Program logics for relaxed consistency
UPMARC Summer School 2014

Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)

1st Lecture, 28 July 2014
Outline

Part I. Weak memory models
   1. Intro to relaxed memory consistency
   2. The C11 memory model

Part II. Program logics
   3. Separation logic
   4. Relaxed separation logic
   5. GPS : ghosts & protocols
   6. Advanced features

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, $x = y = 0$.

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

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, 
\[ x = y = 0. \]

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

In SC, this program can print 01, 10, or 11.

But SC is invalidated by:

- Hardware implementations
- Compiler optimisations
Initially, $x = y = 0$.

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

\[
\text{print}(y); \quad \text{print}(x);
\]

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

$x := 1$; $\quad \quad$ \hspace{1cm} \text{print}(x)$;
$y := 1$; $\quad \quad$ \hspace{1cm} \text{print}(y)$;
$\quad \quad$ \hspace{1cm} \text{print}(x)$;

Can the program print 010?

**Justification:**
The compiler may perform CSE:
Load $x$ into a temporary $t$
and print $t$, $y$, and $t$. 
Initially, $x = y = 0$.

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

Both threads can print 10.
Take the IRIW example:

\[
\begin{align*}
  x &:= 1 & y &:= 1 & \text{print}(x); & \text{print}(y); \\
  & & & \text{print}(y); & \text{print}(x);
\end{align*}
\]

Linearize twice (threads 1-3 and 2-4):

\[
\begin{align*}
  x &:= 1; & y &:= 1; & \text{print}(x); & \text{print}(y); \\
  & & & \text{print}(y); & \text{print}(x);
\end{align*}
\]

That’s the store buffering example (with two extra print statements).
Coherence

Initially, $x = 0$.

$x = 1; \parallel print(x);$

$x = 2; \parallel print(x);$  

Cannot print 10 nor 20 nor 21.

- Programs with one shared variable have SC semantics.
- Ensured by the cache coherence protocol.
Message passing

Initially, \( x = y = 0 \).

\[
\begin{align*}
x &= 1; & \quad \text{print}(y); \\
[\text{WW fence}] & & [\text{RR fence}] \\
y &= 1; & \quad \text{print}(x);
\end{align*}
\]

Cannot print 10.

- No fences needed on x86-TSO
- lwsync/isync on Power
- dmb/isync on ARM
Understanding weak memory consistency

Read the architecture/language specs?
  ▶ Too informal, often wrong.

Read the formalisations?
  ▶ Fairly complex.

Run benchmarks / Litmus tests?
  ▶ Observe only subset of behaviours.

We need a better methodology...
Which memory model?

Hardware or language models?
- Want to reason at “high level”
- TSO \( \sim \) good robustness theorems

C/C++ or Java?
- JMM is broken [Ševčík & Aspinall, ECOOP’08]
- So, only C11 left

Goals:
- Understand the memory model
- Verify intricate concurrent 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 [derived]: synchronized with
  - hb [derived]: happens before

- Axioms constraining the consistent executions.

\[ C(\langle prog \rangle) = \text{set of all consistent exec's.} \]

- if all \( C(\langle prog \rangle) \) race-free on ordinary accesses, \( \llbracket \text{prog} \rrbracket = C(\langle \text{prog} \rangle) \); otherwise, \( \llbracket \text{prog} \rrbracket = \text{“error”} \)
atomic_int \ x = 0; int \ a = 0;
\left( a = 7; \quad \begin{array}{l}
\text{if} \ (x.\text{load}(acq) \neq 0) \\
\text{print}(a);
\end{array}
\right)
\begin{align*}
W_{na}(x, 0) & \downarrow_{sb} W_{na}(a, 0) \\
W_{na}(a, 7) & \downarrow_{sb} R_{acq}(x, 1) \\
W_{rel}(x, 1) & \downarrow_{sb} R_{na}(a, ?)
\end{align*}

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

\text{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, $x = y = 0$.

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

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

Example (IRIW)

Initially, $x = y = 0$.

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

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

Example (Read-Read Coherence)

Initially, $x = 0$.

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

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).
Today:
- Separation logic
- Relaxed separation logic
When should we care about relaxed memory?

All *sane* memory models satisfy the DRF property:

**Theorem (DRF-property)**

If $\sem{Prg}_{SC}$ contains no data races, then $\sem{Prg}_{Relaxed} = \sem{Prg}_{SC}$.

- Program logics that disallow data races are trivially sound.
- What about *racy* programs?
Separation logic assertions

Assertions describe the heap \((\text{Loc} \rightarrow \text{Val})\):

- **emp**: the empty heap
- **\(\ell \mapsto v\)**: a cell at address \(\ell\) containing \(v\)

\[
h \models \ell \mapsto v \iff h = \{\ell \mapsto v\}
\]

- **\(P \ast Q\)**: separating conjunction

\[
h \models P \ast Q \iff \exists h_1 h_2. \ h = h_1 \cup h_2 \land h_1 \models P \land h_2 \models Q
\]

- **\&, \lor, \neg, true, false, \forall, \exists**: as usual
The separating conjunction

Some basic properties:

- $\ast$ is commutative & associative.
- $P \ast \text{emp} \iff \text{emp} \ast P \iff P$
- $\ell \mapsto v \ast \ell \mapsto v' \implies \text{false}$

Useful for describing inductive data structures:

- $\text{list}(x) \define (x = 0 \land \text{emp}) \lor \exists y. x \mapsto y \ast \text{list}(y)$
- $\text{ls}(x, z) \define (x = z \land \text{emp}) \lor \exists y. x \mapsto y \ast \text{ls}(y, z)$
- $\text{tree}(x) \define (x = 0 \land \text{emp}) \lor \exists y, z. x \mapsto y \ast (x+1 \mapsto z \ast \text{tree}(y) \ast \text{tree}(z)$
Separation logic

Key concept of *ownership*:

- Resourceful reading of Hoare triples

\[
\begin{align*}
\{P_1\} & C_1 \{Q_1\} & \{P_2\} & C_2 \{Q_2\} \\
\{P_1 \ast P_2\} & C_1 \| C_2 \{Q_1 \ast Q_2\} \\
\{P\} & C \{Q\} \\
\{P \ast R\} & C \{Q \ast R\}
\end{align*}
\]

- Ensure memory safety & race-freedom

(Par)

(Frame)
Separation logic rules for non-atomic accesses

- Allocation gives you permission to access $x$.

$$\{\text{emp}\} \; x = \text{alloc}(); \; \{\exists v. \; x \mapsto v\}$$

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

$$\{x \mapsto v\} \; t = *x; \; \{x \mapsto v \land t = v\}$$

$$\{x \mapsto v\} \; *x = v'; \; \{x \mapsto v'\}$$
Initially $a = x = 0$.

\[
\begin{align*}
a &= 5; \\
x &.\text{store}(\text{release}, 1); \\
\text{while } (x .\text{load}(\text{acq}) == 0); \\
\text{print}(a);
\end{align*}
\]

This will always print 5.

**Justification:**

$$
\begin{array}{c}
W_{na}(a, 5) \quad R_{acq}(x, 1) \\
\downarrow \quad \downarrow \quad \downarrow \\
W_{rel}(x, 1) \quad R_{na}(x, 5)
\end{array}
$$

Release-acquire synchronization
Ownership transfer by rel-acq synchronizations.

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

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

- Release write $\leadsto$ give away permissions.

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

- Acquire read $\leadsto$ gain permissions.

\[
\{ R_Q(x) \} \ t = x.\text{load}(\text{acq}); \ \{ Q(t) \ast R_Q[t:=\text{emp}](x) \}
\]
Let \( Q(v) \) \( \overset{\text{def}}{=} v = 0 \lor a \mapsto 5. \)

\[
\begin{align*}
\{ \text{true} \} \\
\text{atomic_int } x = 0; \text{ int } a = 0; \\
\{ a \mapsto 0 \ast W_Q(x) \ast R_Q(x) \} \\
a = 5; \\
\{ a \mapsto 5 \ast W_Q(x) \} \\
x.\text{load}(acq) == 0; \\
\{ a \mapsto 5 \} \\
x.\text{store}(1, \text{release}); \\
\{ \text{true} \} \\
\{ a \mapsto 5 \} \\
\{ \text{true} \}
\end{align*}
\]
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:

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

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

Basically, disallow ownership transfer.

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

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

Unfortunately not sound because of a bug in the C11 memory model.
Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

  \[
  \{R_Q(x)\} \ t = x.\text{load}(rlx) \ \{R_Q(x) \ast (Q(t) \neq \text{false})\}
  \]

- Relaxed writes:

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

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

Initially $x = y = 0$.

```
if (x.load(rlx) == 1)    if (y.load(rlx) == 1)
y.store(1, rlx);          x.store(1, rlx);
```

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

**Justification:**

```
R_{rlx}(x, 1) \rightarrow W_{rlx}(y, 1) \rightarrow W_{rlx}(x, 1)
R_{rlx}(y, 1) \rightarrow W_{rlx}(y, 1)
```

Relaxed accesses don’t synchronize
Initially $x = y = 0$.

$$\begin{align*}
\textbf{if } (x.\text{load}(rlx) == 1) & \quad \textbf{if } (y.\text{load}(rlx) == 1) \\
y.\text{store}(1, rlx); & \quad x.\text{store}(1, rlx);
\end{align*}$$

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 $x = y = 0$.

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

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.)
Conclusion

Topics covered today:
- The C11 memory model
- Separation logic
- Relaxed separation logic

Tomorrow:
- Compare and swap
- GPS
- Advanced C11 features