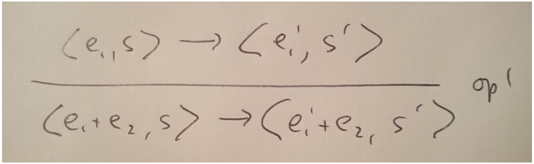
Semantics in Practice

Semantics of Practice

How do we write semantics?

1: pen-and-paper



How do we write semantics?

2: LaTeX

$$\text{(op1)} \quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \;\; op \;\; e_2, s\rangle \longrightarrow \langle e_1' \;\; op \;\; e_2, s'\rangle}$$

How do we write semantics?

2: LaTeX

$$\text{(op1)}\quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \;\; op \;\; e_2, s\rangle \longrightarrow \langle e_1' \;\; op \;\; e_2, s'\rangle}$$

```
\rL{op1} \mc
\autoinfer{}
  { \langle \tsvar{e}_{1},\tsvar{s}\rangle
      \longrightarrow
    \langle \tsvar{e}'_{1},\tsvar{s}'\rangle }
  { \langle \tsvar{e}_{1} \;\;\tsvar{op}\;\;\tsvar{e}_{2},\tsvar{s}\rangle
      \longrightarrow
    \langle \tsvar{e}'_{1} \;\;\tsvar{op}\;\;\tsvar{emyrb_{2myrb,\tsvar{smyrb'\rangle }
```

How do we write semantics?

2: LaTeX

$$\text{(op1)}\quad \frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \;\; op \;\; e_2, s\rangle \longrightarrow \langle e_1' \;\; op \;\; e_2, s'\rangle}$$

```
\rL{op1} \mc
\autoinfer{}
  { \langle \tsvar{e}_{1},\tsvar{s}\rangle
      \longrightarrow
    \langle \tsvar{e}'_{1},\tsvar{s}'\rangle }
  { \langle \tsvar{e}_{1} \;\;\tsvar{op}\;\;\tsvar{e}_{2},\tsvar{s}\rangle
      \longrightarrow
    \langle \tsvar{e}'_{1} \;\;\tsvar{op}\;\;\tsvar{emyrb_{2myrb,\tsvar{smyrb'\rangle }
```

Doable in-the-small, but doesn't scale: too hard to keep consistent

How do we *want to* write semantics?

```
  <e1,s> -> <e1',s'>
------------------------------- :: op1
<e1 op e2,s> -> <e1' op e2,s'>
```

- human-readable
- easy to type and edit
- version-control friendly

Ott

[Owens, Sewell, Zappa Nardelli; 2006–]

You write:

- the concrete grammar for your abstract syntax
- inductive rules over that grammar

Ott:

- parses that (enforcing variable conventions and judgement forms)
- generates typeset version
- supports Ott syntax embedded in LaTeX
- generates OCaml code for abstract syntax type
- generates theorem-prover definitions

Github: `https://github.com/ott-lang/ott` (research software...)

Example: L1 in Ott

```
grammar
e :: 'E_'  ::=               {{ com expressions }}
  | n                         ::   :: num
  | b                         ::   :: bool
  | e1 op e2                  ::   :: op
  | if e1 then e2 else e3     ::   :: if
  | l := e                    ::   :: assign
  | ! l                       ::   :: ref
  | skip                      ::   :: skip
  | e1 ; e2                   ::   :: sequence
  | while e1 do e2            ::   :: while
  | ( e )                     :: M :: paren  {{ ichlo ([[e]]) }}


defn
< e , s > -> < e' , s' > :: :: reduce :: ''
  {{ com $\langle$[[e]],\,[[s]]$\rangle$ reduces to $\langle$[[e']],\,[[s']]$\rangle$ }} by

  n1 + n2 = n
----------------------   :: op_plus
<n1 + n2, s> -> <n, s>

  <e1,s> -> <e1',s'>
-------------------------------- :: op1
<e1 op e2,s> -> <e1' op e2,s'>

  <e2,s> -> <e2',s'>
-------------------------------- :: op2
<e1 op e2,s> -> <e1 op e2',s'>

...
```

Example: L1 in Ott

e	$::=$		expressions
	$\mid$	n	
	$\mid$	b	
	$\mid$	$e_1\ op\ e_2$	
	$\mid$	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	
	$\mid$	$l := e$	
	$\mid$	$!l$	
	$\mid$	$\textbf{skip}$	
	$\mid$	$e_1 ; e_2$	
	$\mid$	$\textbf{while}\ e_1\ \textbf{do}\ e_2$	
	$\mid$	(e) M	

$\boxed{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}$ $\langle e, s\rangle$ reduces to $\langle e', s'\rangle$

$$\frac{n_1 + n_2 = n}{\langle n_1\ +\ n_2, s\rangle \longrightarrow \langle n, s\rangle} \quad \text{op\_plus}$$

$$\frac{n_1 \geq n_2 = b}{\langle n_1\ \geq\ n_2, s\rangle \longrightarrow \langle b, s\rangle} \quad \text{op\_gteq}$$

$$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ op\ e_2, s\rangle \longrightarrow \langle e_1'\ op\ e_2, s'\rangle} \quad \text{op1}$$

Example: OCaml$_{\text{light}}$ [Owens]

Scales from calculi to full-scale languages

OCaml$_{light}$ in Ott

Scott Owens

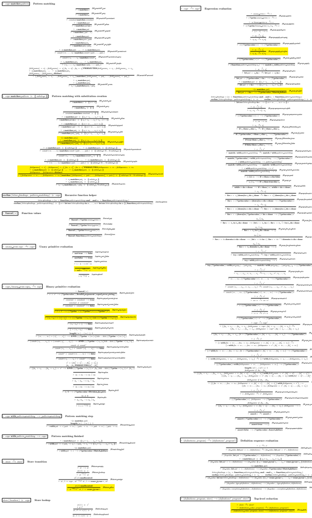
OCaml$_{light}$ (ESOP '08) is a formal semantics for a substantial subset of the Objective Caml core language, suitable for writing and verifying real programs.

OCaml$_{light}$ key points

- Written in Ott
- Faithful to Objective Caml (very nearly)
- Type soundness proof mechanized in HOL (Coq and Isabelle/HOL definitions generated too)
- Operational semantics validated on test programs
- Small-step operational semantics (131 rules)
- Type system (179 rules, below)

- definitions:
  - variant data types (e.g., `type t = I of int | C of char`),
  - record types (e.g., `type t = {f : int; g : bool}`),
  - parametric type constructors (e.g., `type 'a t = C of 'a`),
  - type abbreviations (e.g., `type 'a t = 'a * int`),
  - mutually recursive combinations of the above (excepting abbreviations),
  - exceptions, and values;
- expressions for type annotations, sequencing, and primitive values (functions, lists, tuples, and records);
- `with` (record update), `if`, `while`, `for`, `assert`, `try`, and `raise` expressions;
- let-based polymorphism with an SML-style value restriction;
- mutually-recursive function definitions via `let rec`;
- pattern matching, with nested patterns, as patterns, and "or" (|)patterns;
- mutable references with `ref`, `!`, and `:=`;
- polymorphic equality (the Objective Caml = operator);
- 31-bit word semantics for `int`s (using an existing HOL library); and
- IEEE-754 semantics for `float`s (using an existing HOL library).

The OCaml$_{light}$ Operational Semantics (131 rules)



How do we prove things about semantics?

1. Handwritten proof

How do we prove things about semantics?

1. Handwritten proof

2. LaTeX proof

e.g. `http://www.cl.cam.ac.uk/~pes20/hashtypes-tr-cam.pdf`

How do we prove things about semantics?

1. Handwritten proof

2. LaTeX proof

e.g. `http://www.cl.cam.ac.uk/~pes20/hashtypes-tr-cam.pdf`

Problems:

- error-prone
- very hard to maintain in face of changes to definitions

Solution: *mechanised proof assistants*

(aka *theorem provers*)
Software tools that:

- typecheck mathematical definitions
- do machine-checked primitive proof steps
- higher-level automation (decision procedures, tactics,...)

main tools:

- HOL4 (Mike Gordon et al.)
- Isabelle (Larry Paulson, Tobias Nipkow, et al.)
- Coq (INRIA)
- ACL2 (UT Austin)

HOL4 and Isabelle based on classical higher-order logic, using LCF idea of Robin Milner to ensure soundness relies on small core; Coq based on dependent type theory; ACL2 on pure LISP)

Example: L1 in Isabelle (Victor Gomes)

Github: `https://github.com/victorgomes/semantics`

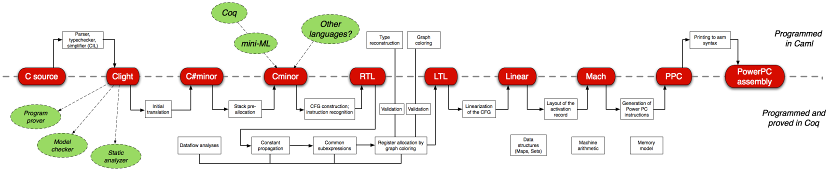
`https://github.com/victorgomes/semantics/blob/master/L1.thy`

Provers enable substantial verified software

- $\text{OCaml}_{\text{light}}$: mechanised HOL4 proof of type soundness

Provers enable substantial verified software

- CompCert: compiler for particular version of C
  `http://compcert.inria.fr/`



Theorem If program has no undefined behaviour w.r.t. the CompCert C semantics, and the compiler terminates successfully, then any behaviour of the compiled program w.r.t. the CompCert assembly semantics is a behaviour of the source program in the CompCert C semantics. [Proof in Coq]

Provers enable substantial verified software

- CompCert: compiler for particular version of C
  `http://compcert.inria.fr/`
- CakeML: verified compiler for ML-like language
  `https://cakeml.org/`
- seL4: verified hypervisor
  `https://sel4.systems/`
- Vellvm: verified LLVM optimisations
  `http://www.cis.upenn.edu/~stevez/vellvm/`
- IronClad, CertiKOS, VST, Everest, CompCertTSO, ...

Amazing!

Amazing!

but... divorced from normal software development process

Amazing!

but... divorced from normal software development process

In normal practice:

- the only way to assess whether s/w is good is to run it on tests
- we have to manually specify allowed outcomes for each test
- we typically have specification documents
  - usually precise about syntax
  - usually ambigous prose description of behaviour
- the *de facto standards* are unclear

Amazing!

but... divorced from normal software development process

Semantics gives us a way of being precise about behaviour

- can use for proof (hand or mechanised), as we've seen
- but so far can't use in *testing*; disconnected from normal development
- and we don't have semantics for key abstractions

http://rems.io

Cambridge Systems (OS/Arch/Security) + Semantics, Imperial, Edinburgh

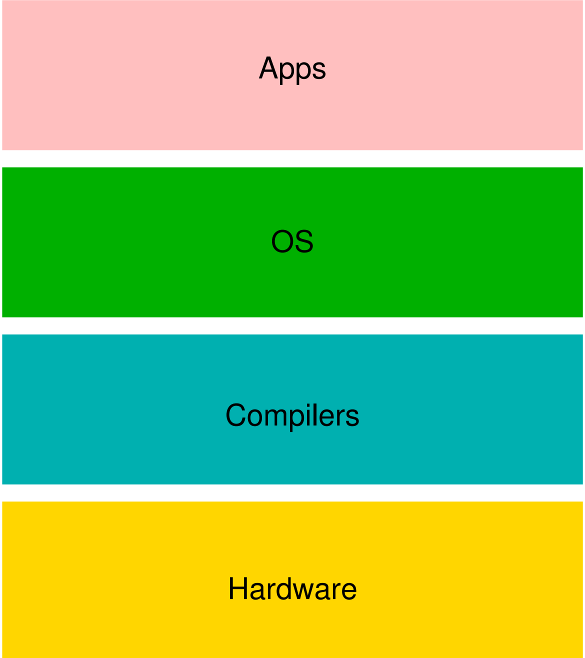
Investigators – Systems: Crowcroft, Madhavapeddy, Moore, Watson

Investigators – Semantics: Gardner, Gordon, Pitts, Sewell, Stark,

Researchers: Campbell, Chisnall, Flur, Fox, French, Gomes, Gray, Joannou, Kell, Matthiesen, Mehnert, Memarian, Mersinjak, Mulligan, Naylor, Nienhuis, Norton-Wright, Ntzik, Pichon-Pharabod, Pulte, Raad, da Rocha Pinto, Roe, Sezgin, Svendsen, Wassell, Watt

Alumni: Batty, Dinsdale-Young, Kammar, Kerneis, Kumar, Lingard, Myreen, Sheets, Tuerk, Villard, Wright

Collaborations: Deacon, Maranget, Reid, Ridge, Sarkar, Williams, Zappa Nardelli, ...


Apps
OS
Compilers
Hardware


Sunlight
Little Caesars
HOT-N-READY
PIZZA
McCain
PIZZA
POCKETS · POCHETTES
Minute Rice
Beatrice
OASIS
KraftDinner
12 PACK
HOT-N-READY
BBQ


BANKING S/W
BROWSER
JS
C
TLS
FILESYSTEM
TCP
ARCH
H/W
Sunlight
Little Caesars
HOT-N-READY
PIZZA
Minute Rice
Beatrice
OASIS
KraftDinner
12 PACK
HOT-N-READY
BBQ

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]

- full verification

  [mechanised proofs of functional correctness (all or nothing)]

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]

- full verification

  [mechanised proofs of functional correctness (all or nothing)]

- reason on idealised models

  [useful for design, but disconnected from real systems]

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]
- full verification

  [mechanised proofs of functional correctness (all or nothing)]
- use 1980s languages instead of 1970s (or 1990s) languages

  [useful, but only hits some problems]
- reason on idealised models

  [useful for design, but disconnected from real systems]

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]
- full verification

  [mechanised proofs of functional correctness (all or nothing)]
- bug-finding analysis tools

  [applicable to real systems, but incomplete and unsound]
- use 1980s languages instead of 1970s (or 1990s) languages

  [useful, but only hits some problems]
- reason on idealised models

  [useful for design, but disconnected from real systems]

Semantics to the rescue?

Options:

- rebuild clean-slate stack

  [good research, but deployable? And... do we know how?]
- full verification

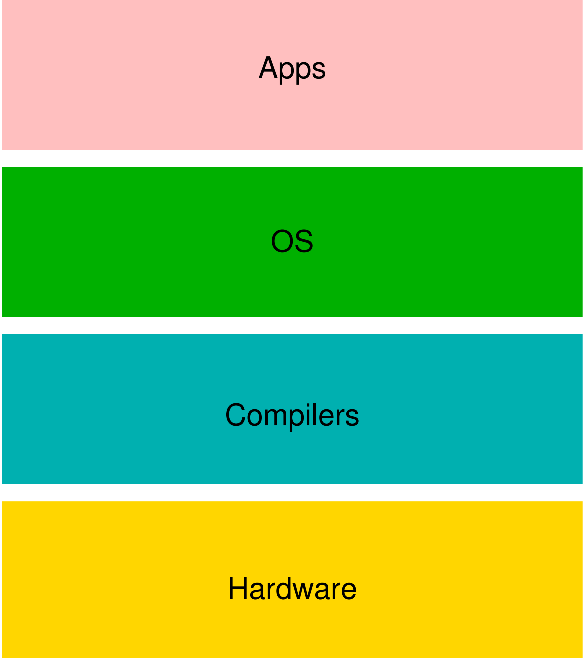
  [mechanised proofs of functional correctness (all or nothing)]
- full *specification* of key interfaces

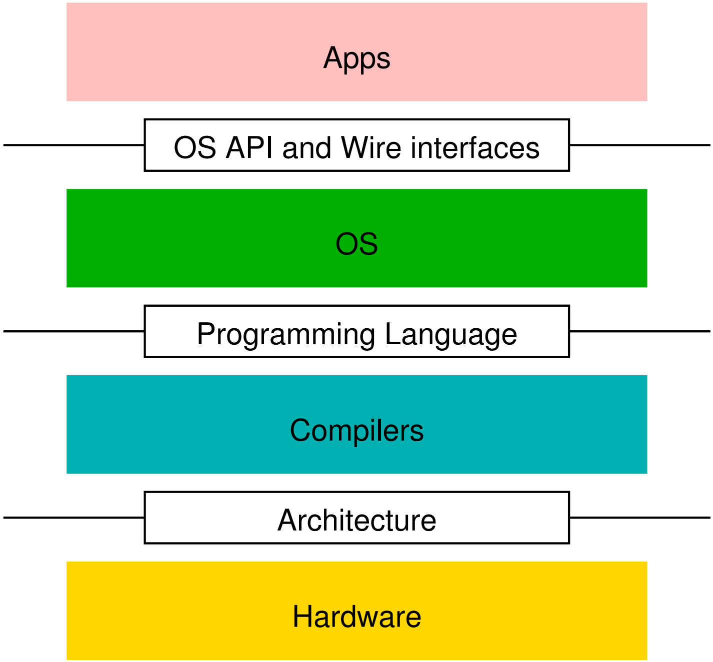
  [for formally based testing and design, + verification where possible]
- bug-finding analysis tools

  [applicable to real systems, but incomplete and unsound]
- use 1980s languages instead of 1970s (or 1990s) languages

  [useful, but only hits some problems]
- reason on idealised models

  [useful for design, but disconnected from real systems]


Apps
OS
Compilers
Hardware


Apps
OS API and Wire interfaces
OS
Programming Language
Compilers
Architecture
Hardware

REMS

http://rems.io

Apps

OS API and Wire interfaces

- TLS: nqsbTLS
- TCP/IP: Huginn-TCP
- POSIX filesystem test oracle: SibylFS
- POSIX filesystem logic

OS

Programming Language

- Sequential C (ISO/de facto): Cerberus
- Concurrent C: C/C++11, OpenCL, new
- C runtime type checking: libcrunch
- ELF linking: linksem
- Verified ML implementation: CakeML

Compilers

Architecture

- Multiprocessor Concurrency (ARM, POWER, x86, GPU)
- Multiprocessor ISA, in Sail and L3 (ARM, POWER, CHERI, MIPS, RISC-V, x86)
- CHERI

Hardware

Semantic Tools

Concurrency Reasoning

REMS

http://rems.io

Apps

OS API and Wire interfaces

- TLS: nqsbTLS
- TCP/IP: Huginn TCP...
- POSIX filesystem test o... ...ibyll FS
- POSIX filesystem logic

IETF

FreeBSD

40 filesystem configs

POSIX

OS

Programming Language

- Sequential C (ISO/d...
- Concurrent C: C/C++11, OpenCL, new
- C runtime type checking: libcrunch
- ELF linking: linksem
- Verified ML implementation: CakeML

C devs / ISO WG14

ISO WG21/WG14

Compilers

ARM, IBM, Qualcomm, Apple, NVIDIA, Linux

Architecture

- Multiprocessor Concurrency
  (ARM, POWER, x86, GPU)
- Multiprocessor ISA, in Sail and L3
  (... POWER, CHERI, MIPS, RISC-V, x86)
- CHERI

ARM, IBM

CTSRD/CHERI team

Hardware

Semantic Tools

Concurrency Reasoning

Key Idea: Semantics *Executable as Test Oracle*

replace

prose descriptions of behaviour (typical in specification docs)

by

semantic specifications that are executable as a test oracle

i.e., programs or executable mathematics that *compute* whether any potential behaviour of the system is allowed or not

(need not be decidable in general, so long as it is often enough)

Key Idea: Semantics *Executable as Test Oracle*

replace

> *prose descriptions of behaviour (typical in specification docs)*

by

> *semantic specifications that are executable as a test oracle*

i.e., programs or executable mathematics that *compute* whether any potential behaviour of the system is allowed or not

(need not be decidable in general, so long as it is often enough)

This:

- greatly simplifies testing – don't need to curate allowed outcomes, so can do random or systematic test generation
- gives a way to investigate de facto standards: *experimental semantics*

How to express semantics executable as a test oracle?

many options:

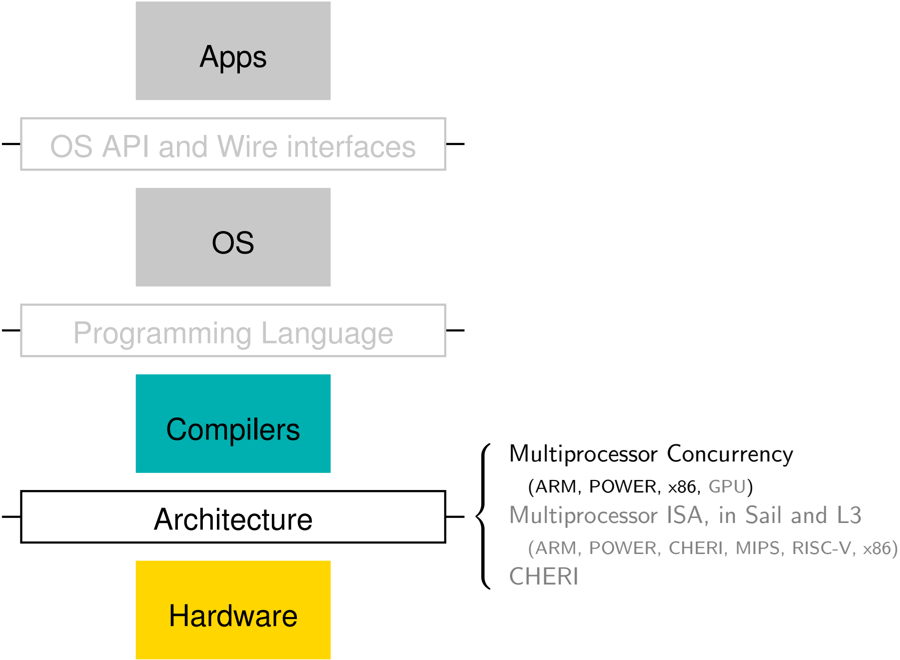
- pure function that checks input/output relation of system
  $spec : (input \times output) \rightarrow bool$
- pure function that checks trace of system
  $spec : (event\ list) \rightarrow bool$
  (plus instrumentation to capture traces)
- function that computes possible transitions of system
  $spec : state \rightarrow ((event \times state)\ set)$
  (e.g. if you can compute the exhaustive tree, and compare that with observed traces from instrumentation)
- relation that defines possible transitions of system
  $spec \subseteq state \times event \times state$
  together with some way to make that executable as the above

How to express semantics executable as a test oracle?

many options:

- pure function that checks input/output relation of system
  $spec : (input \times output) \rightarrow bool$
- pure function that checks trace of system
  $spec : (event\ list) \rightarrow bool$
  (plus instrumentation to capture traces)
- function that computes possible transitions of system
  $spec : state \rightarrow ((event \times state)\ set)$
  (e.g. if you can compute the exhaustive tree, and compare that with observed traces from instrumentation)
- relation that defines possible transitions of system
  $spec \subseteq state \times event \times state$
  together with some way to make that executable as the above

written in any of many languages: pure functional program, theorem prover, even C... Balancing clarity, execution, reasoning


Apps
OS API and Wire interfaces
OS
Programming Language
Compilers
Architecture
Hardware
Multiprocessor Concurrency
(ARM, POWER, x86, GPU)
Multiprocessor ISA, in Sail and L3
(ARM, POWER, CHERI, MIPS, RISC-V, x86)
CHERI

Real-world Concurrency

A naive two-thread mutual-exclusion algorithm:

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

Real-world Concurrency

A naive two-thread mutual-exclusion algorithm:

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

In $L1_{|}$, consider:

$$(x := 1; r_0 := y) | (y := 1; r_1 := x)$$

in initial state: $x = 0$ and $y = 0$

Is a final state with $r_0 = 0$ and $r_1 = 0$ possible?

Real-world Concurrency

A naive two-thread mutual-exclusion algorithm:

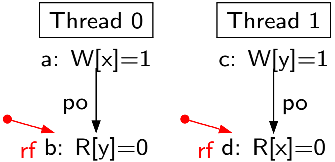
Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

In $L1_{|}$, consider:

$$(x := 1; r_0 := y) | (y := 1; r_1 := x)$$

in initial state: $x = 0$ and $y = 0$

Is a final state with $r_0 = 0$ and $r_1 = 0$ possible?

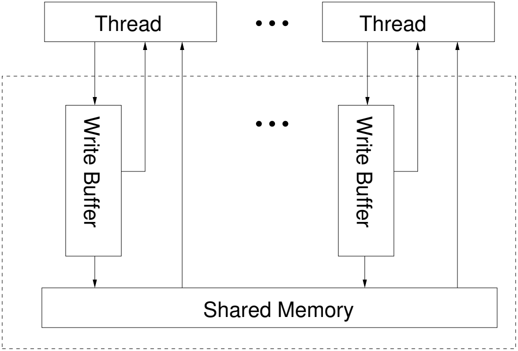


Test SB

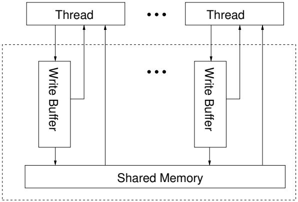
Let's try...

```
~/rsem/tutorial/lectures-acs/runSB.sh
```

x86-TSO Semantics



x86-TSO Semantics



An *x86-TSO abstract machine state m* is a record

$$m : \langle \quad M : addr \to value; \\ B : tid \to (addr \times value)\ list; \\ L : tid\ option \rangle$$

where

- $m.M$ is the shared memory, mapping addresses to values
- $m.B$ gives the store buffer for each thread, most recent at the head
- $m.L$ is the global machine lock indicating when a thread has exclusive access to memory

x86-TSO Abstract Machine: Behaviour

RM: Read from memory

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t)\\ m.M(x) = v\\ \mathrm{no\_pending}(m.B(t), x)\end{array}}{m \xrightarrow{t:\mathsf{R}\,x=v} m}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer.

x86-TSO Abstract Machine: Behaviour

RB: Read from write buffer

$$\dfrac{\begin{array}{l}\text{not\_blocked}(m, t)\\ \exists b_1\ b_2.\ m.B(t) = b_1 \mathbin{++} [(x, v)] \mathbin{++} b_2\\ \text{no\_pending}(b_1, x)\end{array}}{m \xrightarrow{t:\mathrm{R}\,x=v} m}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the newest write to x in its buffer;

x86-TSO Abstract Machine: Behaviour

WB: Write to write buffer

$$\frac{}{m \xrightarrow{t:\mathrm{W}\,x=v} m \oplus \langle\!\![B := m.B \oplus (t \mapsto ([(x,v)] +\!\!+ m.B(t)))]\!\!\rangle}$$

Thread t can write v to its store buffer for address x at any time;

x86-TSO Abstract Machine: Behaviour

WM: Write from write buffer to memory

$$\frac{\begin{array}{c}\mathrm{not\_blocked}(m, t)\\ m.B(t) = b +\!\!+ [(x, v)]\end{array}}{m \xrightarrow{t:\tau_{\ x=v}} m \oplus \langle\!\![M := m.M \oplus (x \mapsto v)]\!\!\rangle \oplus \langle\!\![B := m.B \oplus (t \mapsto b)]\!\!\rangle}$$

If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread

Validation of x86-TSO Semantics

- experiments on various x86 processor implementations
- discussion with vendor architects
- discussion with systems-programmer clients
- mechanised proof of properties

Epilogue

Lecture Feedback

Please do fill in the lecture feedback form – we need to know how the course could be improved / what should stay the same.

What can *you* use semantics for?

1. to understand a particular language — what you can depend on as a programmer; what you must provide as a compiler writer
2. as a tool for language design:
   2.1 for clean design
   2.2 for expressing design choices, understanding language features and how they interact.
   2.3 for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs
4. as tools for making precise specifications, executable as test oracles

The End