Compare and Swap (CAS)

(Also known as compare_exchange, CMPXCHG on x86)

- CAS is an atomic instruction that performs a read and a write if the read gets a certain value.

```
x.CAS(old, new) =def
    atomic {
        if (x.load() == old) {
            x.store(new);
            return true;
        } else {
            return false;
        }
    }
```

- It is widely used for synchronisation, e.g. implementing locks:

```
lock()   =def  while(¬x.CAS(0,1));
unlock() =def  x.store(0);
```

or an atomic update:

```
⟨x := f(x)⟩ ⇝   do {
                    t = x.load();
                    n = f(t);
                } while(¬x.CAS(t,n));
```

- In C11, it also gets an access-type argument:

RLX, REL, ACQ, REL-ACQ, SC

These intuitively mean that the load & the store of the CAS get the appropriate annotation:

load: RLX, RLX, ACQ, ACQ, SC

store: RLX, REL, RLX, REL, SC.

Consistency Axioms for CAS

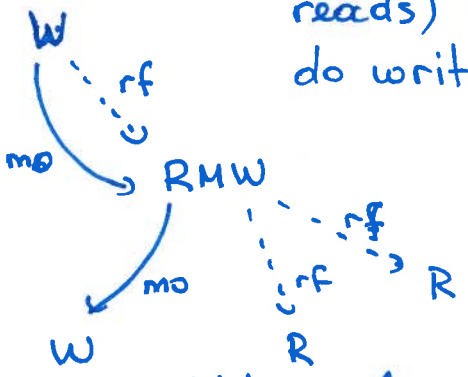
- In C11 executions, successful CAS operations (i.e., ones where the store happens) are modeled by a new kind of event:

$$RMW_{typ}(\ell, v, v')$$

read-modify-write. — value read — value written

Failed CAS operations (i.e., where the store does not happen) are modeled as normal read events.

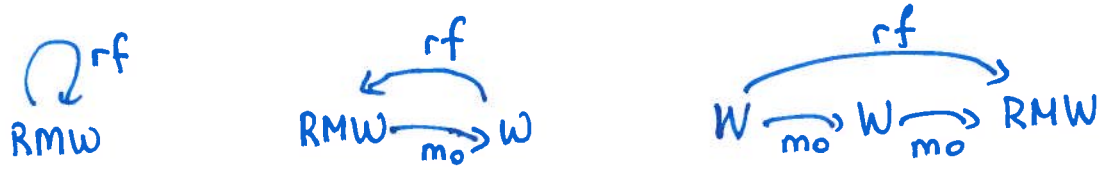
- In the C11 axioms, RMW events are treated both as reads and writes. The have incoming $\xrightarrow{rf}$ edges (as do reads) and outgoing $\xrightarrow{rf}$ edges (as do writes). Moreover, they are included in the memory order ("coherence order"), mo.



- There is one additional axiom asserting that RMWs are atomic:

$$\forall a,b.\quad rf(b) = a \;\wedge\; isRMW(b) \;\Rightarrow$$
$$mo(a,b) \;\wedge\; \nexists c.\; mo(a,c) \wedge mo(c,b)\;.$$

That is, RMWs read from their immediate mo-predecessor. In essence, this axiom rules out the following pictures



Proof Rules for CAS

- We introduce a new permission for performing CAS:

$$P ::= \ldots \mid \mathbf{C}_J(x)$$

- This permission is duplicable:

$$\mathbf{C}_J(x) \iff \mathbf{C}_J(x) * \mathbf{C}_J(x)$$

& generated at allocation:

$$\{ J(v) \} \quad x = \mathrm{alloc}(v); \quad \{ \mathbf{C}_J(x) * \mathbf{W}_J(x) \}$$

Note: No $\mathbf{R}_J(x)$!!!
At allocation, choose whether it's a location used for reads or CAS.

- It allows doing a read without gaining ownership (cf. the relaxed-read rule):

$$\{ \mathbf{C}_J(x) \} \quad t = x.\mathrm{load}(typ) \quad \{ \mathbf{C}_J(x) \wedge \text{"}J(t) \text{ satisfiable"} \}$$

- And importantly, it allows one to do a CAS:

$$\frac{\begin{array}{ll} t \wedge P * J(v) \Rightarrow Q * J(v') & \leftarrow \text{successful case} \\ \neg t \wedge P \Rightarrow Q & \leftarrow \text{unsuccessful CAS} \\ typ \in \{rel, rlx\} \Rightarrow J(v) = \mathrm{emp} & \leftarrow \text{not ACQ} \\ typ \in \{acq, rlx\} \Rightarrow J(v') = \mathrm{emp} & \leftarrow \text{not REL} \end{array}}{\{ \mathbf{C}_J(x) * P \} \quad t = x.\mathrm{CAS}_{typ}(v, v') \quad \{ \mathbf{C}_J(x) * Q \}}$$

- A successful CAS gains the ownership of $J(v)$ and loses ownership of $J(v')$.
- A failed CAS does no ownership transfer.
- If the CAS is not at least of ACQ kind, it cannot acquire any ownership as it doesn't sychronize (so $J(v) = \mathrm{emp}$)
- Symmetrically for release.

Verifying the CAS-based lock

Specs:

There exists a lock predicate $Lock(\ell, P)$ such that:

$$Lock(\ell, P) \iff Lock(\ell, P) * Lock(\ell, P)$$

$$\{ P \} \quad \ell = newLock() \quad \{ Lock(\ell, P) \}$$

$$\{ Lock(\ell, P) \} \quad lock(\ell) \quad \{ Lock(\ell, P) * P \}$$

$$\{ Lock(\ell, P) * P \} \quad unlock(\ell) \quad \{ Lock(\ell, P) \}$$

Implementation / proof:

Let $J(v) \stackrel{def}{=} (v = 0 \wedge P) \vee (v = 1 \wedge emp)$

Let $Lock(\ell, P) \stackrel{def}{=} \mathbf{C}_J(\ell) * \mathbf{W}_J(\ell)$

- Duplicability of Lock follows from that of **C** & **W**.

- newLock:

  $\{ P \} \quad \ell = alloc(0) \quad \{ \mathbf{C}_J(\ell) * \mathbf{W}_J(\ell) \}$

  by the allocation rule

- unlock:

  $\{ Lock(\ell, P) * P \}$

  $\{ \mathbf{W}_J(\ell) * J(0) * \mathbf{C}_J(\ell) \}$

  $\ell.store(0, rel);$ — W-REL & FRAME rules

  $\{ \mathbf{W}_J(\ell) * \mathbf{C}_J(\ell) \}$

  $\{ Lock(\ell, P) \}$

Verifying the CAS-based lock (ctd.)

- **lock:**

$\{ \text{Lock}(\ell, P) \}$

do

$\{ \text{Lock}(\ell, P) \}$

$t = \ell.\text{CAS}_{acq}(0, 1);$ — FRAME away $W_J(\ell)$ / & CAS rule

$\{ \text{Lock}(\ell, P) * (t \wedge P \vee \neg t \wedge \text{emp}) \}$

while $(\neg t)$;

$\{ \text{Lock}(\ell, P) * P \}$

Conditions of the CAS rule:

- $t \wedge J(1) \Rightarrow (t \wedge P \vee \neg t \wedge \text{emp}) * J(0)$
- $\neg t \wedge \text{emp} \Rightarrow (t \wedge P \vee \neg t \wedge \text{emp})$
- $J(0) = \text{emp}$.

Using this implementation of locks, we can now implement CSL atomic blocks as

$$\text{atomic}(C) \rightsquigarrow \text{lock}(\ell); C; \text{unlock}(\ell)$$

where ℓ is a global lock, allocated at the start of the program: Moreover, we can encode (& derive) the CSL rules as follows:

$$J \vdash_{CSL} \{P\}\ C\ \{Q\} \rightsquigarrow$$

$$\vdash_{RSL} \{\text{Lock}(\ell, J) * P\}\ C\ \{\text{Lock}(\ell, J) * Q\}$$