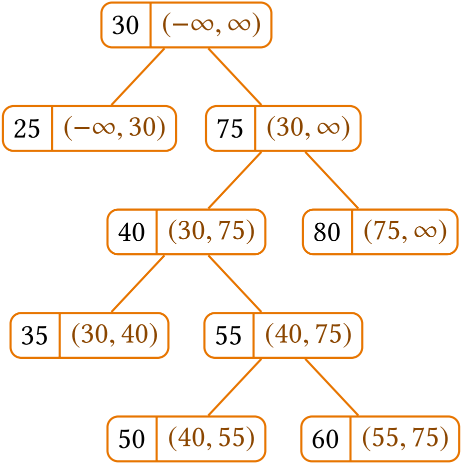
Per-Thread Handle to the BST

$$\text{bst\_ref}_g\ b \triangleq \exists p.\ b \mapsto p * p.\mathtt{lock} \mapsto \blacksquare_g(p)$$

Global Ghost State: $\mathsf{bst\_abs}_g$



Global Ghost State: $\mathrm{bst\_abs}_g$



$\square_g\ (t, (i, j)) \triangleq$

match t with

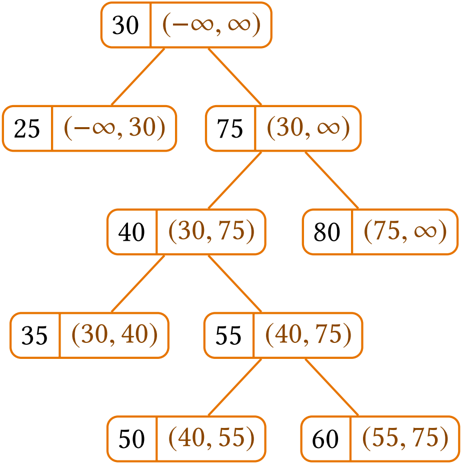
| Leaf $\Rightarrow$ $\text{👻}_g^{.5}\ ((i, j), \mathrm{None})$

| Node $(k, v, t_l, g_l, t_r, g_r) \Rightarrow$

$\text{👻}_g^{.5}\ ((i, j), \mathrm{Some}\ (k, v, g_l, g_r)) \ast$

$\square_{g_l}\ (t_l, (i, k)) \ast \square_{g_r}\ (t_r, (k, j))$

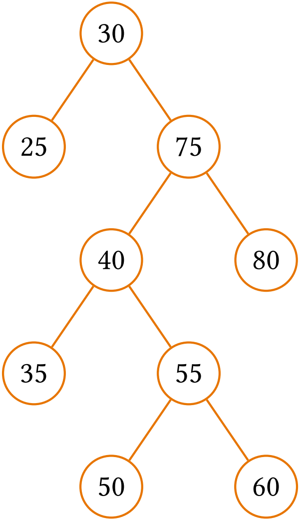
Global Ghost State: $\mathrm{bst\_abs}_g$



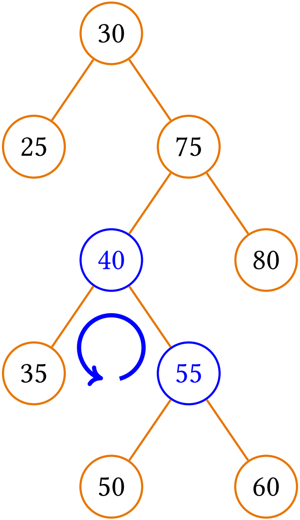
$\square_g\,(t, (i, j)) \triangleq$
match t with
| Leaf $\Rightarrow$ 👻$^{.5}_{g}\,((i, j), \mathrm{None})$
| Node $(k, v, t_l, g_l, t_r, g_r) \Rightarrow$
👻$^{.5}_{g}\,((i, j), \mathrm{Some}\,(k, v, g_l, g_r)) \ast$
$\square_{g_l}\,(t_l, (i, k)) \ast \square_{g_r}\,(t_r, (k, j))$

$\mathrm{bst\_abs}_g\ m \triangleq\ \exists t.\ t\ \mathrm{impl}\ m \wedge$
$\square_g\,(t, (-\infty, +\infty)) \ast$
$\mathrm{ghost\_nodes}\,(\mathrm{ids}(t))$

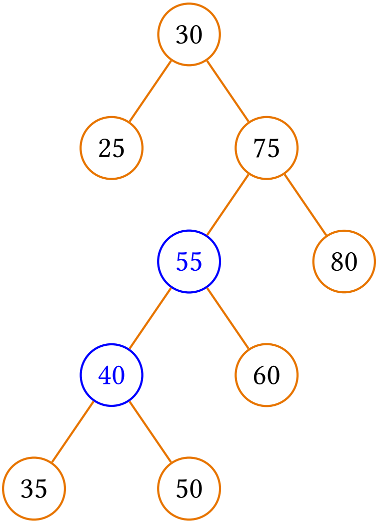
BST Rotation



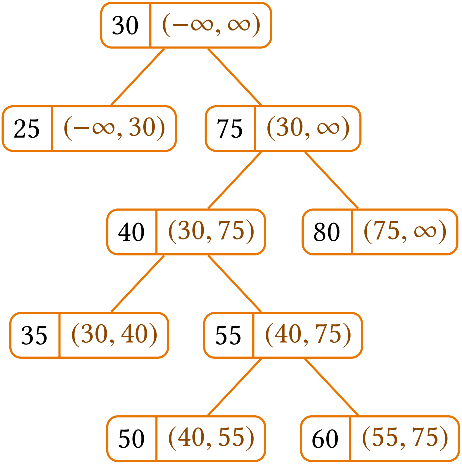
BST Rotation



BST Rotation



Global Ghost State: $\mathrm{bst\_abs}_g$



$\square_g\,(t, (i, j)) \triangleq$
match t with
$\mid \mathsf{Leaf} \Rightarrow \text{👻}^{.5}_{g}\,((i, j), \mathsf{None})$
$\mid \mathsf{Node}\,(k, v, t_l, g_l, t_r, g_r) \Rightarrow$
$\text{👻}^{.5}_{g}\,((i, j), \mathsf{Some}\,(k, v, g_l, g_r)) \ast$
$\square_{g_l}\,(t_l, (i, k)) \ast \square_{g_r}\,(t_r, (k, j))$

$\mathrm{bst\_abs}_g\ m \triangleq\ \exists t.\ t \text{ impl } m \wedge$
$\square_g\,(t, (-\infty, +\infty)) \ast$
$\mathrm{ghost\_nodes}\,(\mathrm{ids}(t))$

Concurrent Binary Search Tree (BST) as an Example

Atomic Specifications

$$\langle m.\,\mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ m\rangle\ \texttt{insert}(p, k, v)\ \langle \mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ (m[k \mapsto v])\rangle$$

$$\langle m.\,\mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ m\rangle\ \texttt{lookup}(p, k)\ \langle v.\,\mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ m \wedge m(k) = v\rangle$$

$$\langle m.\,\mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ m\rangle\ \texttt{delete}(p, k)\ \langle \mathrm{bst\_ref}\ p \mid \mathrm{bst\_abs}\ (m[k \mapsto \_])\rangle$$

Comparison with TaDA-Style Specifications

- The key elements of the proof are:

Comparison with TaDA-Style Specifications

- The key elements of the proof are:
  - Per-node ghost state for each lock invariant, and global ghost state that summarizes all the per-node state

Comparison with TaDA-Style Specifications

- The key elements of the proof are:
  - Per-node ghost state for each lock invariant, and global ghost state that summarizes all the per-node state
  - Complex share accounting for hand-over-hand locking: each lock invariant needs to contain a share of itself and shares of its child locks

Comparison with TaDA-Style Specifications

- The key elements of the proof are:
  - Per-node ghost state for each lock invariant, and global ghost state that summarizes all the per-node state
  - Complex share accounting for hand-over-hand locking: each lock invariant needs to contain a share of itself and shares of its child locks
- Compare to the template proofs for hand-over-hand locking, which use TaDA-style lock specs:

Comparison with TaDA-Style Specifications

- The key elements of the proof are:
  - Per-node ghost state for each lock invariant, and global ghost state that summarizes all the per-node state
  - Complex share accounting for hand-over-hand locking: each lock invariant needs to contain a share of itself and shares of its child locks
- Compare to the template proofs for hand-over-hand locking, which use TaDA-style lock specs:
  - Still need per-node ghost state and connection to global ghost state

Comparison with TaDA-Style Specifications

- The key elements of the proof are:
  - Per-node ghost state for each lock invariant, and global ghost state that summarizes all the per-node state
  - Complex share accounting for hand-over-hand locking: each lock invariant needs to contain a share of itself and shares of its child locks
- Compare to the template proofs for hand-over-hand locking, which use TaDA-style lock specs:
  - Still need per-node ghost state and connection to global ghost state
  - No share accounting: locks are part of the global ghost state and are accessed atomically, never owned by any thread

Conclusion

- Invariant-based specs can still be used to prove logically atomic specs for *fine-grained* locking, with the same approach to ghost state as has been used in the TaDA style.

Conclusion

- Invariant-based specs can still be used to prove logically atomic specs for *fine-grained* locking, with the same approach to ghost state as has been used in the TaDA style.
- It is more complex than using TaDA-style atomic lock specs: per-node ghost state, complex share accounting

Conclusion

- Invariant-based specs can still be used to prove logically atomic specs for *fine-grained* locking, with the same approach to ghost state as has been used in the TaDA style.
- It is more complex than using TaDA-style atomic lock specs: per-node ghost state, complex share accounting
- There might be systems where the old lock specs are necessary (e.g. in VST they were part of the soundness proof), so it's good to know that we can still prove atomic specs for data structures.

Conclusion

- Invariant-based specs can still be used to prove logically atomic specs for *fine-grained* locking, with the same approach to ghost state as has been used in the TaDA style.
- It is more complex than using TaDA-style atomic lock specs: per-node ghost state, complex share accounting
- There might be systems where the old lock specs are necessary (e.g. in VST they were part of the soundness proof), so it's good to know that we can still prove atomic specs for data structures.
- We have just modified VST to support TaDA-style lock specs instead, and are looking forward to simpler atomicity proofs for fine-grained C programs.