A separation logic for refining concurrent objects

Aaron Turon and Mitchell Wand
Northeastern University
Parallel hardware, concurrent algorithms

Contributions from PL/systems:

STM, message passing, work stealing
Parallel hardware, concurrent algorithms

Contributions from PL/systems:

STM, message passing, work stealing

Enabling technology:

fine-grained concurrent data structures
Parallel hardware, concurrent algorithms

Contributions from PL/systems:

  STM, message passing, work stealing

Enabling technology:

  fine-grained concurrent data structures

This talk: study, verify such algorithms with PL tools
int PostInc(int *c) {
    return (*c)++;
}

Want:
    thread safety
    scalability
int PostInc(int *c) {
    return (*c)++;
}

Want:
    thread safety
    scalability

Locks:
    thread safety ✓
    scalability
Hardware-level atomicity

CAS(x, old, upd)

Atomically:
• if \( *x == \text{old} \), sets \( *x = \text{upd} \), returns true
• otherwise, returns false

CAS resolves races at instruction level
int CASPostInc(int *c) {
    int old;
    do {
        old = *c;
    } while (!CAS(c, old, old+1));
    return old;
}

int CASPostInc(int *c) {
    int old;
    do {
        old = *c;
    } while (!CAS(c, old, old+1));
    return old;
}

int AtomicPostInc(int *c) {
    "atomic" {
        return (*c)++;
    }
}
CASPostInc versus AtomicInc

Herlihy & Wing '90: Linearizability

PL perspective: Refinement

\[ P \text{ refines } Q \iff \forall C. C[P] \text{ halts } \Rightarrow C[Q] \text{ halts} \]

Filipović et al. '09: Linearizability => refinement
Our logic takes observable atomicity seriously!
Our logic takes observable atomicity seriously!

Our contributions:

- A specification language that can assert atomicity
- A model theory that captures atomicity and ownership
- A proof theory that connects:
  - concurrent separation logic to refinement
  - rely/guarantee to data abstraction
Our logic takes observable atomicity seriously!

**Our contributions:**

- A specification language that can assert atomicity
- **A model theory that captures atomicity and ownership**
- A proof theory that connects:
  - concurrent separation logic to refinement
  - rely/guarantee to data abstraction
By making **atomicity** and **ownership** explicit, we can clarify and exploit their relationship:

**observed atomicity** is relative to **ownership**
By making **atomicity** and **ownership** explicit, we can clarify and exploit their relationship:

**observed atomicity is relative to ownership**
Question 1: What is atomicity?
What is atomicity?
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

\[(s, s') (s'', s''')\]
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

(s , s' ) ( s'' , s''')

Program   Program   Environment
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

( s, s' )( s'', s'''' )...
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

\[(s, s')(s'', s''')...\]
\[(s, s')(s', s')(s'', s''')...\]
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

$$(s, s')(s'', s''')...$$

$$(s, s')(s', s')(s'', s''')...$$

**Thm (Brookes):** $P$ refines $Q$ iff $\text{traces}(P) \subseteq \text{traces}(Q)$
What is atomicity?

Brookes '93: full abstraction for shared-memory using “transition traces”

\[(s, s')(s'', s''')... \]
\[(s, s')(s', s')(s'', s''')... \]

Thm (Brookes): P refines Q iff \( \text{traces}(P) \subseteq \text{traces}(Q) \)

Def: P atomic iff \( \text{traces}(P) \subseteq \text{id}^* ; S \times S ; \text{id}^* \)
Question 2: What is ownership?
What is ownership?
What is ownership?

```c
int Inc(int *c) {
    int old;

    do {
        old = *c;
    } while (!CAS(c, old, old+1));

    return old;
}
```
What is ownership?

```c
int Inc(int *c) {
    int old;

    do {
        old = *c;
    } while (!CAS(c, old, old+1));

    return old;
}
```
What is ownership?

```c
int Inc(int *c) {
    int old;

    do {
        old = *c;
    } while (!CAS(c, old, old+1));

    return old;
}
```
What is ownership?

```c
int Inc(int *c) {
    int old;
    do {
        old = *c;
    } while (!CAS(c, old, old+1));
    return old;
}
```
Our ownership discipline
Our ownership discipline

\[
\begin{align*}
\{ & 3, 5, 4 \} & \text{Data structure instances}
\end{align*}
\]
Our ownership discipline

Data structure instances

Call to Inc
Our ownership discipline

Data structure instances

Method-private data
Our ownership discipline

Call to Inc

Data structure instances

Method-private data
Our ownership discipline

Call to Inc

- Data structure instances
  - Method-private data
Our ownership discipline

Call to Inc

Data structure instances

Method-private data

Client data
Our ownership discipline

Data abstraction + concurrency:

• Instance data:
  shared by concurrent methods
  hidden from client

• Method-private data interference free
Our ownership discipline

**Dynamic** boundary between private/shared data (“in the eye of the asserter”)

Use separation logic “fence”:

\[ c \rightarrow - \quad \text{for counter} \]

\[ \exists x. s \rightarrow z * \text{List}(x) \quad \text{for stack as linked list} \]
Our ownership discipline

**Dynamic** boundary between private/shared data
(“in the eye of the asserter”)

Use separation logic “fence”:

\[ c \rightarrow - \quad \text{for counter} \]

\[ \text{exists } x. \ s \rightarrow z \ast \text{List}(x) \quad \text{for stack as linked list} \]
\[ \text{where } \text{List}(x) = \]
\[ (x = 0 \text{ or} \]
\[ \text{exists } y. \ x \rightarrow < -, y> \ast \text{List}(y)) \]
Question 3: How do atomicity and ownership relate?
How do atomicity and ownership relate?
How do atomicity and ownership relate?

```c
int Inc(int *c) {
    int old;

    do {
       old = *c;
    } while (!CAS(c, old, old+1));

    return old;
}
```
How do atomicity and ownership relate?

```c
int Inc(int *c) {
    int old;

    do {
        old = *c;
    } while (!CAS(c, old, old+1));

    return old;
}
```
do {
    old = *c;
} while (!CAS(c, old, old+1));
do {
    old = *c;
} while (!CAS(c, old, old+1));

\[
\begin{align*}
\text{snapshot} & \quad \text{CAS} \\
\begin{pmatrix}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4
\end{pmatrix} & \begin{pmatrix}
\text{old} = 4 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 5
\end{pmatrix}
\end{align*}
\]
do {
    old = *c;
} while (!CAS(c, old, old+1));

snapshot
  (old = 0, old = 4)
  (c → 4, c → 4)

CAS
  (old = 4, old = 4)
  (c → 4, c → 5)

snapshot
  (old = 0, old = 4)
  (c → 4, c → 4)

failed CAS
  (old = 7, old = 7)
  (c → 4, c → 4)
do {
    old = *c;
} while (!CAS(c, old, old+1));

Valid, but not atomic

( old = 0, old = 4 ) ( old = 4, old = 4 )
(c → 4, c → 4) (c → 4, c → 5)

Invalid

( old = 0, old = 4 ) ( old = 7, old = 7 )
(c → 4, c → 4) (c → 4, c → 4) ...

old = 4
c → 4
old = 4
c → 4
Original trace: valid but not atomic

\[
\left( \begin{array}{cc}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4 \\
\end{array} \right)
\left( \begin{array}{cc}
\text{old} = 4 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 5 \\
\end{array} \right)
\]
Original trace: valid but not atomic

\[
\left( \begin{array}{cc}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4
\end{array} \right) \left( \begin{array}{cc}
\text{old} = 4 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 5
\end{array} \right)
\]
Original trace: valid but not atomic

\[
\begin{align*}
\left( \begin{array}{c}
\text{old} = 0 \\
\text{c} \rightarrow 4
\end{array} \right)
& \quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 4
\end{array} \right)
\quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 4
\end{array} \right)
\quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 5
\end{array} \right) \\
\left( \begin{array}{c}
\text{old} = 0 \\
\text{c} \rightarrow 4
\end{array} \right)
& \quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 4
\end{array} \right)
\quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 4
\end{array} \right)
\quad \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \rightarrow 5
\end{array} \right)
\end{align*}
\]
Original trace: valid but not atomic

\[
\left( \begin{array}{cc}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4
\end{array} \right) \right)
\left( \begin{array}{cc}
\text{old} = 4 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 5
\end{array} \right)
\]

Fenced trace

\(< \text{old} = 0, (\text{c} \rightarrow 4, \text{c} \rightarrow 4)(\text{c} \rightarrow 4, \text{c} \rightarrow 5), \text{old} = 4 >\)
Original trace: valid but not atomic

\[
\begin{pmatrix}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4 \\
\end{pmatrix}
\begin{pmatrix}
\text{old} = 4 \\
\text{c} \rightarrow 4 \\
\end{pmatrix}
\begin{pmatrix}
\text{old} = 4 \\
\text{c} \rightarrow 5 \\
\end{pmatrix}
\]

**Fenced** trace

\(< \text{old} = 0, (\text{c} \rightarrow 4, \text{c} \rightarrow 4) (\text{c} \rightarrow 4, \text{c} \rightarrow 5), \text{old} = 4 >\)
Original trace: valid but not atomic

\[
\left( \begin{array}{ccc}
\text{old} = 0 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 4
\end{array} \right)
\left( \begin{array}{ccc}
\text{old} = 4 & \text{old} = 4 \\
\text{c} \rightarrow 4 & \text{c} \rightarrow 5
\end{array} \right)
\]

Fenced trace

\(< \text{old} = 0, (\text{c} \rightarrow 4, \text{c} \rightarrow 5), \text{old} = 4 > \quad \text{Atomic!}\>
Second trace: intuitively invalid

\[
\left( \begin{array}{c}
\text{old} = 0 \\
\text{c} \to 4
\end{array} \right) \left( \begin{array}{c}
\text{old} = 4 \\
\text{c} \to 4
\end{array} \right) \left( \begin{array}{c}
\text{old} = 7 \\
\text{c} \to 4
\end{array} \right) \left( \begin{array}{c}
\text{old} = 7 \\
\text{c} \to 4
\end{array} \right) \ldots
\]
Second trace: intuitively invalid

\[
\begin{align*}
\left( \begin{array}{c}
\text{old} = 0 \\
\text{old} = 4
\end{array} \right) & \left( \begin{array}{c}
\text{c} \rightarrow 4 \\
\text{c} \rightarrow 4
\end{array} \right) \left( \begin{array}{c}
\text{old} = 7 \\
\text{old} = 7
\end{array} \right)
\end{align*}
\ldots
\]
Second trace: intuitively invalid

No corresponding fenced trace: trace is \textit{formally} invalid
Fenced traces: hybrid of transition traces and IO relation

\[ \text{Store} \times (\text{Store} \times \text{Store})^* \times \text{Store} \]

Starting private state

Shared state trace

Ending private state
Fenced traces:
    hybrid of transition traces and IO relation

    $\text{Store} \times (\text{Store} \times \text{Store})^* \times \text{Store}$

Fenced projection:
    takes fence, set of traces
    produces set of fenced traces
    discards “invalid” traces
Key Theorem

\[ \text{FTraces}(D_1) \subseteq \text{Ftraces}(D_2) \implies D_1 \text{ refines } D_2 \]

where \( D_1, D_2 \) are data structures
CASPostInc and AtomicPostInc are *contextually equivalent* (as shown by their fenced traces)
CASPostInc and AtomicPostInc are contextually equivalent

(as shown by their fenced traces)

Observed atomicity is relative to ownership
Rely/guarantee

Complex examples need interference assumptions

Rely/guarantee through fenced traces:
  dispose of ill-behaved traces

Rely/guarantee reasoning scoped by data abstraction
Related work

Separation logic

uses ownership to constrain interference, no direct connection to atomicity

Calculus of atomic actions, and Groves's work not “refinement” in our sense; no ownership
Our logic takes observable atomicity seriously!

Our contributions:

- A **specification language** that can assert atomicity
- A **model theory** that captures atomicity and ownership
- A **proof theory** that connects:
  - concurrent separation logic to refinement
  - rely/guarantee to data abstraction
- More examples in the paper
Our logic takes observable atomicity seriously!

Our contributions:

- A **specification language** that can assert atomicity
- A **model theory** that captures atomicity and ownership
- A **proof theory** that connects:
  - concurrent separation logic to refinement
  - rely/guarantee to data abstraction
- More examples in the paper

Thank you