Relaxed Separation Logic

Our goal here is to reason about C11 release-acquire accesses.

Why not SC accesses?

→ Because (in the absence of rel/acq accesses) they are straightforward – Just use the DRF theorem.

Why not relaxed accesses?

→ Eventually, we also want to reason about these, but they are more complicated because of dependency cycles.

(1) Recall that the program

```
                      x = y = 0;
if(x.load(rlx)==1)        ||  if (y.load(rlx) == 1)
   y.store(1,rlx);        ||      x.store(1,rlx);
```

can terminate in a final state satisfying $x=y=1$.

(2) This breaks even the most basic forms of reasoning such as non-relational invariants (i.e., a conjunction whose conjuncts are predicates describing at most one variable each). For example, the assertion

$$x = 0 \quad \wedge \quad y = 0$$

is an invariant of the program above under all interleaving executions, but not under all C11 consistent executions.

(3) It also breaks the DRF theorem: the program above has
("Dep. Cycles") no races under SC semantics, and yet exhibits a behaviour that is not possible under SC.

(4) In a stronger model, where we rule out (hb ∪ rf) cycles, we can reason about relaxed accesses, but we postpone this extension.

Why reason about REL/ACQ accesses?

→ They can be implemented very efficiently on x86-TSO, but also are portable (and reasonably efficient) on Power and ARM, whereas TSO accesses cannot be efficiently implemented on Power/ARM.

→ The different behaviours one gets with REL/ACQ versus TSO don't seem to impact formal reasoning unless one uses memory fences.

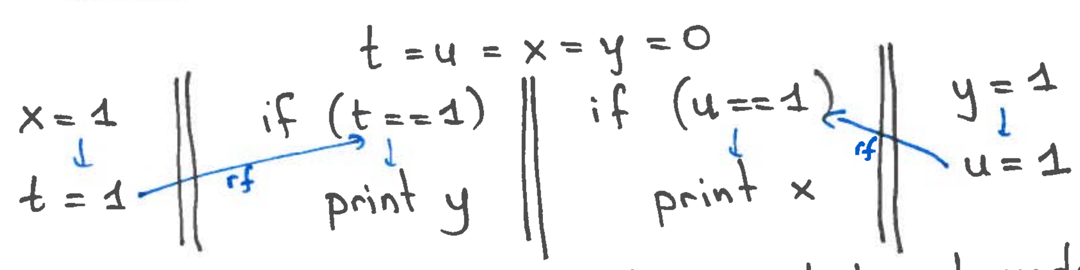
Recall : Store buffering (SB)

$x = y = 0$

$x = 1$	$y = 1$
print y	print x

} Can print 0 & 0 on both TSO and REL/ACQ.

Independent reads of independent writes (IRIW)



Can print 0 & 0 on REL/ACQ, but not under TSO.

→ The C11 memory model is fairly complicated (even when restricted to the release-acquire fragment), so it is useful to have an alternative higher-level explanation of REL/ACQ synchronization.

→ We will actually show that REL/ACQ sychronization corresponds closely to a familiar concept from concurrent separation logic — that of ownership transfer.

Reasoning about non-atomic accesses

Before starting with REL/ACQ accesses, let's start with the non-atomic ones.

- Recall that data races on NA accesses are deemed as program errors and result in undefined behaviour.
- So let's use a program logic that rules out races: (concurrent) separation logic. Recall the basic rules:

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

$$\{x \mapsto v\}\ *x = v';\ \{x \mapsto v'\}$$

(in C syntax) – the typical SL syntax is $[x] = v'$

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

"To access x you must own x."

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1 ; C_2\ \{R\}}\ (SEQ)$$

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}\ \circledast}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ (PAR)$$

$\circledast$ side-conditions elided

$$\frac{}{\{P\}\ skip\ \{P\}}\ (SKIP)$$

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{P\}\ C_2\ \{Q\}}{\{P\}\ C_1 \oplus C_2\ \{Q\}}\ (CHOICE)$$

$$\frac{\{P\}\ C\ \{P\}}{\{P\}\ C^*\ \{P\}}\ (LOOP)$$

side-conditions elided ($writes(C) \cap fv(F) = \emptyset$)

$$\frac{\circledast\ \{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\ (FRAME)$$

$$\frac{\{P\}\ C\ \{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}\ (CONSEQ)$$

What do we do with REL/ACQ accesses?

→ Take an idea from CSL & adapt it: In CSL, we had resource invariants, J. Here, we will have an invariant for each atomic location. The invariant will describe both the value at that (atomic) location and any other state the location owns. That is, location invariants will have type $2^{Val \times heap}$, i.e. $Val \to SLAssertion$.

→ Allocating & initializing an atomic location requires as a precondition that the invariant holds for the new location & the value being written to it. In response, we get the permission to read and write atomically to that location with the loc. inv.

$$\{ J(v) \} \quad x = alloc(v); \quad \{ W_J(x) * R_J(x) \}$$

$W_J(x)$ & $R_J(x)$ are two new assertion forms denoting the permission to write (resp. read) the location x with loc. inv. J.

→ The axiom for writes is straightforward: we have to establish the loc. inv. (and have permission to write).

$$\{ J(v) * W_J(x) \} \; x.store(v, rel); \; \{ W_J(x) \}$$

We need the W permission so that we know x is allocated (else the write would fail) and to ensure the correct J is used.

→ The axiom for reads is a bit more complicated:

$$\{ \mathbf{R}_J(x) \} \quad t = x.load(acq) \quad \{ \mathbf{R}_{J[t:=emp]}(x) * J(t) \}$$

We require the read permission in the precondition and in response we get that the location invariant holds in the postcondition.
Note, however, that unlike the rule for writes, we do not keep the full permission for reading x, once we read x, but only $\mathbf{R}_{J[t:=emp]}$, where

$$J[t:=emp] \stackrel{def}{=} \lambda y.\ \text{if } y=t \text{ then emp else } J(y)$$

This is because if we'd retained the full permission, we would be able to read x twice, and get $J(t)$ out twice, which could be inconsistent (e.g., if $J(-) \stackrel{def}{=} y \mapsto -$)

"
$$\{ \mathbf{R}_J(x) \} \quad t = x.load(acq)$$
$$\{ \mathbf{R}_J(x) * J(t) \}$$
$$t' = x.load(acq)$$
$$\{ \mathbf{R}_J(x) * J(t) * J(t') \}$$
"

By making the postcondition have $\mathbf{R}_{J[t:=emp]}(x)$ instead, we ensure that if we read x again and get the same value, then at least we won't get any new ownership out of reading x the second time.

Example – Message Passing

> Let $J(v) \stackrel{def}{=} v=0 \vee x \mapsto 7$

$\{emp\}$

$x = alloc(0)$

$\{x \mapsto 0\}$

$y = alloc(0)$

$\{x \mapsto 0 * W_J(y) * R_J(y)\}$

Left thread:

$\{x \mapsto 0 * W_J(y)\}$

$*x = 7$

$\{x \mapsto 7 * W_J(y)\}$

$y.store(1, rel)$

$\{W_J(y)\}$

$\|$

Right thread:

$\{R_J(y)\}$

$t = y.load(acq);$

$\{(t=0 \vee x \mapsto 7) * R_{J[t:=emp]}(y)\}$

$assume(t \neq 0);$

$\{x \mapsto 7 * R_{J[t:=emp]}(y)\}$

$t' = *y$

$\{x \mapsto 7 * R_J[t:=emp](y) \wedge t'=7\}$

$\{W_J(y) * x \mapsto 7 * R_J[t:=emp](y) \wedge t'=7\}$

$\{t'=7\}$

Multiple Readers – Multiple Writers

For writers, we can duplicate the permission:

$$\mathbf{W}_J(x) \Leftrightarrow \mathbf{W}_J(x) * \mathbf{W}_J(x)$$

For readers, we cannot duplicate the permission as for writers (because we could then read twice and get J hold twice), but we can "split" the permission:

$$\mathbf{R}_{J_1 * J_2}(x) \Leftrightarrow \mathbf{R}_{J_1}(x) * \mathbf{R}_{J_2}(x)$$

This allows us to reason about programs such as:

```
*a = 7;            ||  if (y.load(acq))  ||  if (y.load(acq)){
*b = 8;            ||     t1 = *a;       ||     t2 = *b;
y.store(1,rel);    ||                    ||     *b = t2+1
                                             }
```

that initialize two data structures and use one REL/ACQ variable to sychronize.

Relaxed accesses

In the model where $(hb \cup rf)$ is acyclic, we can allow the following rules:

$$(\text{W-RLX}) \quad \frac{J(v) = \text{emp}}{\{W_J(x)\}\ x.\text{store}(v, \text{rlx})\ \{W_J(x)\}}$$

$$(\text{R-RLX}) \quad \frac{}{\{R_J(x)\}\ t = x.\text{load}(\text{rlx})\ \left\{ R_J(x) \wedge \text{"}J(t) \text{ is satisfiable"} \right\}}$$

Basically, we cannot transfer any ownership, as relaxed accesses don't synchronize. When reading x, all we get to know is that the invariant of the value read is not false (i.e., is satisfiable);

$\exists h \models J(t)$. there exists a heap satisfying it.

Thus for instance we can verify that the program

$$x = \text{alloc}(0)$$

$$\begin{pmatrix} t = x.\text{load}(\text{rlx}); \\ x.\text{store}(t+2, \text{rlx}) \end{pmatrix}^* \Bigg\| \begin{pmatrix} u = x.\text{load}(\text{rlx}); \\ x.\text{load}(t*2, \text{rlx}) \end{pmatrix}^*$$

$$m = x.\text{load}(\text{rlx});$$

returns an even number as m. (Pick $J(v) \overset{\text{def}}{=} v$ is even.)