Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman
This lecture

- The problems of concurrency
- Disjoint concurrency
- Concurrent separation logic
Concurrency

Concurrent:

“Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint”

- Oxford English Dictionary
Programming language

C ::= ... | C || C | ...
Motivation

• Concurrency is hard:

“If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug.”

Java Sun Tutorial “Threads and Swing”

• Multi-core means concurrency everywhere!
"Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors."

JavaOne Technical session
Testing is hard

“Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors.”

JavaOne Technical session

Verification to the rescue?
Verifying concurrent programs is hard

Have to consider all possible interleavings:
Thread-modular reasoning

- Considers every thread in isolation under some assumption on its environment:
Thread-modular reasoning

- Considers every thread in isolation under some assumption on its environment:

\[ T_1 \quad T_1 \quad T_1 \quad T_2 \]

Captures possible interference from the other threads...
Thread-modular reasoning

• Considers every thread in isolation under some assumption on its environment:

• Avoids direct reasoning about all interleavings
Disjoint Concurrency
Disjoint concurrency

- Language with parallel composition: $C_1 \parallel C_2$
- Every thread operates on its own part of the heap:
Parallel proof rule

\[
\begin{array}{c}
\{P_1\} \quad C_1 \quad \{Q_1\} \\
\{P_2\} \quad C_2 \quad \{Q_2\}
\hline
\{P_1 \ast P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 \ast Q_2\}
\end{array}
\]

variables used in \(C_1\), \(P_1\) and \(Q_1\) not modified by \(C_2\); variables used in \(C_2\), \(P_2\) and \(Q_2\) not modified by \(C_1\)
Parallel proof rule

\[
\{P_1\} \quad C_1 \quad \{Q_1\} \quad \{P_2\} \quad C_2 \quad \{Q_2\} \\
\{P_1 \ast P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 \ast Q_2\}
\]

variables used in C₁, P₁ and Q₁ not modified by C₂; variables used in C₂, P₂ and Q₂ not modified by C₁
Parallel proof rule

\[
\begin{array}{c}
\{P_1\} \quad C_1 \quad \{Q_1\} \\
\{P_1 \ast P_2\} \quad C_1 \parallel C_2 \quad \{Q_1 \ast Q_2\}
\end{array}
\]

variables used in $C_1$, $P_1$ and $Q_1$ not modified by $C_2$;
variables used in $C_2$, $P_2$ and $Q_2$ not modified by $C_1$
Parallel proof rule

\[
\begin{align*}
\{ P_1 \} & \quad C_1 \quad \{ Q_1 \} & \quad \{ P_2 \} & \quad C_2 \quad \{ Q_2 \} \\
\{ P_1 \ast P_2 \} & \quad C_1 \ | \ | \quad C_2 \quad \{ Q_1 \ast Q_2 \}
\end{align*}
\]

variables used in \( C_1, P_1 \) and \( Q_1 \) not modified by \( C_2 \);
variables used in \( C_2, P_2 \) and \( Q_2 \) not modified by \( C_1 \)

- Remember semantics of triples: \( C_1 \) accesses only the memory in \( P_1 \) and the one it allocates itself
- No way to mess up the heap owned by \( C_2 \)!
Example

\{ x \mapsto _ * y \mapsto _ \} \\
\{ x \mapsto _ \} \quad \{ y \mapsto _ \} \\
[x] := 3 \quad || \quad [y] := 4 \\
\{ x \mapsto 3 \} \quad \{ y \mapsto 4 \} \\
\{ x \mapsto 3 * y \mapsto 4 \}
Parallel Dispose tree

```c
struct Tree {
    Tree *Left;
    Tree *Right;
}

disposetree(Tree *x) {
    if (x != NULL) {
        i = x->Left;  
        j = x->Right;
        (disposetree(i) || disposetree(j) || free(x));
    }
}
```
struct Tree {
    Tree *Left;
    Tree *Right;
}

Tree(x) = (x = NULL ∧ emp) ∨
(∃i, j. x → i, j ∗ Tree(i) ∗ Tree(j))
Parallel Dispose tree

\[
\begin{align*}
\text{Tree}(x) &= (x = \text{NULL} \land \text{emp}) \lor \\
&\quad (\exists i, j. \ x \mapsto i, j \ast \text{Tree}(i) \ast \text{Tree}(j)) \\

\{ \text{tree}(x) \land x \neq \text{NULL} \} \\
&\quad \{ \exists i, j. \ \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \} \\
i &= x\rightarrow\text{Left}; \\
&\quad \{ \exists j. \ \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \} \\
j &= x\rightarrow\text{Right}; \\
&\quad \{ \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \} \\
(\text{disposetree}(i) \lor \text{disposetree}(j) \lor \text{free}(x)); \\
&\quad \{ \text{emp} \}
\end{align*}
\]
Example

\[
\begin{align*}
&\{ \text{tree}(i) \} \times \{ \text{tree}(j) \} \times \{ x \mapsto i,j \} \\
&\text{disposetree}(i) \parallel \text{disposetree}(j) \parallel \text{dispose } x \\
&\{ \text{emp} \} \times \{ \text{emp} \} \times \{ \text{emp} \} \\
&\{ \text{emp} \times \text{emp} \times \text{emp} \} \\
&\{ \text{emp} \}
\end{align*}
\]
Can we verify these?

\{ \text{ emp } \}
\begin{align*}
x & := \text{ new}; \\
z & := \text{ new}; \\
[x] & := 4 \ || \ [z] := 5; \\
\{ x \rightarrow 4 \ast z \rightarrow 5 \}
\end{align*}

\{ \text{ emp } \}
\begin{align*}
x & := \text{ new}; \\
[x] & := 4 \ || \ [x] := 5; \\
\{ x \rightarrow \} 
\end{align*}

\{ \text{ emp } \}
\begin{align*}
x & := 4 \ || \ x := 5; \\
\{ \text{ emp } \}
\end{align*}

\{ \text{ emp } \}
\begin{align*}
y & = x + 1 \\
x & := 4 \ || \ y := y + 1; \\
\{ y = x + 2 \}
\end{align*}

Thursday, 3 March 2011
Merge sort

mergesort(x, n)
  if n > 1 then
    local m in
    m := n/2;
    mergesort(x, m) || mergesort(x+m, n-m);
    merge(x, m, n-m)
Merge sort

\[
\begin{align*}
\{ \text{array}(x,n) \} \\
\text{mergesort}(x, n) \\
\{ \text{sorted_array}(x,n) \} \\
\{ \text{sorted_array}(x,m) \ast \text{sorted_array}(x+m,n) \} \\
\text{merge}(x,m,n) \\
\{ \text{sorted_array}(x,m+n) \}
\end{align*}
\]
Merge sort

{ array(x,n) }
mergesort(x, n)
  if n > 1 then
    local m in
    m := n/2;
    mergesort(x,m) || mergesort(x+m,n-m);
  merge(x,m,n-m)

{ sorted_array(x,n) }
Concurrent Separation Logic
Multiple access

How do we verify a program where several threads want access to the same memory? e.g.

\[ x := 43 \parallel x := 47 \]

We protect shared values with locks
Multiple access

How do we verify a program where several threads want access to the same memory? e.g.

\[ x := 43 \quad || \quad x := 47 \]

We protect shared values with locks
Reasoning principle

Separation property: at any time, the state of the program can be partitioned into that owned by each thread and each free lock

\[ \text{Diagram:} \]

- \( T_1 \)
- \( r1 \)
- \( r2 \)
- \( T_2 \)
Reasoning principle

*Separation property:* at any time, the state of the program can be partitioned into that owned by each thread and each free lock

Assign a resource invariant $I$ to every lock $r$: describes the part of the heap protected by the lock
Programming language

C ::= … | resource r in C | with r when B in C | …
Resource Rule

\[ \Delta, r : I \vdash \{ P \} C \{ Q \} \]

\[ \Delta \vdash \{ P \ast I \} \text{ resource } r \text{ in } C \{ Q \ast I \} \]
Lock Rule

\[ \Delta \vdash \{ (P \ast I) \land B \} \quad \text{C} \quad \{ Q \ast I \} \]

\[ \Delta, r : I \vdash \{ P \} \text{ with } r \text{ when } B \text{ in } C \quad \{ Q \} \]
Caveat: side-conditions

There are subtle variable side-conditions used to allow locks to refer to global variables.

Each variable is either associate to

- a single thread; or
- a single lock.

It can then only be modified and used in assertions by the thread, or while the thread holds the associate lock.
Binary Semaphore

We can encode a semaphore as a critical region

\[
\begin{align*}
P(s) &= \text{with } r_s \text{ when } s=1 \text{ do } s := 0 \\
V(s) &= \text{with } r_s \text{ when } s=0 \text{ do } s := 1
\end{align*}
\]

Resource invariant

\[
(s=0 \land \text{emp}) \lor (s=1 \land Q)
\]

Initially,

\[s=0\]
Example

\{ \text{emp} \} 
P(s)
\[x] := 43
V(s)
\{ \text{emp} \}

\{ \text{emp} \} 
P(s)
\[x] := 47
V(s)
\{ \text{emp} \}
Example

\{\text{emp}\}
\text{P}(s)
\{x \mapsto \_\}
[x] := 43
\{x \mapsto \_\}
\text{V}(s)
\{\text{emp}\}
Example

\{ \text{emp} \}

P(s)
\{ x \mapsto _ \}

[x] := 43
\{ x \mapsto _ \}

V(s)
\{ \text{emp} \}

\{ \text{emp} \ast (l_s \land s=1) \} s := 0 \{ x \mapsto _ \ast l_s \} \cdot

\{ \text{emp} \} \text{ with } r_s \text{ when } s=1 \text{ do } s:=0 \{ x \mapsto _ \}
Example

P(s)
{ emp } with r_s when s=1 do s:=0 { x ↦ _ }
{ emp } with r_s when s=0 do s:=1 { emp }

V(s)
{ emp } with r_s when s=1 do s:=0 { x ↦ _ }
{ emp } with r_s when s=0 do s:=1 { emp }

{ x ↦ _ } := 43

{ x ↦ _ }
More Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman
This lecture

• Recap: CSL
• Ownership
• Precision
• Read-sharing
• Auxiliary state
Programming language

C ::= … | resource r in C | with r when B in C | …
Resource sharing

\[
\begin{align*}
&\text{with } l \text{ when true do} \\
&[x] := 56 \\
\end{align*}
\]

\[
\begin{align*}
&\text{with } l \text{ when true do} \\
&[x] := 42 \\
\end{align*}
\]
Reasoning principle

**Separation property**: at any time, the state of the program can be partitioned into that owned by each thread and each free lock.

Assign a **resource invariant** $I$ to every lock $r$: describes the part of the heap protected by the lock.
Parallel proof rule

\[
\begin{array}{ccc}
\{P_1\} & C_1 & \{Q_1\} \\
\{P_2\} & C_2 & \{Q_2\} \\
\{P_1 \ast P_2\} & C_1 & \parallel C_2 & \{Q_1 \ast Q_2\}
\end{array}
\]
Resource Rule

\[ \Delta, r : I \vdash \{ P \} C \{ Q \} \]

\[ \Delta \vdash \{ P * I \} \text{ resource } r \text{ in } C \{ Q * I \} \]
Lock Rule

\[ \Delta \vdash \{ (P \ast I) \land B \} C \{ Q \ast I \} \]

\[ \Delta, r : I \vdash \{ P \} \text{ with } r \text{ when } B \text{ in } C \{ Q \} \]
Ownership
One place buffer

full := false

with buff when full do
  full := false
  y := c
  dispose y

x := new
with buff when ¬full do
  full := true;
  c := x;
One place buffer

$$\text{full} := \text{false}$$

$$\{ (\text{full} \land c \mapsto _) \lor (\neg \text{full} \land \text{emp}) \}$$

with buff when full do
  full := false
  y := c
  dispose y

$\mid$

x := new
with buff when ¬full do
  full := true;
  c := x;
\[
\{ (\text{full} \land c \mapsto _) \lor (\neg\text{full} \land \text{emp}) \}
\]

with buff when full do

\[
\{ \text{full} \land c \mapsto _ \}
\]

\text{full} := \text{false}

\[
\{ \neg\text{full} \land \text{emp} \}
\]

\* \text{y} \mapsto _

\{ \text{y} \mapsto _ \}

dispose \text{y}

\{ \text{emp} \}

\[
x := \text{new}
\]

\[
\{ x \mapsto _ \}
\]

with buff when \neg\text{full} do

\[
\{ (\neg\text{full} \land \text{emp}) \}
\]

\* \text{x} \mapsto _

\{ \text{full} \land c \mapsto _ \}

\text{full} := \text{true};

\text{c} := \text{x};

\{ \text{full} \land c \mapsto _ \}
full := false

with buff when full do
  full := false
  y := c

| x := new
with buff when ¬full do
  full := true;
  c := x;
  dispose x

Can we verify this?
full := false

{ emp }

with buff when full do
  full := false
  y := c

\[\begin{align*}
  x &:= \text{new} \\
  \text{with buff when } \neg \text{full do} \\
  &\quad \text{full := true;}
  \quad c := x;
  \quad \text{dispose } x
\end{align*}\]

Can we verify this?
with buff when full do
  { full \land emp }
  full := false
  y := c
  { \neg full \land emp \land y = c }
  { emp \land y = c }

with buff when \neg full do
  x := new
  { x \mapsto _ } 
  full := true;
  c := x;
  { full \land x \mapsto _ } 
  dispose x
  { emp }
Ownership is in the Eye of the Asserter

The important thing is that threads have compatible assumptions about each other.
Precision
Precise invariants

A formula, $P$, is *precise* iff:

for any heap, there is at most one subheap satisfying the formula

$$\forall h_1, h_2, h. \quad h_1 \preceq h \land h_2 \preceq h \land h_1 \nvDash P \land h_2 \nvDash P \Rightarrow h_1 = h_2$$
Precise predicates?

∀h₁, h₂, h.  h₁ ≤ h ∧ h₂ ≤ h ∧ h₁ ⊭ P ∧ h₂ ⊭ P ⇒ h₁ = h₂

emp

ture

emp ∨ (x → null * y → null)

x → null * true

∀h₁, h₂, h.  h₁ ≤ h ∧ h₂ ≤ h ∧ h₁ ⊭ P ∧ h₂ ⊭ P ⇒ h₁ = h₂

ls(E, F) ⇔ (E = F ∧ emp) ∨ (∃x'. E → x' * ls(x', F))
Rule of conjunction

\[
\{ P_1 \} \quad C \quad \{ Q_1 \} \\
\{ P_2 \} \quad C \quad \{ Q_2 \} \\
\hline
\{ P_1 \land P_2 \} \quad C \quad \{ Q_1 \land Q_2 \}
\]
Subtle soundness

Without rule of conjunction Concurrent Separation Logic is sounds with arbitrary resource invariants.

With precise invariants and the rule of conjunction the logic is sound.
Unsoundness of conjunction rule

Imprecise invariants allow different heap splittings at unlock
Unsoundness of conjunction rule

Imprecise invariants allow different heap splittings at unlock
Unsoundness of conjunction rule

Imprecise invariants allow different heap splittings at unlock
Unsoundness of conjunction rule

Imprecise invariants allow different heap splittings at unlock

$Q_1$ and $Q_2$ describe different parts of the heap, yet can be conjoined:

$$\begin{align*}
\{P\} & \ C \ \{Q_1\} \quad \{P\} \ C \ \{Q_2\} \\
\{P\} & \ C \ \{Q_1 \land Q_2\}
\end{align*}$$
Reynolds’ counter example

We can prove the following holds with the resource invariant \( r: \text{true} \)

\[
\{ \text{true} \} \text{ skip } \{ \text{true} \}.
\]

\[
\{(\text{emp} \lor x \mapsto \_ ) \ast \text{true} \} \text{ skip } \{ \text{emp} \ast \text{true} \}.
\]

\[
\{ \text{emp} \lor x \mapsto \_ \} \text{ with } r \text{ when } \text{true} \text{ do } \text{skip } \{ \text{emp} \}.
\]
Reynolds’ counter example

From the previous proof we can derive both

\{ \text{emp} \times x \mapsto \_ \} \text{ with… } \{ x \mapsto \_ \}

\{ \text{emp} \times x \mapsto \_ \} \text{ with… } \{ \text{emp} \}

which with the rule of conjunction leads to a contradiction.
Read Sharing
Races

*Data race* - a read and write, or two writes, to the same shared data at the same time.

Do we want to forbid all races?

Does concurrent separation logic forbid all races?
Read sharing

{ emp }

x = new

t1 = [x] || t2 = [x]

free(x)

{ emp }

{ emp }
Read sharing

Replace $\rightarrow$ with $\xrightarrow{\pi}$, where $\pi \in (0;1]$ is a permission.

- $\pi < 1$ allows only read access
- $\pi = 1$ allows read and write access
Read sharing

Replace $\mapsto$ with $\xrightarrow{\pi}$, where $\pi \in (0;1]$ is a permission

- $\pi < 1$ allows only read access
- $\pi = 1$ allows read and write access

\[
x^{\pi_1 + \pi_2} \mapsto E \iff x^{\pi_1} \mapsto E \ast x^{\pi_2} \mapsto E
\]

\[
x \mapsto E \iff x \mapsto E
\]
\{ \text{emp} \} \n
\texttt{x} = \texttt{new} \n
\texttt{tl} = [\texttt{x}] \quad \| \quad \texttt{t2} = [\texttt{x}] \n
\texttt{free(x)} \n
\{ \text{emp} \}
\[
\{ \text{emp} \}
\]
\[
x = \text{new}
\]
\[
\{ x \mapsto _ \}
\]
\[
\{ x \mapsto _ \} \times \{ x \mapsto _ \}
\]
\[
\{ x \mapsto _ \} \parallel \{ x \mapsto _ \}
\]
\[
t_1 = [x]
\]
\[
t_2 = [x]
\]
\[
\{ x \mapsto _ \} \parallel \{ x \mapsto _ \}
\]
\[
\{ x \mapsto _ \} \times \{ x \mapsto _ \}
\]
\[
\text{free}(x)
\]
\[
\{ \text{emp} \}
\]
Auxiliary State
Doing Something Twice

\[
\begin{align*}
\{ x \mapsto 0 \} \\
\text{resource } r \text{ in} \\
\left( \begin{array}{c}
\text{with } r \text{ when true} \\
[ x ] := [ x ] + 1 ; \\
\end{array} \right) \\
\text{with } r \text{ when true} \\
[ x ] := [ x ] + 1 ; \\
\{ x \mapsto 2 \}
\end{align*}
\]
Auxiliary State

The problem: the invariant hides the fact that there are just two threads.

We add auxiliary state to track this.

Can add extra state as long as it doesn’t affect the control-flow of the program.
resource r in

with r when true

[x] := [x] + 1

with r when true

[x] := [x] + 1
resource r in

with r when true
\[ x \] := \[ x \] + 1
\[ y \] := 1

with r when true
\[ x \] := \[ x \] + 1
\[ z \] := 1

Auxiliary assignment
\[
\{ \exists i, j. y \overset{0.5}{\rightarrow} i \cdot z \overset{0.5}{\rightarrow} j \cdot x \mapsto i + j \}
\]

resource \( r \) in

with \( r \) when true

\[
[x] := [x] + 1
\]

\[
[y] := 1
\]

with \( r \) when true

\[
[x] := [x] + 1
\]

\[
[z] := 1
\]

Resource Invariant

Auxiliary assignment
\[
\{ y \mapsto 0 \ast z \mapsto 0 \ast x \mapsto 0 \} \\
\text{resource } r \text{ in} \\
\{ y \mapsto 0 \ast z \mapsto 0 \}
\]

\[
\begin{align*}
    \text{with } r \text{ when true} & \quad \quad \quad \quad \quad \quad \quad \quad \text{with } r \text{ when true} \\
    [x] := [x] + 1 & \quad \quad \quad \quad \quad \quad \quad \quad [x] := [x] + 1 \\
    [y] := 1 & \quad \quad \quad \quad \quad \quad \quad \quad [z] := 1
\end{align*}
\]
\[
\{ y \mapsto 0 \ast z \mapsto 0 \ast x \mapsto 0 \} \\
\text{resource } r \text{ in} \\
\{ y \overset{0.5}{\mapsto} 0 \ast z \overset{0.5}{\mapsto} 0 \}
\]

with \( r \) when true

\[
\{ y \overset{0.5}{\mapsto} 0 \}
\]

with \( r \) when true

\[
\{ \exists j. \ y \mapsto 0 \ast z \overset{0.5}{\mapsto} j \ast x \mapsto j \} \\
[x] := [x] + 1 \\
[y] := 1 \\
\{ y \overset{0.5}{\mapsto} 1 \}
\]

\[
[x] := [x] + 1 \\
[z] := 1
\]
\{ y \mapsto 0 \} \quad \{ y \mapsto 0 \}

with r when true
\{ \exists j. y \mapsto 0 \} \quad \{ \exists i. y \mapsto i \}

[ x ] := [ x ] + 1
[ x ] := [ x ] + 1

\{ y \mapsto 0 \} \quad \{ z \mapsto 0 \}

resource r in
\{ y \mapsto 0 \} \quad \{ z \mapsto 0 \}

\{ z \mapsto 0 \}
\{ z \mapsto 0 \}
{(y \mapsto 0 \ast z \mapsto 0 \ast x \mapsto 0) \\
resource r in \\
{(y \mapsto 0) \\
with r when true \\
{(\exists j. y \mapsto 0 \ast z \mapsto j \ast x \mapsto j) \land \\
[x] := [x] + 1 \land \\
[y] := 1 \land \\
{y \mapsto 0}}) \\
| \\
{(z \mapsto 0) \\
with r when true \\
{(\exists i. y \mapsto i \ast z \mapsto 0 \ast x \mapsto i) \land \\
[x] := [x] + 1 \land \\
[z] := 1 \land \\
{z \mapsto 0})} \\
| \\
{(\exists i,j. y \mapsto i \ast z \mapsto j) \ast x \mapsto i+j \ast y \mapsto l \ast z \mapsto l)
Concurrent Separation Logic

References

• O’Hearn. Resources, concurrency and local reasoning. TCS, 2007

• Bornat et al. Permission accounting in separation logic. POPL’05