Mechanising Complicated Operational Semantics

Michael Norrish

UNIVERSITY OF CAMBRIDGE Computer Laboratory
Motivating Mechanisation

- Ideally, formal methods analyses of software build on a formal semantics of the programming languages or systems involved.

- But formal semantics (when they exist!) for real languages/systems are hideously complicated.

- Can we fight this complexity through mechanisation of the semantics?
Mechanisation?

Mechanisation =
“creating a mechanically manipulable form of a (possibly originally pen & paper) formal description”

Will look at my experiences with *C* and *UDP*, using the HOL system.
Why HOL?

Vocabulary confusion alert: HOL, the system, is one of a number of theorem-proving systems that implement proof in higher order logics. Others are Coq, Isabelle/HOL and PVS.
Why Higher Order Logic?

Expressiveness: One system can attack many different targets, embed other logics, leverage off other work (e.g., arithmetic decision procedures), retaining seductive flexibility.

Simplicity: Type systems are not complicated. HOL and Isabelle/HOL have decidable typing. Inference reduces verbosity.
HOL Mechanisation of C

- Wrote a formal operational semantics, using ISO standard and comp.std.c
- Proved meta-theoretic results: determinism
  - will explain this in more detail later
- Did two tiny (very tiny!) verifications
HOL Mechanisation of UDP

With Peter Sewell, Keith Wansbrough & Andrei Serjantov:

- Mechanised complicated labelled transition system, describing sockets API and network message passing for UDP internet protocol
- Found bugs
- Revised description to include time and threads
Comparison of sources

Both *post hoc*.

**C:**
- ISO standard specified semantics
- Consultation with others helped clarify ambiguities

**UDP:**
- Used RFCs, OS documentation, Linux source code
- Clarified with experimental validation
Mechanising C—Outline

- Definition of types in the logic to represent entities of interest: C types, expressions, statements...
- Define typing relation (static semantics)
- Define reduction relation (dynamic semantics)
- Prove results...
C—Type Definitions

For example, C’s types:

\[ \tau ::= \text{int} \mid \text{char} \mid \ldots \mid \tau^* \mid \tau[n] \mid \tau^* \rightarrow \tau \mid \text{struct tag} \]

(Not all possibilities are valid types: must forbid arrays of zero size; functions returning arrays . . .)

Similar definitions for expressions and statements.
Rules for address-taking and pointer dereference:

\[
\Gamma \vdash e : \text{obj}[\tau] \quad \Gamma \vdash e : \tau^* \quad \tau \neq \text{void} \\
\Gamma \vdash \&e : \tau^* \\
\Gamma \vdash *e : \text{obj}[\tau]
\]

The type \text{obj}[\tau] is an l-value of type \tau.

Variables also have \text{obj}[\tau] type.
“Arrays are just pointers”:

\[
\Gamma \vdash e : \text{obj}[\tau] \quad \tau \text{ not an array}
\]

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau}\]

\[
\Gamma \vdash e : \text{obj}[\tau[n]]
\]

\[
\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash e : \tau^*}
\]
Many infamous C behaviours are instances of *undefined behaviour*. Examples include:

- running off the end of arrays
- accessing uninitialised memory
- casting values to incompatible types

*Implementations* may do Weird Stuff when these things happen; my semantics regards them all as aborts.
C—Program States

- map variable names to addresses, and addresses to bytes;
- track address status (allocation, initialisation);
- record pending side effects, update maps;
- store global and local versions of variable and address maps.
One source of non-determinism is order of evaluation.

\[
\begin{align*}
\langle e_1, \sigma \rangle & \rightarrow \langle e'_1, \sigma' \rangle \\
\langle e_1 + e_2, \sigma \rangle & \rightarrow \langle e'_1 + e_2, \sigma' \rangle \\
\langle e_2, \sigma \rangle & \rightarrow \langle e'_2, \sigma' \rangle \\
\langle e_1 + e_2, \sigma \rangle & \rightarrow \langle e_1 + e'_2, \sigma' \rangle
\end{align*}
\]

(Contrast: && only allows evaluation of left argument.)
Another source of non-determinism is side effects:

- Side effects need not be applied immediately
- Side effects need not be applied in order

With \( v \) initially 3,

\[ v++ + v++ + v++ + v++ \]

might result in values anywhere between 12 and 18. (Mightn’t it?)
Within a “phase” of expression evaluation,

- updating the same object twice is undefined behaviour

- updating and referring to the same object is undefined behaviour, unless the reference was made to calculate the new value
## C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v++ + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v++ + *i</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v = v + 1</code></td>
<td>OK</td>
</tr>
<tr>
<td><code>a[a[i]] = 0</code></td>
<td>?</td>
</tr>
</tbody>
</table>
## C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>v++ + v++</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
### C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>v++ + v++</td>
<td>Undefined</td>
</tr>
<tr>
<td>v + v++</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

UKC Talk, 13 May 2002 – p.18
## C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>v++ + v++</td>
<td>Undefined</td>
</tr>
<tr>
<td>v + v++</td>
<td>Undefined</td>
</tr>
<tr>
<td>v++ + *i</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

As long as `i` points to `v`
### C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v++ + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v++ + *i</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v = v + 1</code></td>
<td>OK</td>
</tr>
</tbody>
</table>

as “updating and referring to the same object is undefined behaviour, *unless the reference was made to calculate the new value*”
### C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v++ + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v + v++</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v++ + *i</code></td>
<td>Undefined</td>
</tr>
<tr>
<td><code>v = v + 1</code></td>
<td>OK</td>
</tr>
<tr>
<td><code>a[a[i]] = 0</code></td>
<td>?</td>
</tr>
</tbody>
</table>

When `a[i] = i`
Other aspects of C’s dynamic semantics are straight-forward:

- Statement semantics (excluding goto) has easy definition
- Variables to values via addresses tedious
- Minor questions about function calls
C—Determinism Result

Using HOL, I proved:

- If a (sequence-point free) C expression doesn’t exhibit undefined behaviour, then it will have just one result.

- If such an expression can exhibit undefined behaviour, then it will exhibit undefined behaviour.

Both proofs involve showing diamond properties for expression reduction (→).
C—verification?

The humble strcpy:

```c
void strcpy(char *s, char *t) {
    while (*s++ = *t++);
}
```
C—verification?

A statement of its correctness:

\[
\forall \sigma_0 \quad s\_addr \ t\_addr \ r \ \sigma.
\]

\[
(\Phi_{\sigma_0}("strcpy") = \langle \text{Code for strcpy} \rangle) \land I_{\sigma_0} \subseteq \Lambda_{\sigma_0} \land \text{safe}(\sigma_0) \land
\]

\[
(\exists v. \text{alloc}(\sigma_0, 3 \ast \text{sizeof(char *)} - 1, v)) \land \Pi_{\sigma_0} = \emptyset \land
\]

\[
\{s\_addr \ldots s\_addr + |t|\} \subseteq \Lambda_{\sigma_0} \land \{t\_addr \ldots t\_addr + |t|\} \subseteq I_{\sigma_0} \land
\]

\[
\{t\_addr \ldots t\_addr + |t|\} \cap \{s\_addr \ldots s\_addr + |t|\} = \emptyset \land
\]

\[
s\_addr \ldots s\_addr + |t| \text{ all representable addresses} \land
\]

\[
t\_addr \ldots t\_addr + |t| \text{ all representable addresses} \land
\]

\[
\langle"strcpy"((s\_addr,\text{char *}), (t\_addr,\text{char *})), \sigma_0 \rangle \rightarrow_e \langle r, \sigma \rangle \implies
\]

\[
(r = (\emptyset, \text{void})) \land (I_{\sigma} = I_{\sigma_0} \cup \{s\_addr \ldots s\_addr + |t|\}) \land
\]

\[
(\forall a \in I_{\sigma}. (M_{\sigma}(a) = M_{\sigma_0}[\langle s\_addr \ldots \rangle := t](a)))
\]

where there is a null byte somewhere in memory after \( t\_addr \), thereby defining a finite sequence of characters, referred to above as \( t \).
UDP—Introduction

Authors (Sewell et al.) of existing pen & paper semantics (TACS’2001) felt it was out-of-control:

- 60+ rules
- many auxiliary functions
- LaTeX math-mode doesn’t come with a type-checker

Theorem-proving to the rescue!
Mechanising UDP—Outline

- Definition of types in the logic to represent entities of interest: hosts, networks, packets...
- Define desired invariants over these entities (static semantics)
- Define behaviour of evolving system (dynamic semantics)
- Prove results...
UDP—Type Definitions

Messages:

\[
ipBody ::= \text{port}\uparrow \times \text{port}\uparrow \times \text{data} \quad (\text{UDP packet})
\]
\[
| \quad \text{ip} \times \text{port}\uparrow \times \text{ip} \times \text{port}\uparrow \quad (\text{ICMP host unreachable})
\]
\[
| \quad \text{ip} \times \text{port}\uparrow \times \text{ip} \times \text{port}\uparrow \quad (\text{ICMP port unreachable})
\]

\[
\text{msg} ::= \langle \text{src} : \text{ip}, \text{dest} : \text{ip}, \text{body} : \text{ipBody} \rangle
\]

Sockets:

\[
\text{socket} ::= \langle \text{fd} : \text{fd}, \text{is1} : \text{ip}\uparrow, \text{ps1} : \text{port}\uparrow, \\
\text{is2} : \text{ip}\uparrow, \text{ps2} : \text{port}\uparrow, \\
\text{es} : \text{error}\uparrow, \text{f} : \text{flags}, \text{mq} : (\text{msg} \times \text{ifid})* \rangle
\]
UDP—Invariants

Our invariants included:

- The file descriptor associated with a socket in a host should be associated only with that socket.

- No message in a socket’s incoming queue should include a “martian” address.

And many (more complicated) others...
Model encompasses a number of complicated features:

**Failure:** The network drops/duplicates packets

**Asynchronicity:** e.g., if a message is not available, return from `recvfrom` immediately

**Long range failure:** A bounced packet can set an error flag in the originating socket some time after its sent
UDP—Dynamics Issues II

More complexity:

**Threads:** Each host supports multiple threads at once

**Time:** Time passes, system calls can time out.
UDP—Dynamics I

Dynamics described with a multi-layered, labelled transition system. Host reductions of form:

\[ h \xrightarrow{\text{label}} h' \]

Host \( h \) may be interacting with:

**The network:** Sending or receiving a message

**An application:** Receiving a system call request, or returning a result to an earlier request
A rule describing a thread \textit{tid}'s successful attempt to connect a socket to an external address:

$$
\mathcal{F}(\text{ifds, tid, RUN}_d, \text{SOCK}(fd, *, ps_1, *, *, e, f, mq))
$$

\[ tid \cdot \text{connect}(fd, i, ps) \]

$$
\mathcal{F}(\text{ifds, tid, RET}(\text{OK}())_{dsch}, \text{SOCK}(fd, \uparrow i_1, \uparrow p'_1, \uparrow i, ps, e, f, mq))
$$

where \( F_{\text{context}}(\mathcal{F}) \land i_1 \in \text{outroute}(\text{ifds, i}) \land p'_1 \in \text{autobind}(ps_1, \mathcal{F}) \)
UDP—A Safety Property

\[ h_0 \xrightarrow{\ell} h \land host\_ok(h_0) \Rightarrow host\_ok(h) \]

Property \textit{host\_ok} includes:

if a thread is blocking on a \texttt{recvfrom} call on a file-descriptor \texttt{fd}, then there must be a socket with that descriptor, and it must have a non-null \texttt{ps1} field. (Watch out for another thread calling \texttt{close}.)
Keith Wansbrough

- wrote a simple heart-beat detector system (client/server)
- stated a correctness property in HOL (client gets correct status within 3 seconds)
- proved this by hand, making unrealistically strong assumptions
UDP & HOL

- Initial UDP work predated my involvement
- My first task: mechanise early version of UDP model
- Then, with others: mechanised more complicated version with threads and time
My experience

Positive!

- Type system expresses all I need (no call for binders / terms up to $\alpha$-equivalence)
- Use of (ad hoc) overloading preserves notational convenience
- Definitional tools (types, inductive relations, auxiliary functions) up to the task
- Found bugs (principally type errors and obvious typos)
And then... 

Others want to:

- Build on old semantics to add new, complicated features (threads, time)
- Retain benefits of mechanisation
- Develop/prototype themselves (avoiding me as bottleneck)
Their experience

Mixed:

+ Learn HOL syntax readily (similarity with functional programming)
+ Write “programs in the logic” happily (again FP familiarity)
- Say they want sub-typing (PVS has this)
- Complain at length about error-reporting
A final verdict

From early draft of our ETAPS’02 paper:

As for the use of HOL, our work has supported the view that partial mechanization can be feasible and valuable. At least from the point of view of the naïve users among us, better diagnostics and response time are important, to stop the tool ‘getting in the way’. Automated testing directly from the model would be of great assistance.

We’re keen to have a go at TCP/IP too...
Lessons for the HOL developer

- Users *can* get off to flying starts with HOL
- Particularly if they are principally interested in definition (rather than proof)
- Error-handling, error messages and diagnostics are extremely important, and need to be as good as possible
Why bother?

It may be considerably easier to embody semantics in a custom tool (analysis system), and not have to deal with the pain of interactive theorem-proving.
Why bother?

It may be considerably easier to embody semantics in a custom tool (analysis system), and not have to deal with the pain of interactive theorem-proving. However

- Getting a semantics nailed down is valuable
- Meta-theoretic results are possible
- It *is* possible to base real tools on these semantics
Moreover

Novice users *can* use interactive theorem-provers to make definitions,

and

develop the foundations of a mechanised formal semantics.