Tutorial outline

Part I. Weak memory models
1. Introduction to relaxed concurrency
2. The C11 relaxed memory model

Part II. Relaxed program logics
3. Concurrent separation logic
4. Relaxed separation logic
5. RSL extensions (ongoing)

http://www.mpi-sws.org/~viktor/rsl/
Sequential consistency

Sequential consistency (SC):

- Interleave each thread’s atomic accesses.
- The standard model for concurrency.
- Almost all verification work assumes it.
- Fairly intuitive.

Initially, $x = y = 0$.

$$
\begin{align*}
  & x := 1; \quad \parallel \quad y := 1; \\
  & \quad \text{print}(y); \quad \parallel \quad \text{print}(x);
\end{align*}
$$

In SC, this program can print 01, 10, or 11.
Sequential consistency (SC):

- Interleave each thread’s atomic accesses.
- The standard model for concurrency.
- Almost all verification work assumes it.
- Fairly intuitive.

Initially, $x = y = 0$.

```plaintext
\[
x := 1; \quad \parallel \quad y := 1;
\]

```

But SC is invalidated by:

- Hardware implementations
- Compiler optimisations

In SC, this program can print 01, 10, or 11.
Initially, $x = y = 0$.

\[
x := 1; \quad \parallel \quad y := 1;
\]

\[
print(y); \quad \parallel \quad print(x);
\]

This program can also print 00.
Initially, $x = y = 0$.

$$\begin{align*}
  x &:= 1 \quad & y &:= 1 \\
  print(x); &\quad & print(y); \\
  print(y); &\quad & print(x);
\end{align*}$$

Both threads can print 10.
Initially, \( x = y = 0 \).

\[
x := 1; \quad \quad print(x); \\
y := 1; \quad \quad print(y); \\
\]

The program can print 010.

**Justification:**
The compiler may perform CSE:
Load \( x \) into a temporary \( t \)
and print \( t, y, \) and \( t \).
When should we care about relaxed memory?

All *sane* memory models satisfy the DRF property:

Theorem (DRF-property)

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

\([Prg]_{Relaxed} = [Prg]_{SC}\).

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

Two types of locations: ordinary and atomic

- Races on ordinary accesses $\leadsto$ error

A spectrum of atomic accesses:

- Relaxed $\leadsto$ no fence
- Consume reads $\leadsto$ no fence, but preserve deps
- Release writes $\leadsto$ no fence (x86); lwsync (PPC)
- Acquire reads $\leadsto$ no fence (x86); isync (PPC)
- Seq. consistent $\leadsto$ full memory fence

Primitives for explicit fences
C11 executions

- Execution = set of events & a few relations:
  - sb: sequenced before
  - rf: reads-from map
  - mo: memory order per location
  - sc: seq.consistency order
  - sw \[\text{derived}\]: synchronized with
  - hb \[\text{derived}\]: happens before

- Axioms constraining the consistent executions.

- \( C(\text{\textit{\(prog\)}}) = \text{set of all consistent exec’s.} \)

- if all \( C(\text{\textit{\(prog\)}}) \) race-free on ordinary accesses, \( \llbracket \text{\textit{\(prog\)}} \rrbracket = C(\text{\textit{\(prog\)}}); \) otherwise, \( \llbracket \text{\textit{\(prog\)}} \rrbracket = \text{“error”} \)
Release-acquire synchronization: message passing in C11

```c
atomic_int x = 0; int a = 0;

(a = 7; if (x.load(acq) \neq 0) print(a);

x.store(1, release));
```

\[
\begin{align*}
W_{na}(x, 0) & \downarrow_{sb} W_{na}(a, 0) \\
W_{na}(a, 7) & \downarrow_{sb} W_{rel}(x, 1) \\
R_{acq}(x, 1) & \downarrow_{sb} R_{na}(a, ?)
\end{align*}
\]

happens-before \( \overset{\text{def}}{=} (\text{sequenced-before} \cup \text{sync-with})^+ \)

sync-with \((a, b) \overset{\text{def}}{=} \text{reads-from}(b) = a \land \text{release}(a) \land \text{acquire}(b)\)
Rel-acq synchronization is weaker than SC

Example (SB)

Initially, $x = y = 0$.

- $x.\text{store}(1, \text{release})$;
- $t = y.\text{load}(\text{acquire})$;
- $y.\text{store}(1, \text{release})$;
- $t' = x.\text{load}(\text{acquire})$;

This program may produce $t = t' = 0$.

Example (IRIW)

Initially, $x = y = 0$.

- $x.\text{store}(1, \text{rel})$;
- $a = x.\text{load}(\text{acq})$;
- $c = y.\text{load}(\text{acq})$;
- $d = x.\text{load}(\text{acq})$;

May produce $a = c = 1 \land b = d = 0$. 
Coherence

Example (Read-Read Coherence)

Initially, \( x = 0 \).

\[
\begin{align*}
  x.\text{store} (1, \text{rel}); & \quad x.\text{store} (2, \text{rel}); & \quad a = x.\text{load}(\text{acq}); & \quad c = x.\text{load}(\text{acq}); \\
  & & b = x.\text{load}(\text{acq}); & \quad d = x.\text{load}(\text{acq});
\end{align*}
\]

Cannot get \( a = d = 1 \land b = c = 2 \).

- Plus similar WR, RW, WW coherence properties.
- Ensure SC behaviour for a single variable.
- Also guaranteed for relaxed atomics (the weakest kind of atomics in C11).
Part II
Relaxed Program Logics

- Concurrent separation logic
- Relaxed separation logic
- RSL extensions (ongoing)
Separation logic

Key concept of ownership:

- Resourceful reading of Hoare triples.

\[ \{ P \} \ C \ \{ Q \} \]

- To access a normal location, you must own it:

\[
\{ x \mapsto v \} \ast \{ t. \ t = v \land x \mapsto v \}
\]

\[
\{ x \mapsto v \} \ast x = v'; \ \{ x \mapsto v' \}
\]

- Disjoint parallelism:

\[
\begin{align*}
\{ P_1 \} \ C_1 \ \{ Q_1 \} & \quad \{ P_2 \} \ C_2 \ \{ Q_2 \} \\
\{ P_1 \ast P_2 \} & \ C_1 \parallel C_2 \ \{ Q_1 \ast Q_2 \}
\end{align*}
\]
Ownership transfer by rel-acq synchronizations.

- Atomic allocation $\leadsto$ pick loc. invariant $Q$.

\[
\{ Q(v) \} \ x = alloc(v); \ \{ W_Q(x) \ast R_Q(x) \}
\]

- Release write $\leadsto$ give away permissions.

\[
\{ Q(v) \ast W_Q(x) \} \ x.store(v, rel); \ \{ \text{true} \}
\]

- Acquire read $\leadsto$ gain permissions.

\[
\{ R_Q(x) \} \ t = x.load(acq); \ \{ Q(t) \}
\]
Message passing in RSL

Let \( Q(v) \overset{\text{def}}{=} v = 0 \lor &a \mapsto 7. \)

\[
\begin{align*}
\{\text{true}\} \\
\text{atomic\_int } x = 0; \text{ int } a = 0; \\
\{&a \mapsto 0 \ast W_{Q}(x) \ast R_{Q}(x)\} \\
a = 7; \\
\{&a \mapsto 7 \ast W_{Q}(x)\} \\
x.\text{store}(1, \text{release}); \\
\{\text{true}\} \\
\{\text{true}\}
\end{align*}
\]
Multiple readers/writers

Write permissions can be duplicated:

\[ \mathbf{W}_Q(\ell) \iff \mathbf{W}_Q(\ell) \ast \mathbf{W}_Q(\ell) \]

Read permissions cannot, but may be split:

\[ \mathbf{R}_{Q_1 \ast Q_2}(\ell) \iff \mathbf{R}_{Q_1}(\ell) \ast \mathbf{R}_{Q_2}(\ell) \]

\[
\begin{align*}
a &= 7; \\
b &= 8; \\
x.\text{store}(1, \text{rel}) &; \\
\quad t = x.\text{load}(\text{acq}); \\
\quad \text{if } (t \neq 0) &; \\
\quad \text{print}(a); \\
\quad t' = x.\text{load}(\text{acq}); \\
\quad \text{if } (t' \neq 0) &; \\
\quad \text{print}(b);
\end{align*}
\]
Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

\[
\{ R_Q(x) \} \; x.\text{load}(rlx) \; \{ y. \; Q(y) \not\equiv \text{false} \}
\]

- Relaxed writes:

\[
Q(v) = \text{emp} \\
\Rightarrow \\
\{ W_Q(x) \} \; x.\text{store}(v, rlx) \; \{ \text{true} \}
\]
Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:
  \[ \{ R_Q(x) \} \ x.\text{load}(rlx) \{ y. \ Q(y) \neq false \} \]

- Relaxed writes:
  \[
  Q(v) = \text{emp} \\
  \{ W_Q(x) \} \ x.\text{store}(v, rlx) \{ \text{true} \}
  \]

Unfortunately not sound because of a bug in the C11 memory model.
Dependency cycles in C11

Initially $x = y = 0$.

\[
\text{if } (x.\text{load}(rlx) == 1) \quad \text{if } (y.\text{load}(rlx) == 1)
\]
\[
y.\text{store}(1, rlx); \quad \quad \quad x.\text{store}(1, rlx);
\]

The formal C11 model allows $x = y = 1$.

**Justification:**

\[
R_{rlx}(x, 1) \quad R_{rlx}(y, 1) \\
W_{rlx}(y, 1) \quad W_{rlx}(x, 1)
\]

Relaxed accesses don’t synchronize.
Dependency cycles in C11

Initially $x = y = 0$.

$$\textbf{if } (x.load(rlx) == 1) \quad \textbf{if } (y.load(rlx) == 1)$$

$$y.store(1, rlx); \quad x.store(1, rlx);$$

The formal C11 model allows $x = y = 1$.

**What goes wrong:**
Non-relational invariants are unsound.

$$x = 0 \land y = 0$$

The DRF-property does not hold.
Dependency cycles in C11

Initially $x = y = 0$.

\[
\text{if } (x.load(rlx) == 1) \quad \parallel \quad \text{if } (y.load(rlx) == 1)
\]
\[
y.store(1, rlx);
\]
\[
x.store(1, rlx);
\]

The formal C11 model allows $x = y = 1$.

**How to fix this:**

Don’t use relaxed writes

\[
\lor
\]

Require $acyclic(sb \cup rf)$.

(Disallow RW reordering.)
Compare and swap (CAS)

- New assertion form, $P := \ldots \mid R^U_Q(x)$.
- Duplicable, $R^U_Q(x) \iff R^U_Q(x) \ast R^U_Q(x)$.

\[
X \in \{rel, rlx\} \Rightarrow Q(v) \equiv emp \\
X \in \{acq, rlx\} \Rightarrow Q(v') \equiv emp \\
P \ast Q(v) \Rightarrow Q(v') \ast R[v/z] \\
\{P\} x.load(Y) \{z. z \neq v \Rightarrow R\} \\
\{R^U_Q(x) \ast W_Q(x) \ast P\} x.CAS(v, v', X, Y) \{z. R\}
\]
Mutual exclusion locks

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

$\text{Lock}(x, J) \overset{\text{def}}{=} W_{Q_J}(x) \ast R_{Q_J}(x)$

$new-lock() \overset{\text{def}}{=} 
\begin{cases} 
\{J\} \\
\text{res} = \text{alloc}(1) \\
\{\text{Lock}(\text{res}, J)\} 
\end{cases}$

$\text{unlock}(x) \overset{\text{def}}{=} 
\begin{cases} 
\{J \ast \text{Lock}(x, J)\} \\
x.\text{store}(1, \text{rel}) \\
\{\text{Lock}(x, J)\} 
\end{cases}$

$\text{lock}(x) \overset{\text{def}}{=} 
\begin{cases} 
\{\text{Lock}(x, J)\} \\
\text{repeat} \\
\quad \{\text{Lock}(x, J)\} \\
\quad y = x.\text{CAS}(1, 0, \text{acq}, \text{rlx}) \\
\quad \{\text{Lock}(x, J) \ast \left( y = 0 \land \text{emp} \lor y = 1 \land J \right)\} \\
\text{until } y \neq 0 \\
\quad \{J \ast \text{Lock}(x, J)\} 
\end{cases}$
Three key features:

- Location invariants protocols
- Ghost state/tokens
- Escrows for ownership transfer

Example (Racy message passing)

Initially, $x = y = 0$.

$x$.store(1, $r/\ell x$); \hspace{1em} x$.store(1, $r/\ell x$); \hspace{1em} t = y$.load$($acq$);$

\hspace{1em} y$.store(1, $rel$); \hspace{1em} y$.store(1, $rel$); \hspace{1em} t' = x$.load($r/\ell x$);

Cannot get $t = 1 \land t' = 0$. 
Racy message passing in GPS

Protocol for $x$: \[ A: x = 0 \quad \rightarrow \quad B: x = 1 \]

Protocol for $y$: \[ C: y = 0 \quad \rightarrow \quad D: y = 1 \land x.st \geq B \]

Acquire reads gain knowledge, not ownership.

\[
\begin{align*}
\{ x.st \geq A \land y.st \geq C \} & \quad x.\text{store}(1, rlx); \quad \{ x.st \geq A \land y.st \geq C \} \\
\{ x.st \geq B \land y.st \geq C \} & \quad y.\text{store}(1, rel); \quad \{ x.st \geq B \land y.st \geq D \} \\
\{ x.st \geq B \land y.st \geq D \} & \quad t = y.\text{load}(acq); \quad \begin{cases} t = 0 \land x.st \geq A \\ t = 1 \land x.st \geq B \end{cases} \\
\{ t = 0 \lor (t = 1 \land t' = 1) \} & \quad t' = x.\text{load}(rlx); \quad \{ t = 0 \lor (t = 1 \land t' = 1) \}
\end{align*}
\]
GPS ghosts and escrows

To gain ownership, we use ghost state & escrows.

\[
P \ast P \Rightarrow \text{false} \quad \text{Esc}(P, Q) \quad \text{Esc}(P, Q) \ast P \Rightarrow Q
\]

Example (Message passing using escrows)

Invariant for \( x \): \( x = 0 \lor \text{Esc}(K, \&a \mapsto 7) \).

\[
\begin{align*}
\{ \&a \mapsto 0 \} \\
a &= 7; \\
\{ \&a \mapsto 7 \} \\
\{ \text{Esc}(K, \&a \mapsto 7) \} \quad \longleftarrow \quad \{ K \} \\
\text{x.store}(1, \text{rel}); \\
\end{align*}
\]

\[
\begin{align*}
\{ \&a \mapsto 7 \} \\
\{ K \} \\
\text{if } (x.\text{load}(acq) \neq 0) \\
\{ K \ast \text{Esc}(K, \&a \mapsto 7) \} \\
\{ \&a \mapsto 7 \} \\
\text{print}(a); \\
\end{align*}
\]
Incorrect message passing

```c
int a; atomic_int x = 0;

(a = 7; || if (x.load(rlx) ≠ 0) {
      x.store(1, rlx);
      print(a);
      } )
```

Viktor Vafeiadis
Relaxed Separation Logic

25/28
Message passing with C11 memory fences

```
int a; atomic_int x = 0;
(a = 7; if (x.load(rlx) \neq 0){
    fence(release);
    fence(acq);
    x.store(1, rlx);
    print(a); })
```

![Diagram of message passing with C11 memory fences](image)
Reasoning about fences
Joint work with Marko Doko (in progress)

Let \( Q(v) \overset{\text{def}}{=} v = 0 \lor \& a \mapsto 7. \)

\[
\begin{align*}
\{ &\& a \mapsto 0 \ast W_{Q}(x) \ast R_{Q}(x) \} \\
\{ &\& a \mapsto 0 \ast W_{Q}(x) \} \\
a = 7; \\
\{ &\& a \mapsto 7 \ast W_{Q}(x) \} \\
fence(\text{release}) \\
\{ \triangle (\& a \mapsto 7) \ast W_{Q}(x) \} \\
x.\text{store}(1, rlx); \\
\{ \text{true} \}
\end{align*}
\]

\[
\begin{align*}
t &= x.\text{load}(rlx); \\
\{ \nabla(t = 0 \lor \& a \mapsto 7) \} \\
\text{if } (t \neq 0) \\
fence(\text{acq}); \\
\{ &\& a \mapsto 7 \} \\
\text{print}(a); \\
\{ \text{true} \}
\end{align*}
\]

\[
\begin{align*}
\{ P \} &\text{ fence(\text{release}) } \{ \triangle P \} \\
\{ \nabla P \} &\text{ fence(\text{acq}) } \{ P \} \\
\{ R_{Q}(x) \} &\text{ x.\text{load}(rlx) } \{ y. \nabla Q(y) \} \\
\{ W_{Q}(x) \ast \triangle Q(v) \} &\text{ x.\text{store}(v, rlx) } \{ \text{true} \}
\end{align*}
\]
Release-consume synchronization

Initially $a = x = 0$.

\[
\begin{align*}
  a &= 5; \\
  x &. \text{store} (\text{release}, \& a); \\
  t &= x. \text{load} (\text{consume}); \\
  \text{if } (t \neq 0) &\text{ print} (*t);
\end{align*}
\]

This program cannot crash nor print 0.

**Justification:**

\[ W_{\text{na}}(a, 5) \quad \xrightarrow{\text{Release-consume synchronization}} \quad W_{\text{rel}}(x, \& a) \]

\[ R_{\text{con}}(x, \& a) \quad \xrightarrow{\text{Release-consume synchronization}} \quad R_{\text{na}}(a, 5) \]
Release-consume synchronization

Initially $a = x = 0$. Let $J(t) \overset{\text{def}}{=} t = 0 \lor t \mapsto 5$.

\[
\begin{align*}
\{&a \mapsto 0 \ast W_J(x)\} & \{&R_J(x)\} \\
\{&a \mapsto 5 \ast W_J(x)\} & \{&t = x.load(\text{consume})\} \\
x.\text{store}(\text{release}, &a); & \{&\nabla_t (t = 0 \lor t \mapsto 5)\} \\
\end{align*}
\]

This program cannot crash nor print 0.

Index the $\nabla$ with program variable $t$. $t$ data dependence $\implies$ locally open $\nabla_t$.

Again, work in progress...
Conclusion

Take away:

Formal reasoning about C11 programs is not so difficult after all.

We’re not quite there yet; there’s still a lot to do:

Liveness, refinement, tool support, . . .

Topics that were not covered:

- The soundness proof:
  Really interesting and fully mechanized in Coq.
- Found 4+1 bugs in the C11 model.
  Program logic as a debugging tool for WMM.