Starling

Cheap Automatic Verification of Low-Level Concurrency

Matt Windsor

Research Seminar 2018-04-19

based on work with Mike Dodds, Matthew J. Parkinson, and Ben Simner

How do we prove safety properties of programs?

Hoare logic

{precondition} `command` **{postcondition}**

{P} `C` **{Q}** **+** **{P}** `D` **{R}** **=** **{P}** `C;D` **{R}**

(C.A.R. Hoare, ‘An Axiomatic Basis for Computer Programming’, 1969)

Hoare logic proof outlines

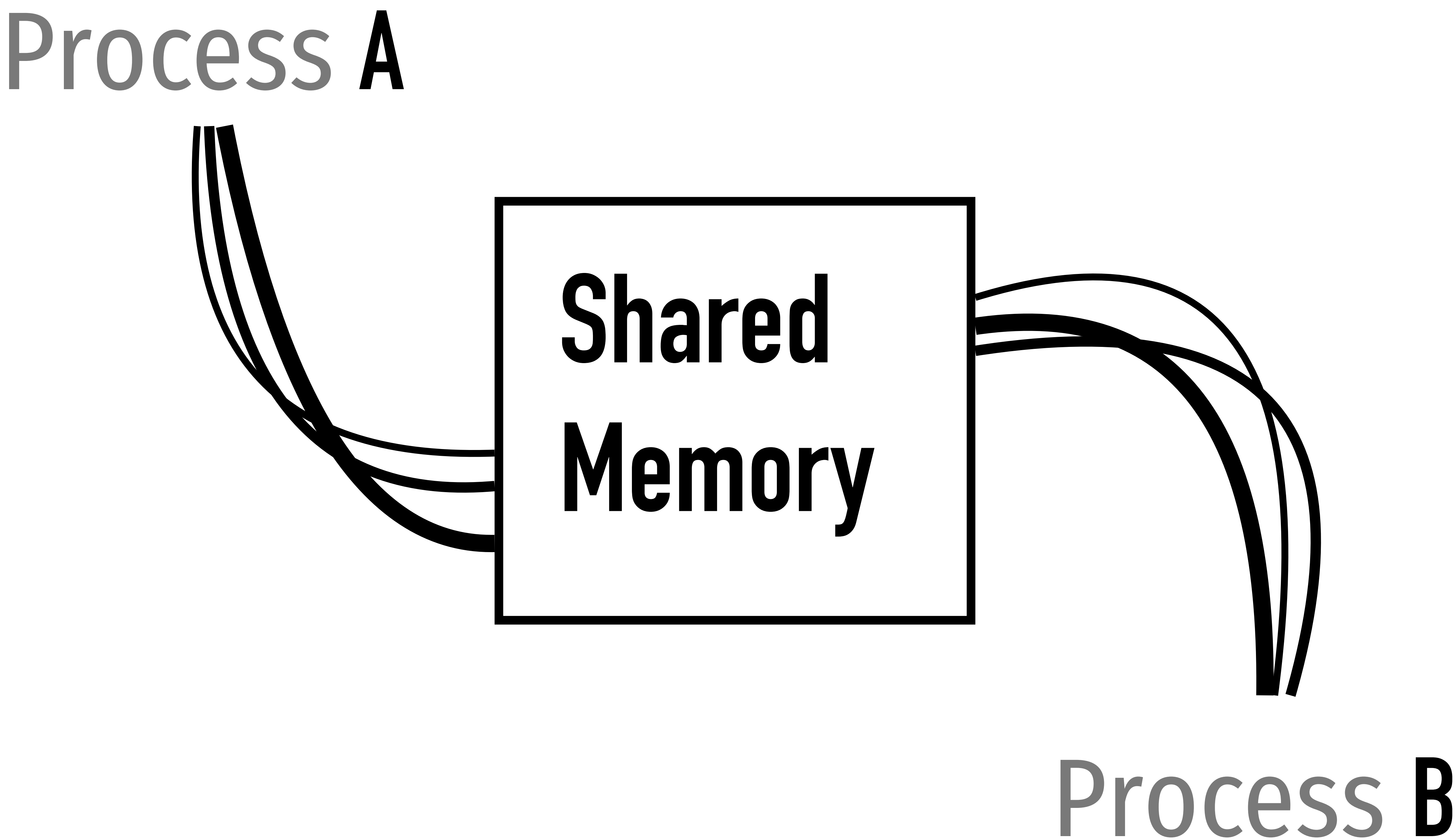
```
{true}
{
  {x = x + y*0}
    r = x;
  {x = r + y*0}
    q = 0;
  {x = r + y*q}
    while (y <= r) {
      {x = (r - y) + y*(1+q)}
        r -= y;
      {x = r + y*(1+q)}
        q++;
      {x = r + y*q}
    }
  {y > r && x = r + y*q}
}
{y > x = r + y*q}
```

$$\begin{tikzcd}[column sep=huge, row sep=huge] \text{Sequential proof} \arrow[r, "\text{Hoare logic laws}"'] & \text{Hoare triples} \arrow[d, "\text{General solver}"] \\ & \text{Result} \end{tikzcd}$$

What about concurrent programs?

We call a program, or algorithm, ‘concurrent’ if

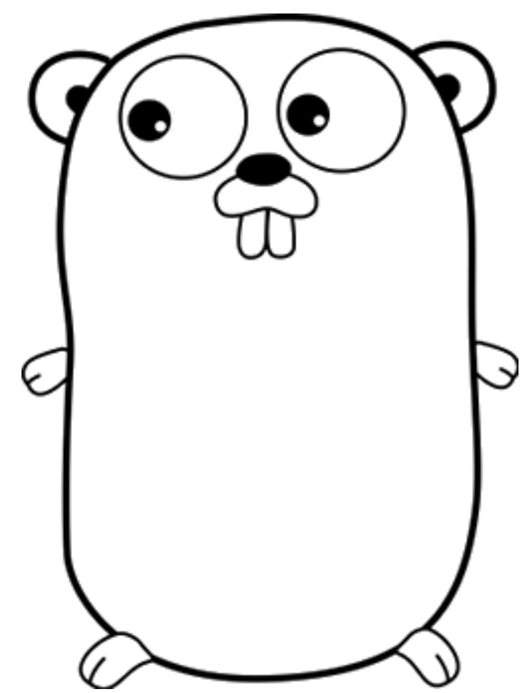
we can run it as two, or more, simultaneous processes.


Process A
Shared Memory
Process B

Shared memory problems

- Process A reads value in the middle of Process B editing it.
- Process A overwrites Process B’s data.
- Process A and B race each other to set some variable.

Communicate to share memory, don't share memory to communicate?



High-level abstraction

↓

Fast, risky shared memory implementation

Atomic actions

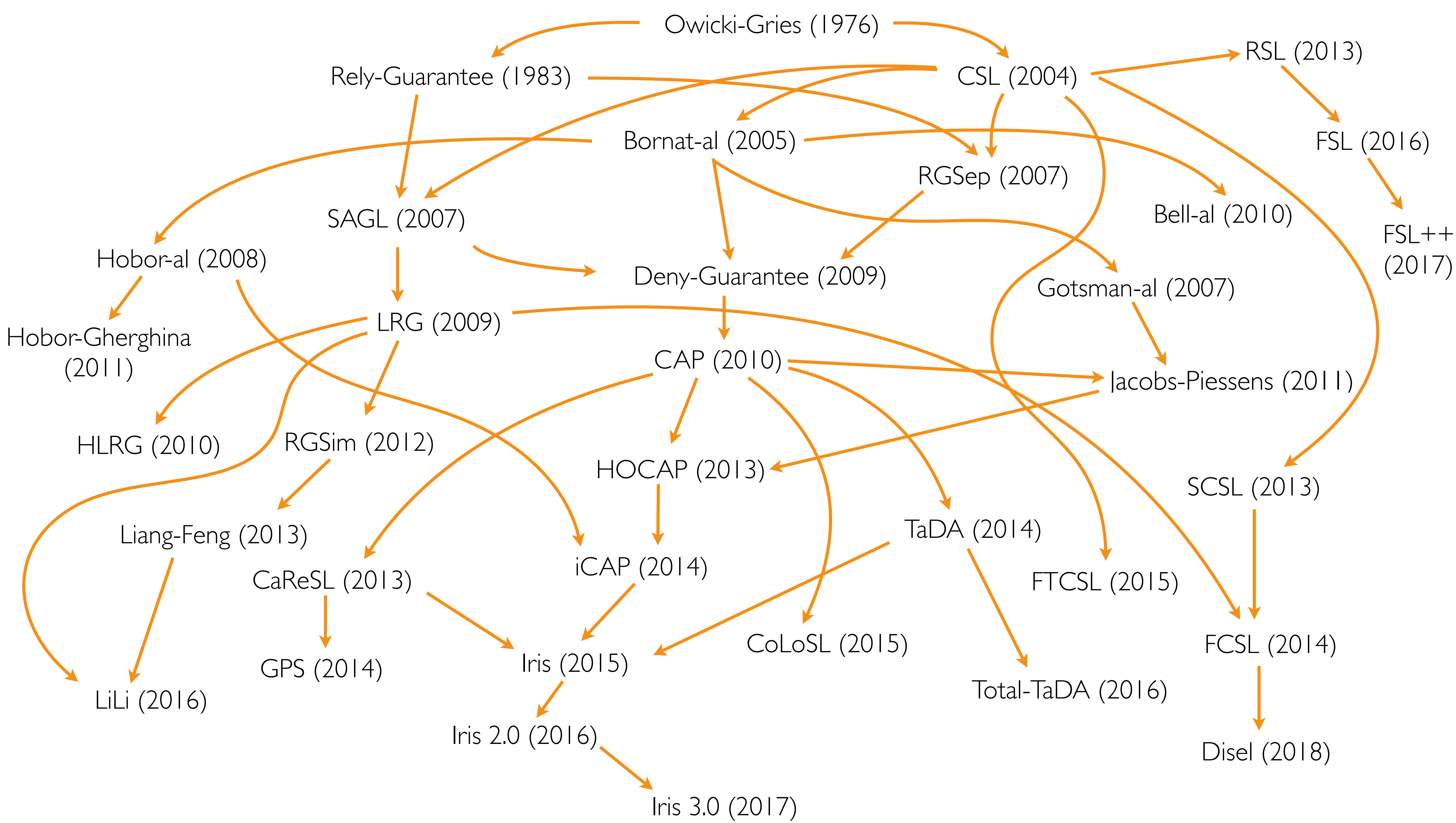
Less overhead, but...

...back to complex protocols of shared memory changes

```
#define atomic_store(object, desired) __c11_atomic_store(object, desired, __ATOMIC_SEQ_CST)
#define atomic_store_explicit __c11_atomic_store

#define atomic_load(object) __c11_atomic_load(object, __ATOMIC_SEQ_CST)
#define atomic_load_explicit __c11_atomic_load

#define atomic_exchange(object, desired) __c11_atomic_exchange(object, desired, __ATOMIC_SEQ_CST)
#define atomic_exchange_explicit __c11_atomic_exchange

#define atomic_compare_exchange_strong(object, expected, desired)
__c11_atomic_compare_exchange_strong(object, expected, desired, __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)
#define atomic_compare_exchange_strong_explicit __c11_atomic_compare_exchange_strong

#define atomic_compare_exchange_weak(object, expected, desired)
__c11_atomic_compare_exchange_weak(object, expected, desired, __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)
#define atomic_compare_exchange_weak_explicit __c11_atomic_compare_exchange_weak

#define atomic_fetch_add(object, operand) __c11_atomic_fetch_add(object, operand, __ATOMIC_SEQ_CST)
#define atomic_fetch_add_explicit __c11_atomic_fetch_add

#define atomic_fetch_sub(object, operand) __c11_atomic_fetch_sub(object, operand, __ATOMIC_SEQ_CST)
#define atomic_fetch_sub_explicit __c11_atomic_fetch_sub

#define atomic_fetch_or(object, operand) __c11_atomic_fetch_or(object, operand, __ATOMIC_SEQ_CST)
#define atomic_fetch_or_explicit __c11_atomic_fetch_or

#define atomic_fetch_xor(object, operand) __c11_atomic_fetch_xor(object, operand, __ATOMIC_SEQ_CST)
#define atomic_fetch_xor_explicit __c11_atomic_fetch_xor

#define atomic_fetch_and(object, operand) __c11_atomic_fetch_and(object, operand, __ATOMIC_SEQ_CST)
#define atomic_fetch_and_explicit __c11_atomic_fetch_and
```

(from Clang <stdatomic.h>, https://clang.llvm.org/doxygen/stdatomic_8h_source.html)

How do we prove safety of atomic-action concurrent programs?



(by Ilya Sergey, http://ilyasergey.net/other/CSL-Family-Tree.pdf)

Starling is a generic program logic, and tool for automating proofs in it

Concurrent proof

Program logic reducer

Hoare triples

General solver

Result

Starling

Sample Starling proof

```
shared int ticket;  // The next ticket to hand out.
shared int serving; // The current ticket holding the lock.

view holdTick(int t);  view holdLock();

method lock() {
  {| emp |}
    thread int t;  // The thread's current ticket.
    thread int s;  // The thread's current view of serving.
    <| t = ticket++; |>
  {| holdTick(t) |}
    do {
      {| holdTick(t) |}
        <| s = serving; |>
      {| if (s == t) { holdLock() } else { holdTick(t) } |}
    } while (s != t);
  {| holdLock() |}
}

method unlock() {
  {| holdLock() |} <| serving++; |> {| emp |}
}

constraint emp                                -> ticket >= serving;
constraint holdTick(t)                        -> ticket > t;
constraint holdLock()                         -> ticket != serving;
constraint holdLock()    * holdTick(t)        -> serving != t;
constraint holdTick(ta) * holdTick(tb)        -> ta != tb;
constraint holdLock()    * holdLock()         -> false;
```

Algorithm	No. lines Starling input	No. lines auxiliary input	No. lines proof input	No. lines gen GH	No. generated terms	No. SMT-elim terms	Time, total excl GH (s)	Time, on tool (s)	Time, SMT (s)	Time, GH (s)	Mem use, Starling (MiB)	Mem use, GH (MiB)
SMT/Z3:												
ARC (static)	52	-	19	-	40	40	1.62	1.55	0.08	-	118	-
Ticket lock (static)	47	-	16	-	18	18	1.49	1.44	0.05	-	94	-
Spinlock (static)	35	-	10	-	12	12	1.51	1.47	0.04	-	87	-
Reader/writer lock	109	-	45	-	160	160	1.85	1.67	0.19	-	192	-
Peterson's algo.	94	-	27	-	72	72	2.35	2.05	0.30	-	136	-
GRASShopper:												
ARC (alloc)	59	13	32	482	20	5	1.55	1.54	0.02	1.56	92	10.2
Ticket lock (alloc)	59	80	104	1054	66	30	1.48	1.46	0.02	3.64	87	10.8
Spin lock (alloc)	54	18	38	689	56	31	1.57	1.56	0.02	2.45	88	10.6
CLH queue-lock	124	10	58	1407	50	21	1.47	1.45	0.02	3.87	84	11.3
Lock-coupling list	79	118	154	5019	240	116	1.96	1.94	0.02	35.31	96	30.2

https://github.com/septract/starling-tool

Official Starling$_{\text{tool}}$ repository

https://github.com/MattWindsor91/starling-tool

My development fork of Starling$_{\text{tool}}$

https://gitlab.com/MattWindsor91/starling-coq

Mechanisation in Coq (WIP)

End.