On the C11 Memory Model and a Program Logic for C11 Concurrency

Manuel Penschuck

Aktuelle Themen der Softwaretechnologie
Software Engineering & Programmiersprachen – FB 12
Goethe Universität - Frankfurt am Main

04. Feb. 2014

Topics

Common Weakness Enumeration[1]

"TOP 25 Most Dangerous Software Errors"

(41) CWE-456: Missing Initialisation

(40) CWE-825: Expired Pointer Dereference

(36) CWE-476: NULL Pointer Dereference

(33) CWE-362: Concurrent Execution using Shared Resource with Improper Synchronisation ('Race Condition')

```
int init = 1;
int uninit;
int result
      = init + uninit;
```

[1] http://cwe.mitre.org/top25/archive/2011/2011_onthecusp.html

Common Weakness Enumeration[1]

"TOP 25 Most Dangerous Software Errors"

(41) CWE-456: Missing Initialisation

(40) CWE-825: Expired Pointer Dereference

(36) CWE-476: NULL Pointer Dereference

(33) CWE-362: Concurrent Execution using Shared Resource with Improper Synchronisation ('Race Condition')

```
free(p);
*p = 47;
```

[1] http://cwe.mitre.org/top25/archive/2011/2011_onthecusp.html

Common Weakness Enumeration[1]

"TOP 25 Most Dangerous Software Errors"

(41) CWE-456: Missing Initialisation

(40) CWE-825: Expired Pointer Dereference

(36) CWE-476: NULL Pointer Dereference

(33) CWE-362: Concurrent Execution using Shared Resource with Improper Synchronisation ('Race Condition')

```
p = malloc(sizeof(int));
  // out of memory !
*p = 42;
```

[1] http://cwe.mitre.org/top25/archive/2011/2011_onthecusp.html

Common Weakness Enumeration[1]

"TOP 25 Most Dangerous Software Errors"

(41) CWE-456: Missing Initialisation

(40) CWE-825: Expired Pointer Dereference

(36) CWE-476: NULL Pointer Dereference

(33) CWE-362: Concurrent Execution using Shared Resource with Improper Synchronisation ('Race Condition')

```
tmp = shared;
tmp = tmp&1
          ? 3*tmp +1
          :   tmp>>1;
shared = tmp;
```

[1] http://cwe.mitre.org/top25/archive/2011/2011_onthecusp.html

Hidden Races

```
int data = 0;

void worker() {
    for(int i=0; i < N; i++) {
        data++;
    }
}

int main(..) {
    pardo [worker() worker()] and join;
    printf("data=%d", data);
}
```

Hidden Races

```
void worker() { // gcc -O0
    for(int i=0; i < N; i++) {
        register int tmp = data; // RAM: expensive
        tmp++;
        data = tmp;              // RAM: expensive
    }
}
```

```
void worker() { // gcc -O1
    register int tmp = data;
    for(int i=0; i < N; i++)
        tmp++;

    data = tmp;
}
```

```
void worker() { // gcc -O3
    register int tmp = data;
    tmp += N;
    data = tmp;
}
```

Situation until now

- C99/C0X standard: purely sequential
- Thread and synchronisation support via libraries
  (PThread, TBB, MS Windows Runtime Lib, C++/BOOST, ...)
- Limited compiler support
  - Optimisation unsound in concurrent programs?
  - Synchronisation inefficient (`__sync()` in GCC)

Situation until now

- C99/C0X standard: purely sequential
- Thread and synchronisation support via libraries
  (PThread, TBB, MS Windows Runtime Lib, C++/BOOST, ...)
- Limited compiler support
  - Optimisation unsound in concurrent programs?
  - Synchronisation inefficient (`__sync()` in GCC)

Situation until now

- C99/C0X standard: purely sequential
- Thread and synchronisation support via libraries
  (PThread, TBB, MS Windows Runtime Lib, C++/BOOST, ...)
- Limited compiler support
  - Optimisation unsound in concurrent programs?
  - Synchronisation inefficient (`__sync()` in GCC)

Some Extensions of C11

- Optional concurrency mode
- Threads via `threads.h`
  similar to POSIX threads (thread management, mutex, barrier)
- Atomic Memory Access via `stdatomic.h`
- Memory Model!

Some Extensions of C11

- Optional concurrency mode
- Threads via `threads.h`
  similar to POSIX threads (thread management, mutex, barrier)
- Atomic Memory Access via `stdatomic.h`
- Memory Model!

Some Extensions of C11

- Optional concurrency mode
- Threads via `threads.h`
  similar to POSIX threads (thread management, mutex, barrier)
- Atomic Memory Access via `stdatomic.h`
- Memory Model!

Some Extensions of C11

- Optional concurrency mode
- Threads via `threads.h`
  similar to POSIX threads (thread management, mutex, barrier)
- Atomic Memory Access via `stdatomic.h`
- Memory Model!

C11 Memory Model

- Extends existing “evaluation rules”
- Allow compiler and processor as many optimisations as possible
- Support for
  - Sequential consistency
  - Acquire-Release
  - Consume-Release
  - Relaxed (just atomic)

C11 Memory Model

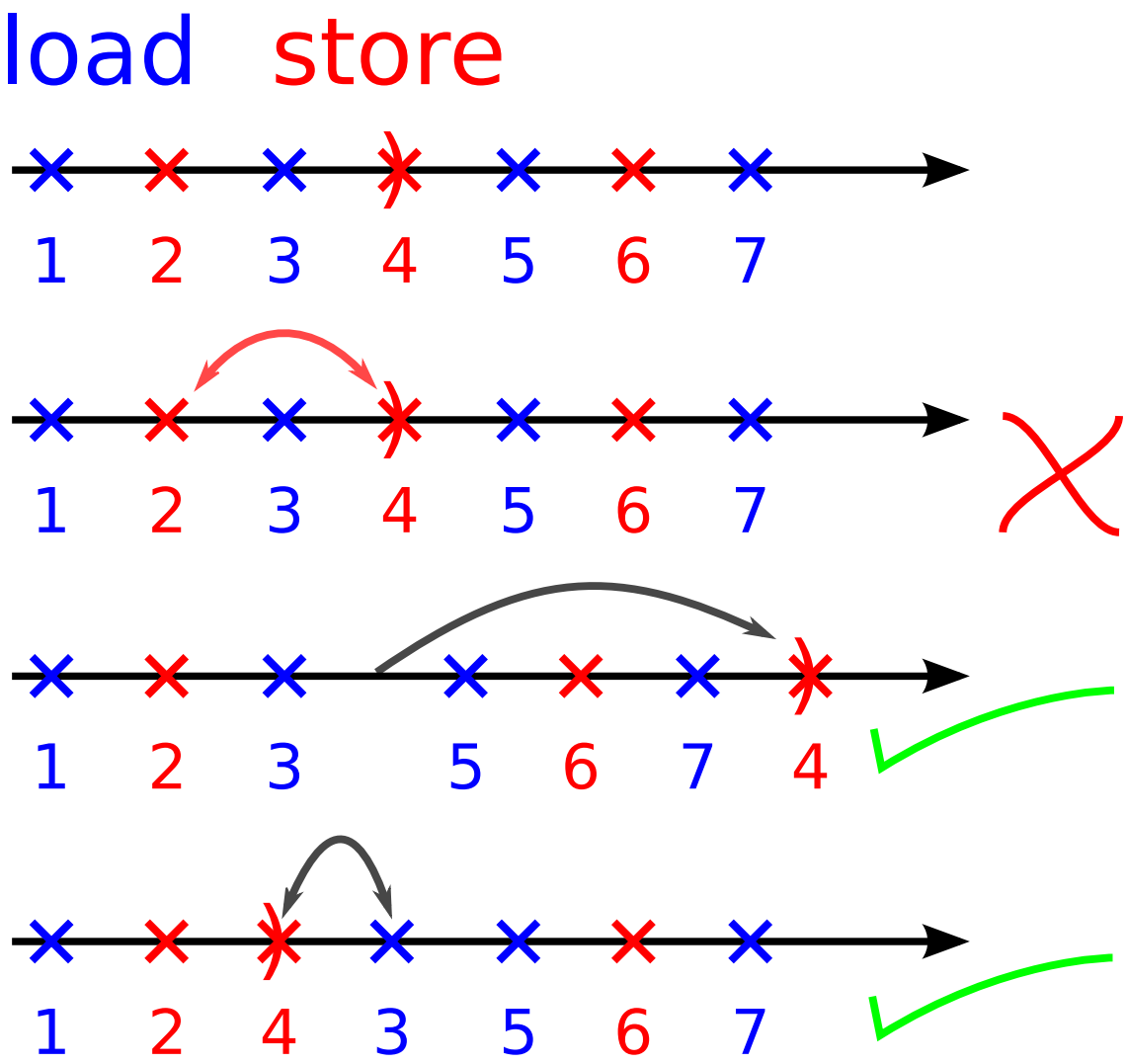
- Extends existing “evaluation rules”
- Allow compiler and processor as many optimisations as possible
- Support for
  - Sequential consistency
  - Acquire-Release
  - Consume-Release
  - Relaxed (just atomic)

C11 Memory Model

- Extends existing “evaluation rules”
- Allow compiler and processor as many optimisations as possible
- Support for
  - Sequential consistency
  - Acquire-Release
  - Consume-Release
  - Relaxed (just atomic)

Acquire-Release

- All effects of instructions preceding a release R have be observable before R
- All reads succeeding an acquire A have to happen after A
- Very efficient on x86 (plain reads/writes)



Acquire-Release

- All effects of instructions preceding a release R have be observable before R
- All reads succeeding an acquire A have to happen after A
- Very efficient on x86 (plain reads/writes)

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}(); \\
r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}();\ [f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ r \leftarrow [v]_{\mathsf{na}} \end{array} \right)
\end{array}
$$

Acquire-Release

- All effects of instructions preceding a release R have be observable before R
- All reads succeeding an acquire A have to happen after A
- Very efficient on x86 (plain reads/writes)

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}(); \\
r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}();\ [f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ r \leftarrow [v]_{\mathsf{na}} \end{array} \right)
\end{array}
$$

stdatomic.h

- `atomic_load`, `-store`
- `atomic_fetch_add`, `-sub`, `-or`, `-xor`, `-and`
- `atomic_exchange`
- `atomic_compare_exchange_weak`, `-strong`
- `atomic_flag_clear`, `-test_and_set` ...

stdatomic.h

- atomic_load, -store
- atomic_fetch_add, -sub, -or, -xor, -and
- atomic_exchange
- atomic_compare_exchange_weak, -strong
- atomic_flag_clear, -test_and_set ...

```
for(int i=0; i<N; i++) {
//  data++;
  atomic_fetch_add(&data, 1);
}
```

stdatomic.h

- atomic_load, -store
- atomic_fetch_add, -sub, -or, -xor, -and
- atomic_exchange
- atomic_compare_exchange_weak, -strong
- atomic_flag_clear, -test_and_set ...

```
atomic_exchange(*p, val){
  old = *p;

  *p = val;
  return old;
}
```

stdatomic.h

- atomic_load, -store
- atomic_fetch_add, -sub, -or, -xor, -and
- atomic_exchange
- atomic_compare_exchange_weak, -strong
- atomic_flag_clear, -test_and_set ...

```
atomic_compare_exchange(*p,ref,val){
  old = *p;
  if (old == ref) {
    *p = val;
    return true;
  }

  return false;
}

spinlock(*p, threadId) {
  while(at_com_xchg_weak(
            p, 0, threadId));
}
```

stdatomic.h

- `atomic_load`, `-store`
- `atomic_fetch_add`, `-sub`, `-or`, `-xor`, `-and`
- `atomic_exchange`
- `atomic_compare_exchange_weak`, `-strong`
- `atomic_flag_clear`, `-test_and_set` ...

Program Logics

The Hoare Triple

$$(P)\ \{Q\}\ (R)$$

If the assertion P is true before [the] initiation of a program Q, then the assertion R will be true on its completion [if the program terminates].

$$(x \mapsto 1)\ \{x \leftarrow x + 1\}\ (x \mapsto 2)$$

$$(x \mapsto 1)\ \{\texttt{while}\ (\texttt{true});\}\ (x \mapsto 2)$$

The Hoare Triple

$$(P)\ \{Q\}\ (R)$$

If the assertion P is true before [the] initiation of a program Q, then the assertion R will be true on its completion [if the program terminates].

$$(x \mapsto 1)\ \{x \leftarrow x + 1\}\ (x \mapsto 2)$$

$$(x \mapsto 1)\ \{\texttt{while}\,(\texttt{true});\}\ (x \mapsto 2)$$

The Hoare Triple

$$(P)\ \{Q\}\ (R)$$

If the assertion P is true before [the] initiation of a program Q, then the assertion R will be true on its completion [if the program terminates].

$$(x \mapsto 1)\ \{x \leftarrow x + 1\}\ (x \mapsto 2)$$

$$(x \mapsto 1)\ \{\texttt{while (true);}\}\ (x \mapsto 2)$$

The Hoare Triple

$$(P)\ \{Q\}\ (R)$$

If the assertion P is true before [the] initiation of a program Q, then the assertion R will be true on its completion [if the program terminates].

$$(x \mapsto 1)\ \{x \leftarrow x + 1\}\ (x \mapsto 2)$$

$$(x \mapsto 1)\ \{\texttt{while (true);}\}\ (x \mapsto 2)$$

Hoare Logic: "Rule of Composition"

$$\frac{(P)\ \{Q_1\}\ (R) \qquad (R)\ \{Q_2\}\ (T)}{(P)\ \{Q_1;\ Q_2\}\ (T)}$$

$$\frac{(x \mapsto 1)\ \{x \leftarrow x+1\}\ (x \mapsto 2) \qquad (x \mapsto 2)\ \{y \leftarrow x\}\ (x \mapsto 2 \wedge y \mapsto 2)}{(x \mapsto 1)\ \{x \leftarrow x+1;\ y \leftarrow x\}\ (x \mapsto 2 \wedge y \mapsto 2)}$$

Hoare Logic: "Rule of Composition"

$$\frac{(P)\ \{Q_1\}\ (R) \qquad (R)\ \{Q_2\}\ (T)}{(P)\ \{Q_1;\ Q_2\}\ (T)}$$

$$\frac{(x \mapsto 1)\ \{x \leftarrow x+1\}\ (x \mapsto 2) \qquad (x \mapsto 2)\ \{y \leftarrow x\}\ (x \mapsto 2 \wedge y \mapsto 2)}{(x \mapsto 1)\ \{x \leftarrow x+1;\ y \leftarrow x\}\ (x \mapsto 2 \wedge y \mapsto 2)}$$

Separation Logic

- Targets shared data structures (pointers, references)
- Efficient fully-automated verifier
- Distinguishes between
  - *Registers* represented by the Stack
    $s : \text{Vars} \mapsto \text{Values}$ with $\text{dom}(s) < \infty$
  - *Addressable Memory* represented by the Heap
    $h : \text{Addresses} \mapsto \text{Values}$ with $\text{dom}(h) < \infty$
- Separates independent code sections using the separation conjunction $*$

Separation Logic

- Targets shared data structures (pointers, references)
- Efficient fully-automated verifier
- Distinguishes between
  - *Registers* represented by the Stack
    $s : \text{Vars} \mapsto \text{Values}$ with $\text{dom}(s) < \infty$
  - *Addressable Memory* represented by the Heap
    $h : \text{Addresses} \mapsto \text{Values}$ with $\text{dom}(h) < \infty$
- Separates independent code sections using the separation conjunction $*$

Separation Logic

- Targets shared data structures (pointers, references)
- Efficient fully-automated verifier
- Distinguishes between
  - *Registers* represented by the Stack
    $s : \text{Vars} \mapsto \text{Values}$ with $\text{dom}(s) < \infty$
  - *Addressable Memory* represented by the Heap
    $h : \text{Addresses} \mapsto \text{Values}$ with $\text{dom}(h) < \infty$
- Separates independent code sections using the separation conjunction $*$

Separation Logic

- Targets shared data structures (pointers, references)
- Efficient fully-automated verifier
- Distinguishes between
  - *Registers* represented by the Stack
    $s : \mathrm{Vars} \mapsto \mathrm{Values}$ with $\mathrm{dom}(s) < \infty$
  - *Addressable Memory* represented by the Heap
    $h : \mathrm{Addresses} \mapsto \mathrm{Values}$ with $\mathrm{dom}(h) < \infty$
- Separates independent code sections using the separation conjunction $*$

Separation Logic: "Framing Rule"

$$\frac{(p)\ \{Q_1\}\ (q)}{(p * r)\ \{Q_1\}\ (q * r)}$$

$$(x \neq y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$$

Separation Logic: "Framing Rule"

$$\frac{(p)\ \{Q_1\}\ (q)}{(p * r)\ \{Q_1\}\ (q * r)}$$

$$(x \neq y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$$

Separation Logic: "Framing Rule"

$$\frac{(p)\ \{Q_1\}\ (q)}{(p * r)\ \{Q_1\}\ (q * r)}$$

$$(x \neq y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$$

$$(x = y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$$

Separation Logic: "Framing Rule"

$$\frac{(p)\ \{Q_1\}\ (q)}{(p * r)\ \{Q_1\}\ (q * r)}$$

$$(x \neq y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$$

~~$(x = y \wedge (x \mapsto 2 * y \mapsto 2))\ \{[x] \leftarrow 3\}\ (x \mapsto 3 * y \mapsto 2)$~~

Relaxed Separation Logic

- A (complete) C-language subset
- Rules to infer a Hoare-Triple annotation
- Central Theorem (Coq-proven):
  If annotation is successful $\Rightarrow$ program is memory sound

Relaxed Separation Logic

$$P, Q \quad := \quad \overbrace{\mathbf{false} \mid P \Rightarrow Q \mid \forall x.\, P}^{\text{complete first order logic}} \mid \overbrace{\mathbf{emp} \mid e \overset{k}{\mapsto} e' \mid P * Q}^{\text{known from SL}}$$
$$\mid \mathrm{Rel}(e, \mathcal{Q}) \mid \mathrm{Acq}(e, \mathcal{Q}) \mid \mathrm{RMWAcq}(e, \mathcal{Q})$$
$$\mid \mathrm{Init}(e) \mid \mathrm{Uninit}(e)$$

Relaxed Separation Logic

$$P, Q \quad := \quad \overbrace{\mathbf{false} \mid P \Rightarrow Q \mid \forall x.\, P}^{\text{complete first order logic}} \mid \overbrace{\mathbf{emp} \mid e \overset{k}{\mapsto} e' \mid P * Q}^{\text{known from SL}}$$

$$\mid \mathrm{Rel}(e, \mathcal{Q}) \mid \mathrm{Acq}(e, \mathcal{Q}) \mid \mathrm{RMWAcq}(e, \mathcal{Q})$$

$$\mid \mathrm{Init}(e) \mid \mathrm{Uninit}(e)$$

Relaxed Separation Logic

$$P, Q \quad := \quad \overbrace{\mathbf{false} \mid P \Rightarrow Q \mid \forall x.\, P}^{\text{complete first order logic}} \mid \overbrace{\mathbf{emp} \mid e \overset{k}{\mapsto} e' \mid P * Q}^{\text{known from SL}}$$
$$\mid \mathsf{Rel}(e, \mathcal{Q}) \mid \mathsf{Acq}(e, \mathcal{Q}) \mid \mathsf{RMWAcq}(e, \mathcal{Q})$$
$$\mid \mathsf{Init}(e) \mid \mathsf{Uninit}(e)$$

Relaxed Separation Logic

$$\textbf{(emp)}$$

$$v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}();$$

$$f \leftarrow \mathsf{alloc}();$$

$$[f]_{\mathsf{rlx}} \leftarrow 0;$$

$$\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array}\right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array}\right)$$

Relaxed Separation Logic

$$(\mathbf{emp})$$

$$v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}();$$

$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r))$$

$$f \leftarrow \mathsf{alloc}();$$

$$[f]_{\mathsf{rlx}} \leftarrow 0;$$

$$\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array}\right) \Big\| \left(\begin{array}{l} \mathbf{repeat}\, [f]_{\mathsf{acq}}\, \mathbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array}\right)$$

Non-Atomic Allocation

$$(\mathbf{emp})\ \{\mathsf{alloc}()\}\ (x.\, \mathsf{Uninit}(x))$$

Relaxed Separation Logic

$$v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}();$$
$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r))$$
$$f \leftarrow \mathsf{alloc}();$$
$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r) * \mathsf{Rel}(f, \mathcal{Q}) * \mathsf{Acq}(f, \mathcal{Q}))$$
$$[f]_{\mathsf{rlx}} \leftarrow 0;$$
$$\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array} \right)$$

Atomic Allocation

$$(\textbf{emp})\ \{\mathsf{alloc}()\}\ (x.\, \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}))$$

Relaxed Separation Logic

$$v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}();$$
$$f \leftarrow \mathsf{alloc}();$$
$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r) * \mathsf{Rel}(f, \mathcal{Q}) * \mathsf{Acq}(f, \mathcal{Q}))$$
$$[f]_{\mathsf{rlx}} \leftarrow 0;$$
$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r) * \mathsf{Rel}(f, \mathcal{Q}) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f))$$
$$\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array} \right)$$

Relaxed Write (simplified)

$$(\,\mathsf{Rel}(..))\ \{[l] \leftarrow v\}\ (\,\mathsf{Init}(l))$$

Relaxed Separation Logic

$$v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}();$$
$$f \leftarrow \mathsf{alloc}();$$
$$[f]_{\mathsf{rlx}} \leftarrow 0;$$
$$(\mathsf{Uninit}(v) * \mathsf{Uninit}(r) * \mathsf{Rel}(f, \mathcal{Q}) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f))$$
$$\left(\begin{array}{l}(\mathsf{Uninit}(v) * \mathsf{Rel}(f, \mathcal{Q})) \\ [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1\end{array}\right) \Bigg\| \left(\begin{array}{l}(\mathsf{Uninit}(r) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f)) \\ \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}}\end{array}\right)$$

Parallel Composition

$$\frac{(P_1)\ \{Q_1\}\ (R_1) \qquad (P_2)\ \{Q_2\}\ (R_2)}{(P_1 * P_2)\ \{Q_1 || Q_2\}\ (R_1 * R_2)}$$

Relaxed Separation Logic

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}(); \\
[f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} (\mathsf{Uninit}(v) * \mathsf{Rel}(f, \mathcal{Q})) \\ [v]_{\mathsf{na}} \leftarrow 47 \\ (v \mapsto 47 * \mathsf{Rel}(f, \mathcal{Q})) \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Bigg\| \left(\begin{array}{l} (\mathsf{Uninit}(r) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f)) \\ \textbf{repeat}\ [f]_{\mathsf{acq}}\ \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array} \right)
\end{array}
$$

Non-Atomic Write

$$
\left(l \overset{1}{\mapsto} \_ \vee \mathsf{Uninit}(l)\right)\ \left\{[l]_{\mathsf{na}} \leftarrow v\right\}\ \left(l \overset{1}{\mapsto} v\right)
$$

Relaxed Separation Logic

$$v \leftarrow \mathsf{alloc}();\ r \leftarrow \mathsf{alloc}();$$
$$f \leftarrow \mathsf{alloc}();$$
$$[f]_{\mathsf{rlx}} \leftarrow 0;$$

$$\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ (\underbrace{v \mapsto 47}_{\mathcal{Q}} * \mathsf{Rel}(f, \mathcal{Q})) \\ [f]_{\mathsf{rel}} \leftarrow 1 \\ (\mathsf{Rel}(f, \mathcal{Q})) \end{array}\right) \Big\| \left(\begin{array}{l} (\mathsf{Uninit}(r) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f)) \\ \textbf{repeat}\ [f]_{\mathsf{acq}}\ \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array}\right)$$

Atomic Write

$$(\mathcal{Q}(v) * \mathsf{Rel}(l, \mathcal{Q}))\ \{[l]_{\mathsf{rel}} \leftarrow v\}\ (\mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}))$$

Relaxed Separation Logic

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}();\ r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}(); \\
[f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \\ (\mathsf{Rel}(f, \mathcal{Q})) \end{array} \right) \Big\| \left(\begin{array}{l} (\mathsf{Uninit}(r) * \mathsf{Acq}(f, \mathcal{Q}) * \mathsf{Init}(f)) \\ \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ (\underbrace{v \overset{1}{\mapsto} 47}_{\mathcal{Q}} * \mathsf{Uninit}(r) * \textbf{true}) \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \end{array} \right)
\end{array}
$$

Atomic Read (simplified)

$$
\left(\mathsf{Init}(l) * \mathsf{Acq}(l, \mathcal{Q}) \right) \left\{ [l]_{\mathsf{acq}} \right\} \left(v. \mathcal{Q}(v) * \mathsf{Acq}(\ldots) \right)
$$

Relaxed Separation Logic

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}();\ r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}(); \\
[f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ (\underbrace{v \overset{1}{\mapsto} 47}_{\mathcal{Q}} * \mathsf{Uninit}(r) * \textbf{true}) \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \\ (\textbf{true} * r \mapsto 47 * v \mapsto 47) \end{array} \right)
\end{array}
$$

Non-Atomic Read

$$(l \overset{k}{\mapsto} v)\ \{[l]_{\mathsf{na}}\}\ (x.x = v \wedge l \overset{k}{\mapsto} v)$$

Relaxed Separation Logic

$$
\begin{array}{c}
v \leftarrow \mathsf{alloc}();\ \ r \leftarrow \mathsf{alloc}(); \\
f \leftarrow \mathsf{alloc}(); \\
[f]_{\mathsf{rlx}} \leftarrow 0; \\
\left(\begin{array}{l} [v]_{\mathsf{na}} \leftarrow 47 \\ [f]_{\mathsf{rel}} \leftarrow 1 \\ (\mathsf{Rel}(f, \mathcal{Q})) \end{array} \right) \Big\| \left(\begin{array}{l} \textbf{repeat}\, [f]_{\mathsf{acq}}\, \textbf{end} \\ [r]_{\mathsf{na}} \leftarrow [v]_{\mathsf{na}} \\ (\textbf{true} * r \mapsto 47 * v \mapsto 47) \end{array} \right) \\
(\textbf{true} * r \mapsto 47)
\end{array}
$$

Parallel Composition

$$
\frac{(P_1)\ \{Q_1\}\ (R_1) \qquad (P_2)\ \{Q_2\}\ (R_2)}{(P_1 * P_2)\ \{Q_1 || Q_2\}\ (R_1 * R_2)}
$$

Conclusion & Outlook

- C11 is now officially fit for concurrent programs
- RSL shows how to perform formal arguments on (subset) of C11-MM
- Fully automated verifier?

Thank you for your attention!

Questions?

Conclusion & Outlook

- C11 is now officially fit for concurrent programs
- RSL shows how to perform formal arguments on (subset) of C11-MM
- Fully automated verifier?

Thank you for your attention!
Questions?