

FSL: A Program Logic for C11 Memory Fences

Marko Doko Viktor Vafeiadis

Max Planck Institute for Software Systems
(MPI-SWS)

VMCAI 2016

Why C11?

Oddities of weak memory

```
        x = 0;
        y = 0;
x = 1;  || y = 1;
print y; || print x;
```

Why C11?

Oddities of weak memory

```
x = 0;
y = 0;
x = 1;  || y = 1;
print y; || print x;
```

Both threads can print 0!

Why C11?

Oddities of weak memory

```
x = 0;  
y = 0;  
x = 1;  || y = 1;  
print y; || print x;
```

Both threads can print 0!

W(x,0)

W(y,0)

W(x,1)

W(y,1)

R(y,0)

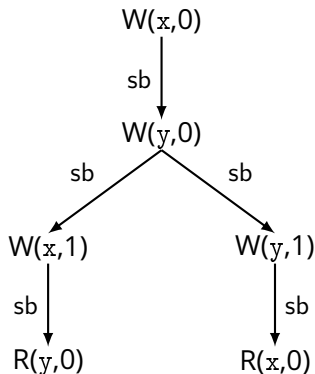
R(x,0)

Why C11?

Oddities of weak memory

```
x = 0;
y = 0;
x = 1; || y = 1;
print y; || print x;
```

Both threads can print 0!



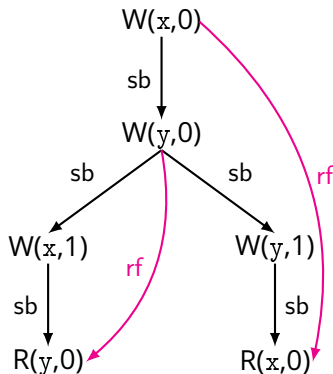
sb – sequenced-before

Why C11?

Oddities of weak memory

```
x = 0;
y = 0;
x = 1; || y = 1;
print y; || print x;
```

Both threads can print 0!



sb – sequenced-before
rf – reads-from

C11 model through examples

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;
x = 1;
    if(x == 1){
        print(a);
    }
```


C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1; ← race print(a);
        || }

```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1; ||   print(a);
      || }
      ||
      ||
```

race

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; ||   print(a);
          || }
          ||
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1;  ← race print(a);
        || }

```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; ← race print(a);
        || }

```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1;  ← race print(a);
        || }
        ||
```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; ← race print(a);
        || }
        ||
```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1;    print(a);
        || }
        ||
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1; ← race print(a);
        || }
        ||
```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; ← race print(a);
        || }
        ||
```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1; ← rf print(a);
        || }
        ||
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1;  ← race print(a);
      || }
      ||
```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; ← race print(a);
      || }
      ||
```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1; ← rf sync print(a);
      || }
      ||
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1;   race print(a);
        }

```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; race print(a);
        }

```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1; rf sync print(a);
        }

```

4

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
fencerel;   fenceacq;
xrlx = 1;   print(a);
        }

```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1; || race print(a);
      || }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; || race print(a);
          || }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1; || rf print(a);
          || sync }
```

4

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
fencerel; || rf fenceacq;
xrlx = 1; || print(a);
          || }
```


C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42; || if(x == 1){
x = 1; || race print(a);
      || }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
xrlx = 1; || race print(a);
          || }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xacq == 1){
xrel = 1; || rf print(a);
          || sync }
```

4

```
int a = 0;
atomic_int x = 0;
a = 42; || if(xrlx == 1){
fencerel; || rf print(a);
          || sync }
```

C11 model through examples

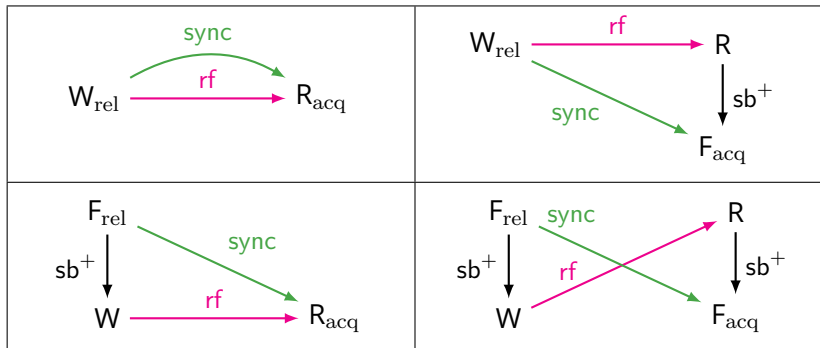
Why use fences?

Release and acquire constructs are expensive!

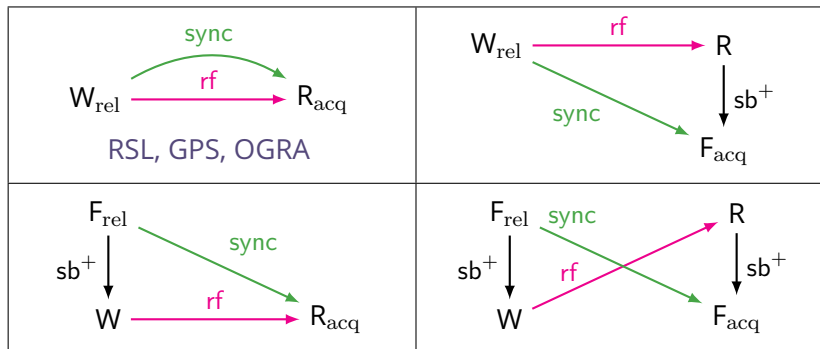
```
int a = 0;
atomic_int x = 0;
a = 42;
x_rel = 1;
||| if(x_acq == 1){
    print(a);
}
```

```
int a = 0;
atomic_int x = 0;
a = 42;
fence_rel;
x_rlx = 1;
||| if(x_rlx == 1){
    fence_acq;
    print(a);
}
```

The synchronizes-with relation



The synchronizes-with relation

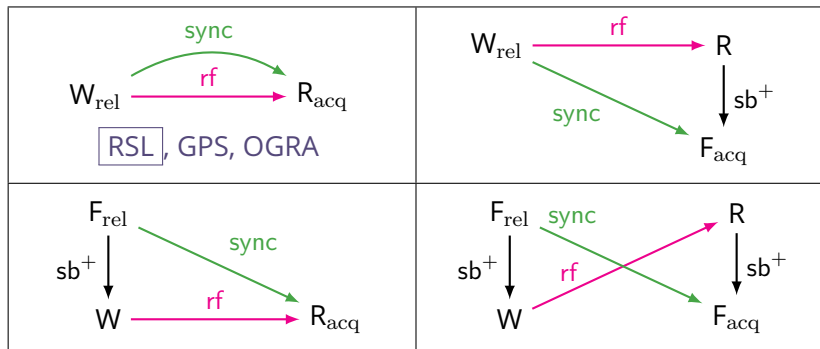


RSL Relaxed Separation Logic (V. Vafeiadis, C. Narayan; OOPSLA '13)

GPS Ghosts, Protocols, and Separation (A. Turon, V. Vafeiadis, D. Dreyer; OOPSLA '14)

OGRA Owiki-Gries for Release-Acquire (O. Lahav, V. Vafeiadis; ICALP '15)

The synchronizes-with relation



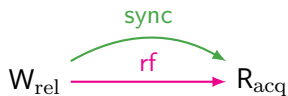
RSL Relaxed Separation Logic (V. Vafeiadis, C. Narayan; OOPSLA '13)

GPS Ghosts, Protocols, and Separation (A. Turon, V. Vafeiadis, D. Dreyer; OOPSLA '14)

OGRA Owicki-Gries for Release-Acquire (O. Lahav, V. Vafeiadis; ICALP '15)

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)



```
int a = 0;
```

```
atomic_int x = 0;
```

```
a = 42;
```

```
x_rel = 1;
```

```
if(x_acq == 1){
```

```
    print(a);
```

```
}
```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

{true}

```
int a = 0;
```

```
atomic_int x = 0;
```

```
a = 42;
```

```
xrel = 1;
```

```
if(xacq == 1){
```

```
    print(a);
```

```
}
```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

```
    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
```

```
a = 42;
```

```
xrel = 1;
```

```
||| if(xacq == 1){
```

```
    print(a);
```

```
||| }
```


Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
a = 42;
x_rel = 1;
|||
if(x_acq == 1){
    print(a);
}

```

$\{Q(v)\}$
atomic_int x = v
 $\{\text{Rel}(x, Q) * \text{Acq}(x, Q)\}$

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    xrel = 1;
    |||
    if(xacq == 1){
        print(a);
    }

```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)}
    x_rel = 1;
    |||
    if(x_acq == 1){
        print(a);
    }
    |||
```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)}
    xrel = 1;
    {Rel(x, Q)}
    {true}
    |||
    if(xacq == 1){
        print(a);
    }

```

$\{\text{Rel}(x, Q) * Q(v)\}$
 $x_{\text{rel}} = v$
 $\{\text{Rel}(x, Q)\}$

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)} || {Acq(x, Q)}
    a = 42; || if(xacq == 1){
    {&a ↦ 42 * Rel(x, Q)} ||
    xrel = 1; ||     print(a);
    {Rel(x, Q)} ||
    {true} ||     }

```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)} || {Acq(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)} || {Acq(x, Q)}
    x_rel = 1;
    {Rel(x, Q)} || {Acq(x, Q)}
    {true} || {Acq(x, Q)}
    }

```

$\{Acq(x, Q)\} t = x_{acq} \{Q(t)\}$

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)} || {Acq(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)} || {&a ↦ 42}
    xrel = 1;
    {Rel(x, Q)} || {true}
    {true} || {print(a);}
    || {true}
    || }
```

Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)} || {Acq(x, Q)}
    a = 42;                    if(xacq == 1){
    {&a ↦ 42 * Rel(x, Q)}      {&a ↦ 42}
    xrel = 1;                  print(a);
    {Rel(x, Q)}                {true}
    {true}                      }
    {true}

```


Relaxed Separation Logic (RSL)

V. Vafeiadis, C. Narayan (OOPSLA 2013)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

$\{true\}$

int a = 0;

- no data races

- memory safety

- no reads of uninitialized locations

$\{\&a \mapsto 0 * R\}$

a = 42;

$\{\&a \mapsto 42 * \text{Rel}(x, Q)\}$

$\{\text{Rel}(x, Q)\}$

$\{true\}$

$\{true\}$

$\{true\}$

$\{\&a \mapsto 42\}$

print(a);

$\{true\}$

$\{true\}$

Fenced Separation Logic (FSL)

```
int a = 0;
```

```
atomic_int x = 0;
```

```
a = 42;
```

```
fencerel;
```

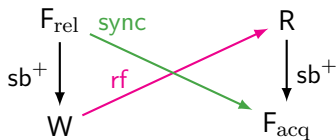
```
xrlx = 1;
```

```
if(xrlx == 1){
```

```
    fenceacq;
```

```
    print(a);
```

```
}
```

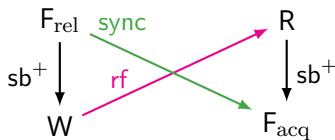


Fenced Separation Logic (FSL)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)}
    fencerel;
    ???
    xrlx = 1;
    {Rel(x, Q)}
    {true}
    |||
    {Acq(x, Q)}
    if (xrlx == 1) {
        ???
        fenceacq;
        {&a ↦ 42}
        print(a);
        {true}
    }
    {true}
  
```



Fenced Separation Logic (FSL)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```

    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)}
    fence_rel;
    {Δ(&a ↦ 42) * Rel(x, Q)}
    x_rlx = 1;
    {Rel(x, Q)}
    {true}
    |||
    {Acq(x, Q)}
    if (x_rlx == 1) {
        ???
        fence_acq;
        {&a ↦ 42}
        print(a);
        {true}
    }
    {true}

```

$\{P\} \text{fence_rel} \{\Delta P\}$

$\{\text{Rel}(x, Q) * \Delta Q(v)\}$
 $x_{\text{rlx}} = v$
 $\{\text{Rel}(x, Q)\}$

Fenced Separation Logic (FSL)

$Q \stackrel{\text{def}}{=} \lambda v. (v = 0 \vee \&a \mapsto 42)$

```
    {true}
    int a = 0;
    {&a ↦ 0}
    atomic_int x = 0;
    {&a ↦ 0 * Rel(x, Q) * Acq(x, Q)}
    {&a ↦ 0 * Rel(x, Q)}
    a = 42;
    {&a ↦ 42 * Rel(x, Q)}
    fence_rel;
    {Δ(&a ↦ 42) * Rel(x, Q)}
    x_rlx = 1;
    {Rel(x, Q)}
    {true}
    |||
    {true}
```

$\{Acq(x, Q)\}$

$t = x_{rlx}$

$\{\nabla Q(t)\}$

$\{\nabla P\} \text{fence}_{acq} \{P\}$

$\{Acq(x, Q)\}$

if ($x_{rlx} == 1$) {

$\{\nabla(\&a \mapsto 42)\}$

 fence_acq;

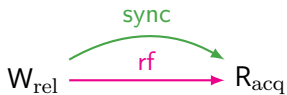
$\{\&a \mapsto 42\}$

 print(a);

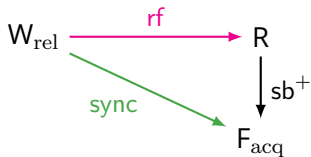
$\{true\}$

}

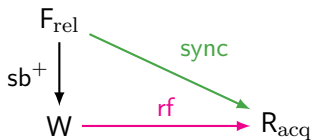
$\{true\}$



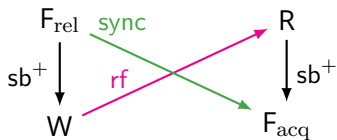
RSL, GPS, OGRA, **FSL**



FSL



FSL



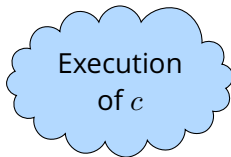
FSL

The semantics of triples

Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?

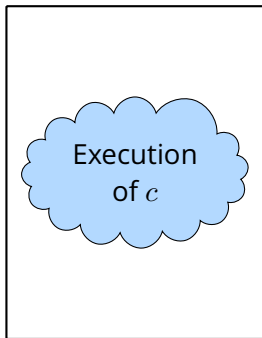
The semantics of triples

Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



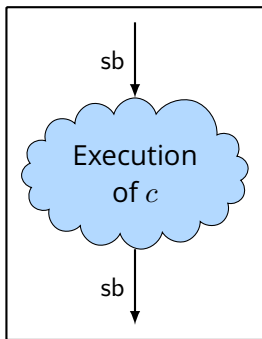
The semantics of triples

Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



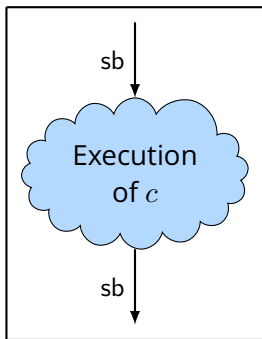
The semantics of triples

Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



The semantics of triples

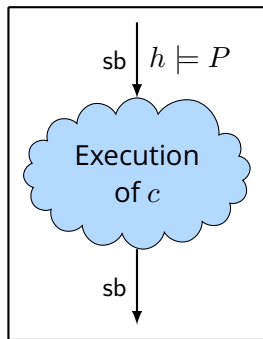
Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



Annotate heaps on sb and rf edges in the execution graph.

The semantics of triples

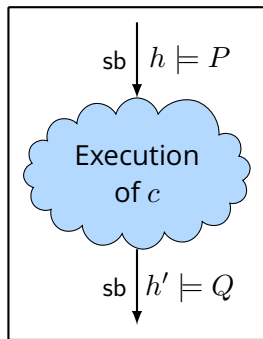
Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



Annotate heaps on sb and rf edges in the execution graph.

The semantics of triples

Without a notion of state, what is the meaning of $\{P\} c \{Q\}$?



Annotate heaps on sb and rf edges in the execution graph.

Local validity

a very simplified example

$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$

Local validity

a very simplified example

$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$

$$W_{\text{rlx}}(\mathbf{x}, 1)$$

Local validity

a very simplified example

$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$

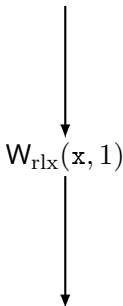


$$\mathbf{W}_{\text{rlx}}(\mathbf{x}, 1)$$

Local validity

a very simplified example

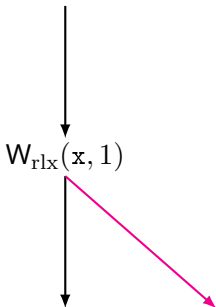
$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$



Local validity

a very simplified example

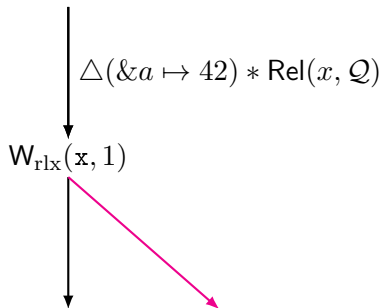
$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$



Local validity

a very simplified example

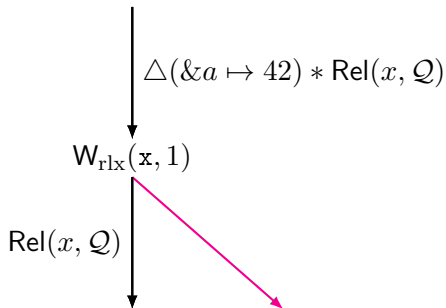
$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$



Local validity

a very simplified example

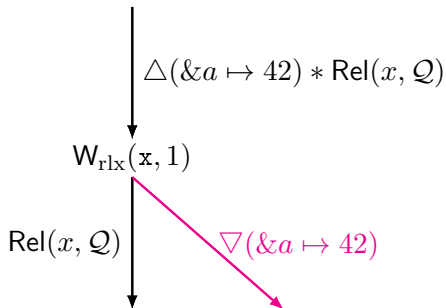
$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$



Local validity

a very simplified example

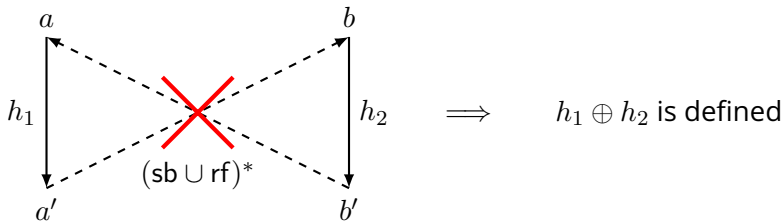
$$\{\Delta(\&a \mapsto 42) * \text{Rel}(x, Q)\} \mathbf{x}_{\text{rlx}} = 1 \{\text{Rel}(x, Q)\}$$



Independent heap compatibility

Definition (Independent edges)

A set of edges \mathcal{T} in an execution graph is *pairwise independent* if for all $(a, a'), (b, b') \in \mathcal{T}$, we have $\neg(\text{sb} \cup \text{rf})^*(a', b)$.



Lemma (Independent heap compatibility)

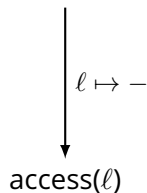
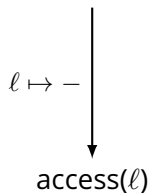
For every validly annotated execution, and pairwise independent set of edges \mathcal{T} , heaps annotated on edges in \mathcal{T} are combinable.

Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).

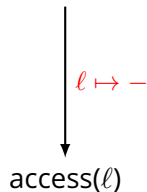
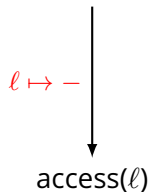
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Theorem

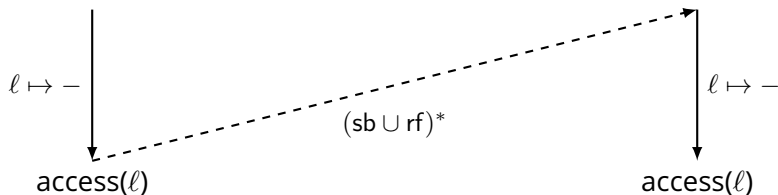
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

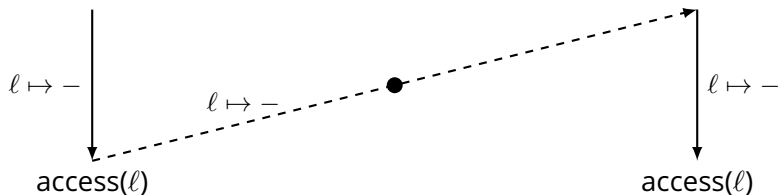
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



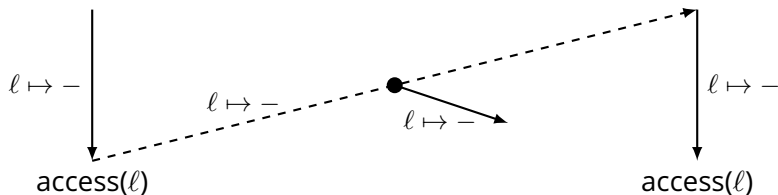
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Theorem

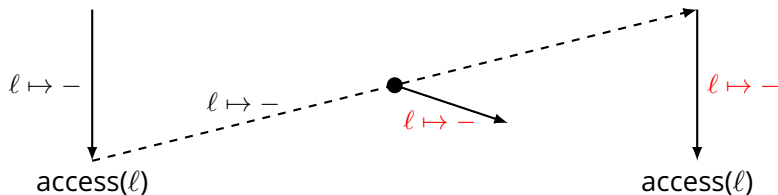
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

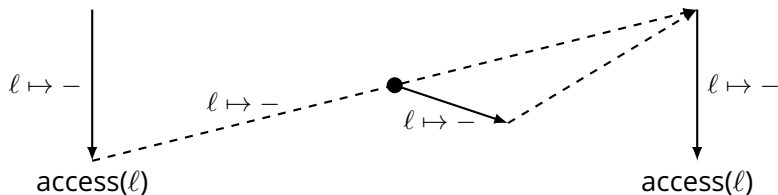
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Theorem

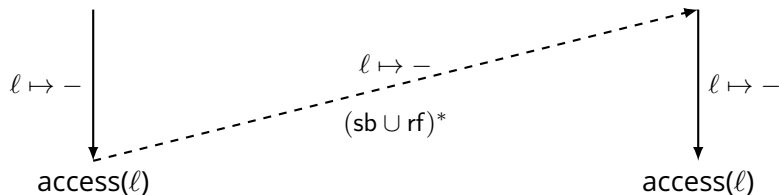
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

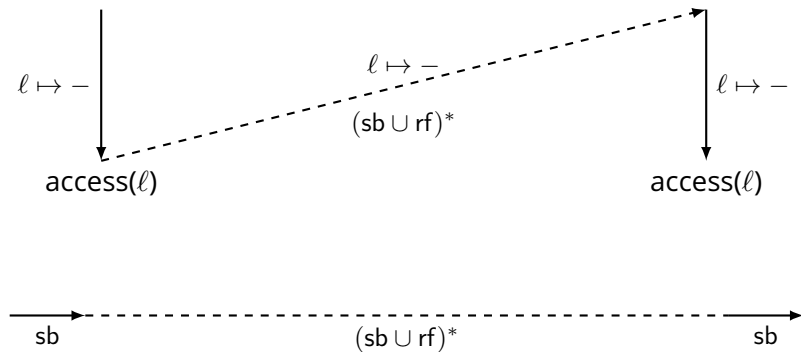
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

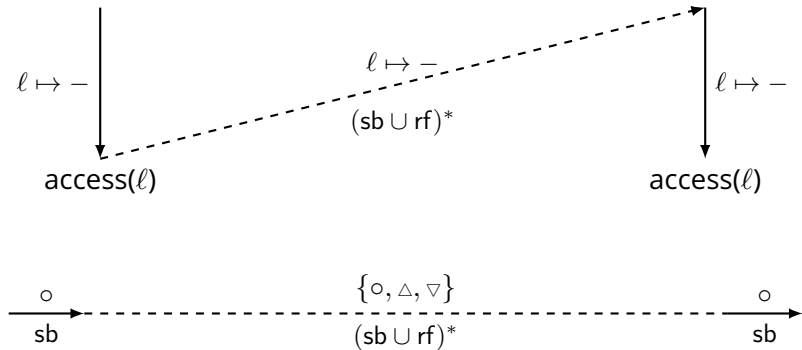
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

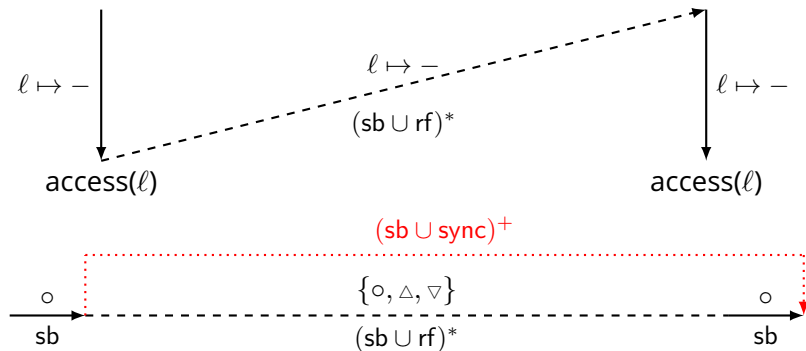
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

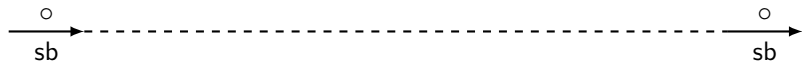
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



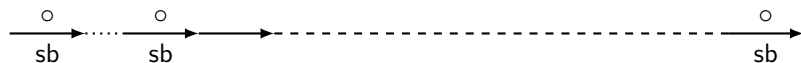
Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Theorem

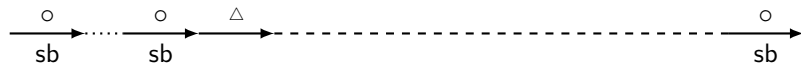
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

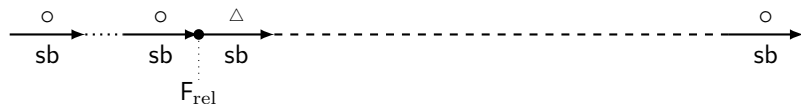
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

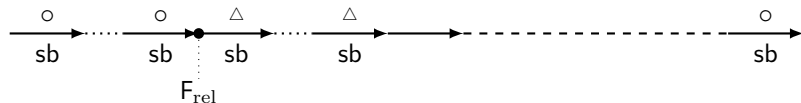
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

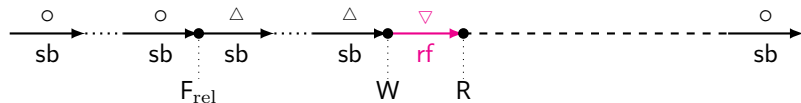
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

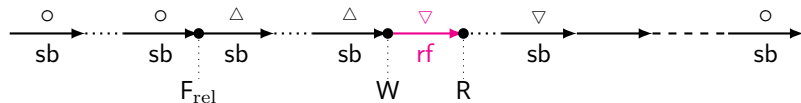
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

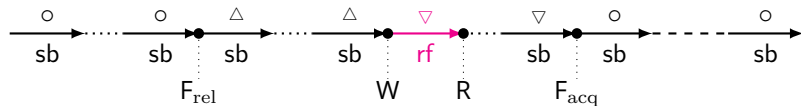
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

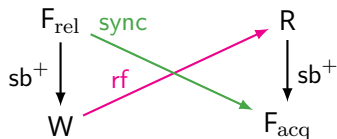
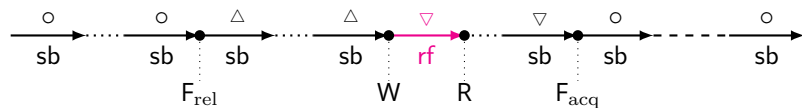
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

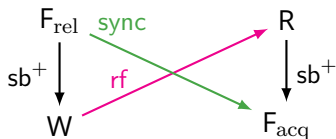
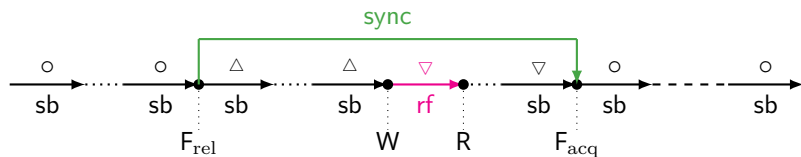
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

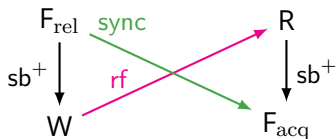
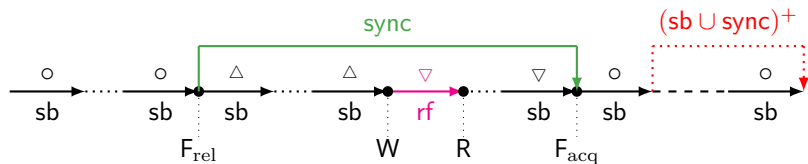
In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Data race freedom

Theorem

In a validly annotated execution, any two non-atomic accesses to the same location are ordered by $(sb \cup \text{sync})^+$ (i.e. they are not racing).



Summary and future work

Summary:

- FSL is the first logic that supports C11-style memory fences.
- FSL ensures
 - data race freedom,
 - memory safety, and
 - all reads read from initialized locations.
- Soundness proof is formalized in Coq:
<http://plv.mpi-sws.org/fsl/>



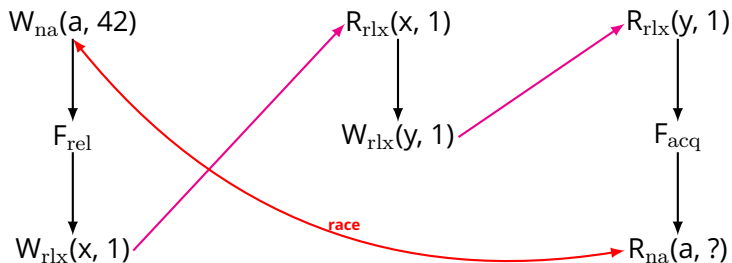
Future work:

- Support for CAS instructions and fractional permissions.
- Verify real-world algorithms (such as Rust's Arc).

Why the two modalities?

$Q \stackrel{\text{def}}{=} \lambda v. \text{if } v = 0 \text{ then emp else } a \mapsto 42$

$\{a \mapsto 0 * \text{Rel}(x, Q)\}$	$\{\text{Acq}(x, Q)\}$	$\{\text{Acq}(y, Q)\}$
<code>a = 42;</code>	<code>while(x_{rlx} == 0);</code>	<code>while(y_{rlx} == 0);</code>
$\{a \mapsto 42 * \text{Rel}(x, Q)\}$	$\{\diamond a \mapsto 42\}$	$\{\diamond a \mapsto 42\}$
<code>fence_{rel};</code>	<code>\{ \diamond a \mapsto 42 \}</code>	<code>fence_{acq};</code>
$\{\diamond a \mapsto 42 * \text{Rel}(x, Q)\}$	<code>y_{rlx} = 1;</code>	$\{a \mapsto 42\}$
<code>x_{rlx} = 1;</code>	$\{\text{true}\}$	<code>print(a);</code>
$\{\text{true}\}$		$\{a \mapsto 42\}$



Some important properties of FSL assertions

- Release permissions are duplicable:

$$\text{Rel}(\ell, Q) \iff \text{Rel}(\ell, Q) * \text{Rel}(\ell, Q)$$

- Acquire permissions are splittable:

$$\text{Acq}(\ell, Q_1) * \text{Acq}(\ell, Q_2) \iff \text{Acq}(\ell, \lambda v. Q_1(v) * Q_2(v))$$

- Modalities (Δ and ∇) distribute over disjunction, conjunction, and separating conjunction:

$$\begin{array}{ll} \Delta(P \wedge Q) \iff \Delta P \wedge \Delta Q & \nabla(P \wedge Q) \iff \nabla P \wedge \nabla Q \\ \Delta(P \vee Q) \iff \Delta P \vee \Delta Q & \nabla(P \vee Q) \iff \nabla P \vee \nabla Q \\ \Delta(P * Q) \iff \Delta P * \Delta Q & \nabla(P * Q) \iff \nabla P * \nabla Q \end{array}$$