Relaxed separation logic

Viktor Vafeiadis       Chinmay Narayan
MPI-SWS               IIT Delhi
Goal: Understand concurrent programs.

Tool: Concurrent program logics:
- Concurrent Separation Logic
- OG, RG, RGSep, LRG, DG, CAP, CaReSL...

*** What about weak memory models? ***
All *sane* memory models satisfy the DRF property:

**Theorem (DRF-property)**

\[ \text{If} \ [Prg]_{SC} \text{ contains no data races, then} \]

\[ [Prg]_{\text{Relaxed}} = [Prg]_{SC}. \]

- Program logics that disallow data races are trivially sound.
- What about *racy* programs?
C11 operations

Two types of locations: ordinary and atomic
  - Races on ordinary accesses $\leadsto$ undefined

Several kinds of atomic accesses:
  - Sequentially consistent (reads & writes)
  - Release (writes)
  - Acquire (reads)
  - Relaxed (reads & writes)

A few more advanced constructs:
  - Fences, consume reads, ... (ignored here)
Execution = set of events & a few relations:
  - sb: sequenced before
  - rf: reads-from map
  - mo: memory order per location
  - sc: seq.consistency order
  - sw: synchronizes with (derived)

\[ W\text{-release} \xrightarrow{\text{rf}} R\text{-acq} \iff W\text{-release} \xrightarrow{\text{sw}} R\text{-acq} \]

- hb: happens before (derived, \( hb \overset{\text{def}}{=} (sb \cup sw)^+ \))

Axioms constraining the consistent executions.
Message passing example

\[
\begin{align*}
[a]_{na} & := 0; \\
[x]_{rlx} & := 0; \\
\left( [a]_{na} & := 10; \parallel \text{if } ([x]_{acq} = 1) \right) \\
[x]_{rel} & := 1; \parallel \text{print } [a]_{na}; \\
\end{align*}
\]

\[
W_{na}(a, 0) \\
\downarrow \text{sb} \\
W_{rlx}(x, 0) \\
\downarrow \text{sb} \\
W_{na}(a, 10) \\
\downarrow \text{sb} \\
W_{rel}(x, 1) \\
\downarrow \text{sb} \\
\rightarrow R_{acq}(x, 1) \\
\downarrow \text{sb} \\
\rightarrow R_{na}(a, 10) \\
\downarrow \text{sb}
\]
Separation logic recap

\[
[\ell \mapsto v] \overset{\text{def}}{=} \{ h \mid h(\ell) = v \}
\]

\[
[P_1 \cdot P_2] \overset{\text{def}}{=} \{ h_1 \uplus h_2 \mid h_1 \in [P_1] \land h_2 \in [P_2] \}
\]

Proof rules:

\[
\{ \ell \mapsto \_ \} [\ell] := v \{ \ell \mapsto v \} \quad \text{(WRI)}
\]

\[
\frac{\{ P \} C \{ Q \}} \quad \{ P \cdot R \} C \{ Q \cdot R \} \quad \text{(FRM)}
\]

\[
\frac{\{ P_1 \} C_1 \{ Q_1 \} \quad \{ P_2 \} C_2 \{ Q_2 \}} \quad \{ P_1 \cdot P_2 \} C_1 \parallel C_2 \{ Q_1 \cdot Q_2 \} \quad \text{(PAR)}
\]
Introduce two assertion forms:

\[ P ::= \ldots \mid \ell \xrightarrow{\text{rel}} Q \mid \ell \xrightarrow{\text{acq}} Q \]

where \( Q \in \text{Val} \rightarrow \text{Assn} \).

Initially (simplified rule):

\[
Q(v) = \text{emp} \\
\{\text{emp}\} x := \text{alloc}_{\text{atom}}(v) \{ x \xrightarrow{\text{rel}} Q \ast x \xrightarrow{\text{acq}} Q \}
\]
Read-acquire & write-release permissions (2/2)

- **Release writes:**
  \[
  \{ Q(v) \ast \ell \xrightarrow{\text{rel}} Q \} \ [\ell]_{\text{rel}} := v \ \{ \ell \xrightarrow{\text{rel}} Q \}
  \]

- **Acquire reads:**
  \[
  \{ \ell \xrightarrow{\text{acq}} Q \} \ \times := [\ell]_{\text{acq}} \ \{ Q(x) \ast \ell \xrightarrow{\text{acq}} Q[x:=\text{emp}] \}
  \]
  where \( Q[x:=P] \overset{\text{def}}{=} \lambda y. \text{if } x=y \text{ then } P \text{ else } Q(y) \).

- **Splitting permissions:**
  \[
  \ell \xrightarrow{\text{rel}} Q \ast \ell \xrightarrow{\text{rel}} Q \iff \ell \xrightarrow{\text{rel}} Q
  \]
  \[
  \ell \xrightarrow{\text{acq}} Q_1 \ast \ell \xrightarrow{\text{acq}} Q_2 \iff \ell \xrightarrow{\text{acq}} (Q_1 \ast Q_2)
  \]
Simple ownership transfer example

Let \( Q := \{(0, \text{emp}), (1, a \rightarrow 2)\} \).

\[
\begin{align*}
\{ \text{emp} \} \\
\{ a := \text{alloc}_{na}(0); x := \text{alloc}_{atom}(0); \}
\end{align*}
\]

\[
\begin{align*}
\{ a \rightarrow 0 \ast x \rightarrow Q \ast x \rightarrow Q \}
\end{align*}
\]

\[
\begin{align*}
\{ a \rightarrow 0 \ast x \rightarrow Q \}
\end{align*}
\]

\[
\begin{align*}
\{ x \rightarrow Q \}
\end{align*}
\]

repeat

\[
\begin{align*}
r := [x]_{acq}
\end{align*}
\]

\[
\begin{align*}
\{ r = 0 \ast x \rightarrow Q \lor r = 1 \ast a \rightarrow 2 \}
\end{align*}
\]

until \((r \neq 0); \)

\[
\begin{align*}
r := [a]_{na}
\end{align*}
\]

\[
\begin{align*}
\{ r = 2 \ast a \rightarrow 2 \}
\end{align*}
\]

{ \( r = 2 \ast a \rightarrow 2 \) }
Relaxed atomics

Basically, disallow ownership transfer.

- Relaxed reads:
  \[
  \{ \ell \xrightarrow{\text{acq}} Q \} \times := [\ell]_{\text{rlx}} \{ \ell \xrightarrow{\text{acq}} Q \land (Q(x) \neq \text{false}) \}
  \]

- Relaxed writes:
  \[
  Q(v) = \text{emp} \\
  \{ \ell \xrightarrow{\text{rel}} Q \} [\ell]_{\text{rlx}} := v \{ \ell \xrightarrow{\text{rel}} Q \}
  \]

- Unsound in C11 because of dependency cycles.
let $a = \text{alloc}_{\text{atom}}(0)$ in
let $b = \text{alloc}_{\text{atom}}(0)$ in

$(\text{if } 1 = [a]_{\text{rlx}} \text{ then}) \ || \ (\text{if } 1 = [b]_{\text{rlx}} \text{ then})$

$[b]_{\text{rlx}} := 1 \ || \ [a]_{\text{rlx}} := 1$

A problematic consistent execution:

\[\text{[Initialization actions not shown]}\]

\[\begin{align*}
\rightarrow \text{sb} & \quad \rightarrow \text{rf} & \rightarrow \text{rf} & \rightarrow \text{sb} \\
R_{\text{rlx}}(a, 1) & \quad R_{\text{rlx}}(b, 1) & & & W_{\text{rlx}}(b, 1) & \quad W_{\text{rlx}}(a, 1)
\end{align*}\]

[Crude fix: Require $hb \cup rf$ to be acyclic.]
Compare and swap (CAS)

- New assertion form, $P := \ldots | \ell^{\text{macq}} Q$.
- Duplicable, $\ell^{\text{macq}} Q \iff \ell^{\text{macq}} Q^* \iff \ell^{\text{macq}} Q$.
- Proof rule for CAS:

$$
\begin{align*}
P & \Rightarrow \ell^{\text{macq}} Q \ast \text{true} \\
(P \ast Q(v)) & \Rightarrow \ell^{\text{rel}} Q' \ast Q'(v') \ast R[v/z] \\
X \in \{\text{rel, rlx}\} & \Rightarrow Q(v) = \text{emp} \\
X \in \{\text{acq, rlx}\} & \Rightarrow Q'(v') = \text{emp} \\
\{P\} z := [\ell]_Y \{z \neq v \Rightarrow R\} & \Rightarrow \{P\} z := \text{CAS}_{X,Y}(\ell, v, v') \{R\}
\end{align*}
$$
Mutual exclusion locks

Let $Q_J(v) \overset{\text{def}}{=} (v = 0 \land \text{emp}) \lor (v = 1 \land J)$

$Lock(x, J) \overset{\text{def}}{=} x \overset{\text{rel}}{\mapsto} Q_J \ast x \overset{\text{macq}}{\mapsto} Q_J$

new-lock() \overset{\text{def}}{=} 
\{
  J
\}
res := alloc_{\text{atom}}(1)
\{
  Lock(res, J)
\}

unlock(x) \overset{\text{def}}{=} 
\{
  J \ast Lock(x, J)
\}
[x]_{\text{rel}} := 1
\{
  Lock(x, J)
\}

lock(x) \overset{\text{def}}{=} 
\{
  Lock(x, J)
\}

repeat
\{
  Lock(x, J)
\}
y := CAS_{\text{acq,rlx}}(x, 1, 0)
\{
  Lock(x, J) \ast \left( y = 0 \land \text{emp} \lor y = 1 \land J \right)
\}
until y \neq 0
\{
  J \ast Lock(x, J)
\}
Technical challenges

- Assertions in heaps
  \[\implies\] Store syntactic assertions (modulo \(\ast\)-ACI)

- No (global) notions of state and time
  \[\implies\] Define a *logical* local notion of state
  \[\implies\] Annotate hb edges with logical state

- No operational semantics
  \[\implies\] Use the axiomatic semantics
  \[\implies\] Induct over max hb-path distance from top
Possible extensions / future work

- Take more advanced program logics (rely-guarantee, RGSep, deny-guarantee, ...) and adapt them to C11 concurrency
- Handle the more advanced C11 constructs: consume atomics & fences
- Build a tool & verify real programs