Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman
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 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
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:

\[ T_1 \rightarrow T_2 \]

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

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

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

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

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

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

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

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

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 \( C_1 \)

- 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 _ \} \ || \ { y \mapsto _ }  
[x] := 3 \ || \ [y] := 4  
\{ x \mapsto 3 \} \ || \ { y \mapsto 4 }  
\{ x \mapsto 3 \ * \ y \mapsto 4 \}
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) }  
disposetree(Tree *x) {  
    if (x != NULL) {  
        i = x->Left;  
        j = x->Right;  
        (disposetree(i) || disposetree(j) || free(x));  
    }  
} { emp }  

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

\[ \text{Tree}(x) = (x = \text{NULL} \land \text{emp}) \lor \\
(\exists i, j. x \mapsto i, j \ast \text{Tree}(i) \ast \text{Tree}(j)) \]

\{ \text{tree}(x) \land x \neq \text{NULL} \}
\{ \exists i, j. \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \}
\{ i = x->\text{Left}; \}
\{ \exists j. \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \}
\{ j = x->\text{Right}; \}
\{ \text{tree}(i) \ast \text{tree}(j) \ast x \mapsto i, j \}
(\text{disposetree}(i) || \text{disposetree}(j) || \text{free}(x));
\{ \text{emp} \}
Example

\{ \text{tree}(i) \,*\, \text{tree}(j) \,*\, x \mapsto i,j \} \\
\{ \text{tree}(i) \} \quad \{ \text{tree}(j) \} \quad \{ x \mapsto i,j \} \\
\text{dispose} \text{tree}(i) \, || \, \text{dispose} \text{tree}(j) \, || \, \text{dispose } x \\
\{ \text{emp} \} \quad \{ \text{emp} \} \quad \{ \text{emp} \} \\
\{ \text{emp} \,*\, \text{emp} \,*\, \text{emp} \} \\
\{ \text{emp} \}
Can we verify these?

{ emp }
\[ x := \text{new}; \]
\[ z := \text{new}; \]
\[ [x] := 4 \ || \ [z] := 5; \]
\[ \{ x \mapsto 4 \ *, \ z \mapsto 5 \} \]

{ emp }
\[ x := \text{new}; \]
\[ [x] := 4 \ || \ [x] := 5; \]
\[ \{ x \mapsto \_ \} \]

{ emp }
\[ x := 4 \ || \ x := 5; \]
\[ \{ \text{emp} \} \]

{ y = x+1 } 
\[ x := 4 \ || \ y := y+1; \]
\[ \{ y = x+2 \} \]
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

\{
array(x,n)
\}

mergesort(x, n)
\{
sorted_array(x,n)
\}

\{
sorted_array(x,m) \ast sorted_array(x+m,n)
\}

merge(x,m,n)
\{
sorted_array(x,m+n)
\}
Merge sort

\[
\begin{align*}
&\{\text{array}(x,n)\} \\
&\text{mergesort}(x, n) \\
&\quad \text{if } n > 1 \text{ then} \\
&\quad \quad \text{local m in} \\
&\quad \quad m := n/2; \\
&\quad \quad \text{mergesort}(x+m,n-m); \\
&\quad \quad \text{merge}(x,m,n-m) \\
&\{\text{sorted\_array}(x,n)\}
\end{align*}
\]
Concurrent Separation Logic
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 \textit{locks}
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
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

\[ T_1 \]
\[ r_1 \]
\[ T_2 \]
\[ r_2 \]
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

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

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

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

\[\Delta, r : I \vdash \{ P \} \text{ with } r \text{ when } B \text{ in } C \ \{ Q \}\]
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} \} \quad \begin{array}{c}
P(s) \\
x := 43 \\
V(s) \\
\{ \text{emp} \} \end{array} \quad \begin{array}{c}
P(s) \\
x := 47 \\
V(s) \\
\{ \text{emp} \} \end{array}
Example

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

\[
\begin{align*}
\{ \text{emp} \} \\
P(s) & \quad \{ x \mapsto \_ \} \\
[x] & := 43 \\
\{ x \mapsto \_ \} \\
V(s) & \quad \{ \text{emp} \} \\
\end{align*}
\]

\[
\{ \text{emp} * (l_s \land s=1) \} \quad s := 0 \quad \{ x \mapsto \_ * l_s \} \\
\{ \text{emp} \} \text{ with } r_s \text{ when } s=1 \text{ do } s:=0 \quad \{ x \mapsto \_ \}
\]
Example

\[
\begin{align*}
\{ \text{emp} \} \\
P(s) & \quad \{ x \mapsto \_ \} \\
[x] & := 43 \\
\{ x \mapsto \_ \} \\
V(s) & \quad \{ \text{emp} \}
\end{align*}
\]

\[
\begin{align*}
. & \quad \{ \text{emp} \ast (l_s \land s=1) \} \\& \quad s := 0 \quad \{ x \mapsto \_ \ast l_s \} \\
\{\text{emp}\} \text{ with } r_s \text{ when } s=1 \text{ do } s:=0 \quad \{ x \mapsto \_ \}
\end{align*}
\]

\[
\begin{align*}
. & \quad \{ x \mapsto \_ \ast (l_s \land s=0) \} \\& \quad s := 1 \quad \{ \text{emp} \ast l_s \} \\
\{x \mapsto \_\} \text{ with } r_s \text{ when } s=0 \text{ do } s:=1 \quad \{ \text{emp} \}
\end{align*}
\]
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

full := false

\{ (full \land c \mapsto _) \lor (\neg full \land emp) \}\n
with buff when full do
  full := false
  y := c
  dispose y

Resource
Invariant

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

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

\begin{align*}
\text{with buff when full do} \\
\{ \, \text{full} \land c \mapsto \_ \, \} \\
\text{full} := \text{false} \\
y := c \\
\{ \, (\neg \text{full} \land \text{emp}) \, \} \\
* y \mapsto \_ \\
\{ \, y \mapsto \_ \, \} \\
\text{dispose y}
\end{align*}

\begin{align*}
\text{x := new} \\
\{ \, x \mapsto \_ \, \} \\
\text{with buff when } \neg \text{full do} \\
\{ \, (\neg \text{full} \land \text{emp}) \, \} \\
* x \mapsto \_ \\
\text{full} := \text{true}; \\
c := x; \\
\{ \, \text{full} \land c \mapsto \_ \, \}
\end{align*}
Ownership is in the eye of the assertor

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 the following?
Ownership is in the eye of the assertor

full := false

\{ emp \}

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

Can we verify the following?

Resource Invariant

x := new
with buff when \neg\ full do
  full := true;
  c := x;
dispose x
next time:
Semantics