Semantics in Practice
Semantics of Practice
How do we write semantics?

1: pen-and-paper
How do we write semantics?

2: LaTeX

\[(\text{op1}) \quad \langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle \]

\[\langle e_1 \ \text{op} \ e_2, s \rangle \longrightarrow \langle e'_1 \ \text{op} \ e_2, s' \rangle \]
How do we write semantics?

2: LaTeX

\[
\begin{align*}
\langle e_1, s \rangle & \rightarrow \langle e'_1, s' \rangle \\
\langle e_1 \text{ op } e_2, s \rangle & \rightarrow \langle e'_1 \text{ op } e_2, s' \rangle
\end{align*}
\]

\textbackslash rL\{op1\} \textbackslash mc
\textbackslash autoinfer\{
\{ \textbackslash langle \text{tsvar\{e\}_\{1\}},\text{tsvar\{s\}\rangle \\
\text{longrightarrow} \\
\textbackslash langle \text{tsvar\{e\}_\{1\}},\text{tsvar\{s\}'\rangle \}
\{ \textbackslash langle \text{tsvar\{e\}_\{1\} } \text{\textbackslash ; } \text{tsvar\{op\} } \text{\textbackslash ; } \text{tsvar\{e\}_\{2\}},\text{tsvar\{s\}\rangle \\
\text{longrightarrow} \\
\textbackslash langle \text{tsvar\{e\}_\{1\} } \text{\textbackslash ; } \text{tsvar\{op\} } \text{\textbackslash ; } \text{tsvar\{emyrb\}_\{2myrb\},\text{tsvar\{smyrb\}'\rangle \}
\}
\}
How do we write semantics?

2: LaTeX

\[
\begin{align*}
\langle e_1, s \rangle & \rightarrow \langle e'_1, s' \rangle \\
\langle e_1 \; op \; e_2, s \rangle & \rightarrow \langle e'_1 \; op \; e_2, s' \rangle
\end{align*}
\]

\text{Doable in-the-small, but doesn’t scale: too hard to keep consistent}
How do we want to write semantics?

\[
<e_1, s> \rightarrow <e_1', s'>
\]

------------------------------ :: op1

\[
<e_1 \text{ op } e_2, s> \rightarrow <e_1' \text{ op } e_2, s'>
\]

- 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$\rangle$, \$\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

\[ e ::= \begin{align*} & n \\ & b \\ & e_1 \text{ op } e_2 \\ & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\ & l := e \\ & !l \\ & \text{skip} \\ & e_1 ; e_2 \\ & \text{while } e_1 \text{ do } e_2 \\ & (e) \end{align*} \]

\[ \langle e, s \rangle \rightarrow \langle e', s' \rangle \quad \langle e, s \rangle \text{ reduces to } \langle e', s' \rangle \]

\[ \frac{n_1 + n_2 = n}{\langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle} \quad \text{op\_plus} \]

\[ \frac{n_1 \geq n_2 = b}{\langle n_1 \geq n_2, s \rangle \rightarrow \langle b, s \rangle} \quad \text{op\_gteq} \]

\[ \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightarrow \langle e'_1 \text{ op } e_2, s' \rangle} \quad \text{op1} \]

\[ \langle n, s \rangle \rightarrow \langle e, s \rangle \]
Example: OCaml\textsubscript{light} [Owens]

Scales from calculi to full-scale languages
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

- OCaml\textsubscript{light}: mechanised HOL4 proof of type soundness
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 ambiguous 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
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, Mehnerd, 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, ...
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]
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

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

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

Semantic Tools
Concurrency Reasoning
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

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

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

C devs / ISO WG14
ISO WG21/WG14

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
  \[ spec : (input \times output) \to bool \]

- pure function that checks trace of system
  \[ spec : (event\ list) \to bool \]
  (plus instrumentation to capture traces)

- function that computes possible transitions of system
  \[ spec : state \to (((event \times state)\ set) \]
  (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
  \[ \text{spec} : (\text{input} \times \text{output}) \rightarrow \text{bool} \]

▶ pure function that checks trace of system
  \[ \text{spec} : (\text{event list}) \rightarrow \text{bool} \]
  (plus instrumentation to capture traces)

▶ function that computes possible transitions of system
  \[ \text{spec} : \text{state} \rightarrow ((\text{event} \times \text{state}) \text{ set}) \]
  (e.g. if you can compute the exhaustive tree, and compare that with observed traces from instrumentation)

▶ relation that defines possible transitions of system
  \[ \text{spec} \subseteq \text{state} \times \text{event} \times \text{state} \]
  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
### Multiprocessor Concurrency
- (ARM, POWER, x86, GPU)
- Multiprocessor ISA, in Sail and L3
  - (ARM, POWER, CHERI, MIPS, RISC-V, x86)
  - CHERI
A naive two-thread mutual-exclusion algorithm:

<table>
<thead>
<tr>
<th>Initial state: $x=0$ and $y=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
</tr>
<tr>
<td>$x=1$</td>
</tr>
<tr>
<td>if ($y==0$) { ...critical section... }</td>
</tr>
</tbody>
</table>
A naive two-thread mutual-exclusion algorithm:

<table>
<thead>
<tr>
<th>Initial state: $x=0$ and $y=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>$x=1$</td>
</tr>
<tr>
<td>if ($y==0$) { ...critical section... }</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Initial state: x=0 and y=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>x=1</td>
</tr>
<tr>
<td>if (y==0) { ...critical section... }</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W[x]=1</td>
<td>c: W[y]=1</td>
</tr>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>rf</td>
<td>rf</td>
</tr>
<tr>
<td>b: R[y]=0</td>
<td>d: R[x]=0</td>
</tr>
</tbody>
</table>
Let’s try...

~/rsem/tutorial/lectures-acs/runSB.sh
x86-TSO Semantics

Thread

Write Buffer

Shared Memory

Thread

Write Buffer
An x86-TSO abstract machine state $m$ is a record

$$m : \langle \begin{align*}
M & : \text{addr} \rightarrow \text{value}; \\
B & : \text{tid} \rightarrow (\text{addr} \times \text{value}) \text{ list}; \\
L & : \text{tid option}\end{align*} \rangle$$

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
$\textbf{RM: Read from memory}$

\[
\begin{align*}
\text{not\_blocked}(m, t) \\
m.M(x) &= v \\
\text{no\_pending}(m.B(t), x) \\
\hline \\
&m \quad t:\text{R}x=v \quad m
\end{align*}
\]

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.
RB: Read from write buffer

not_blocked\((m, t)\)

\[ \exists b_1 \ b_2. \ m.B(t) = b_1 \ pallottolo[\(x, \nu\)] \ pallottolo b_2 \]

no_pending\((b_1, x)\)

\[ \begin{array}{c}
  m \\
  t:R x=\nu \\
  \rightarrow \\
  m
\end{array} \]

Thread \(t\) can read \(\nu\) from its store buffer for address \(x\) if \(t\) is not blocked and has \(\nu\) as the newest write to \(x\) in its buffer;
WB: Write to write buffer

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

Thread \( t \) can write \( v \) to its store buffer for address \( x \) at any time;
WM: Write from write buffer to memory

\[
\text{not\_blocked}(m, t) \\
m.B(t) = b \uparrow\uparrow [(x, v)] \\
\]

\[
m \xrightarrow{t: \tau\ x=v} m \oplus \langle [M := m.M \oplus (x \mapsto v)] \rangle \oplus \langle [B := m.B \oplus (t \mapsto b)] \rangle
\]

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