Australian National University

COMP3610/6361
Principles of Programming Languages

Peter Höfner

Oct 20, 2023

Section 25

Conclusion

Learning Outcome I

1. Understand the role of theoretical formalisms, such as operational and denotational semantics

   - IMP language
   - operational semantics
   - denotational semantics
   - axiomatic semantics
   - functions
     (call-by-name, call-by-value)
   - references
   - extensions
     (data structures, error handling, object-orientation,. . .)

Learning Outcome II

2. Apply these semantics in the context of programming languages

    - IMP language + extensions
    - configurations
    - derivations
    - transitions

Learning Outcome III

3. Evaluate differences (advantages/disadvantages) of these theoretical formalisms
    - small-step vs big-step
    - operational vs denotational vs axiomatic (vs algebraic)

Learning Outcome IV

4. Create operational or denotational semantics of simple imperative programs

    - IMP + extensions + types
    - derivations
    - transitions

Learning Outcome V

5. Analyse the role of types in programming languages

    - types
    - subtypes
    - progress and preservation properties
    - Curry-Howard correspondence

Learning Outcome VI

6. Formalise properties and reason about programs

    - Isabelle/HOL
    - semantic equivalences
    - decorated programs
    - Floyd-Hoare logic, wlp
    - Owicki-Gries, Rely-Guarantee

Learning Outcome VII

7. Apply basic principles for formalising concurrent programming languages

   - Guarded Command Language
   - process algebra
     (value-passing CCS and pure CCS)
   - semantic equivalences
   - Owicki-Gries, Rely-Guarantee

Learning Outcome VIII

8. Additional Outcomes

   - structural induction
   - substitution
   - ...

We covered A LOT

. . . but it’s only the tip of the iceberg

The Message I

Good language design?

- precise definition of what the language is
  (so can communicate among the designers)
- technical properties
  (determinacy, decidability of type checking, etc.)
- pragmatic properties
  (usability in-the-large, implementability)

(that’s also an answer to LO1)

The Message II

What can you use semantics for?

- to understand a particular language
  - what you can depend on as a programmer
  - what you must provide as a compiler writer
- as a tool for language design:
  - for clean design
  - for expressing design choices, understanding language features and how they interact
  - for proving properties of a language, eg type safety, decidability of type inference.
- as a foundation for proving properties of particular programs verified software

Trend: Verified Software

- increasingly important
- "rough consensus and running code" (trial and error) is not sufficient
- develop operational models of *real-world* languages/applications

- progress in verification makes it possible

  build end-to-end verified systems

  - formal semantics for (a large subset of C) [see M. Norrish]
  - CompCert/CakeML: verified compilers (full compiler verified in Coq/HOL4)
  - seL4: high-assurance, high-performance operating system microkernel (proofs in Isabelle/HOL)
  - formal semantics for hardware (PPC, x86, ARM)

Are We Done

- more ‘standard’ features
  - dependent types
  - continuations
  - lazy evaluation
  - side effects
- more support for separation of concerns
  - low-level features, such as memory models
  - high-level features, such as broadcast
- more applications
  - optimisations
  - code generation

More Features – Dependent Types

- having “compile-time” types that depend on “run-time” values
- can avoid out-of-bounds errors

More Features – Dependent Types

example: typing Lists with Lengths

non-dependant type for list (similar to trees)

$$
\begin{array}{ll}
\text{nil} & : \text{IList} \\
\text{cons} & : \text{int} \rightarrow \text{IList} \rightarrow \text{IList} \\
\text{hd} & : \text{IList} \rightarrow \text{int} \\
\text{tl} & : \text{IList} \rightarrow \text{IList} \\
\text{isnil} & : \text{IList} \rightarrow \text{bool}
\end{array}
$$

More Features – Dependent Types

Example: Typing Lists with Lengths

dependant type for list (carry around length)

nil : IList 0
cons : Πn:nat. int $\rightarrow$ (IList n) $\rightarrow$ (IList (succ n))
hd : Πn:nat. (IList (succ n)) $\rightarrow$ int
tl : Πn:nat. (IList (succ n)) $\rightarrow$ (IList n)
~~isnil~~ :

More Features – Dependent Types

Example: typing lists with lengths

- using and checking dependent types

$$
\begin{aligned}
&(\textbf{fn}\ n : \mathsf{nat} \Rightarrow (\textbf{fn}\ l : \mathsf{IList}(\mathsf{succ}\ (\mathsf{succ}\ n)) \Rightarrow \\
&\qquad (\mathsf{hd}\ (\mathsf{succ}\ n)\ l) + \\
&\qquad (\mathsf{hd}\ n\ (\mathsf{tl}\ (\mathsf{succ}\ \ n)\ l)) \\
&))
\end{aligned}
$$

- propositions as dependent types
  (Curry–Howard lens)

$$\mathsf{get} : \Pi m : \mathsf{nat}.\ \Pi n : \mathsf{nat}.\ (\mathsf{Less}\ m\ n) \to (\mathsf{IList}\ n) \to \mathsf{int}$$

More Feature – Hardware Model

Fundamental Question

What is the behaviour of memory?

- . . . at the programmer abstraction
- . . . when observed by concurrent code

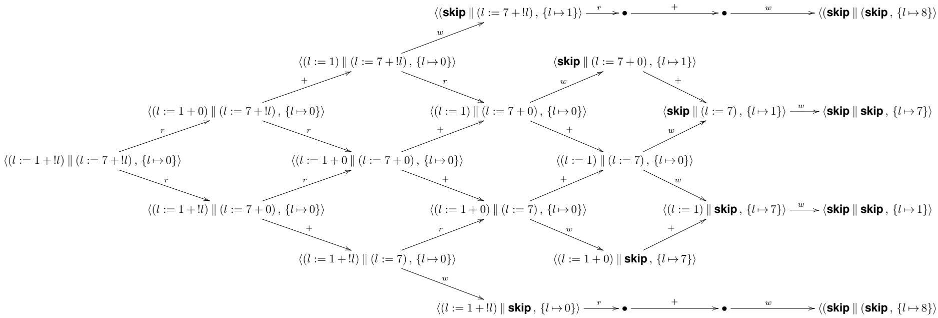
More Feature – Hardware Model

First Model: Sequential Consistency

Multiple threads acting on a sequentially consistent (SC) shared memory:

> *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program*
>
> *[Lamport, 1979]*

More Feature – Hardware Model



More Feature – Hardware Model

- implement naive mutual exclusion
- specify concepts such as “atomic” (see GCL)
- but on x86 hardware you have these behaviours
    - hardware busted?
    - program bad?
    - model is wrong?

SC is not a good model of x86 (or of Power, ARM, Sparc, Itanium. . .)

More Feature – Hardware Model
New problem?

No: IBM System 370/158MP in 1972, already non-SC

More Feature – Hardware Model

But still a research question

- mainstream architectures and languages are key interfaces
- . . . but it is been very unclear exactly how they behave

- **more fundamentally:**
  - it is been (and in significant ways still is) unclear how we can specify that precisely
  - *if* we can do that, we can build on top:
    explanation, testing, emulation, static/dynamic analysis, model-checking, proof-based verification,. . .

More Features – Broadcast

Motivation:
model communication

- network protocols
- communication protocols
- . . .

Broadcast in CCS

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\eta} P'}{P|Q \xrightarrow{\eta} P'|Q} \qquad \frac{P \xrightarrow{c} P',\ Q \xrightarrow{\bar{c}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{Q \xrightarrow{\eta} Q'}{P|Q \xrightarrow{\eta} P|Q'}$$

$$\frac{P \xrightarrow{\ell} P'}{P[f] \xrightarrow{f(\ell)} P'[f]} \qquad \frac{P \xrightarrow{\ell} P'}{P\backslash c \xrightarrow{\ell} P'\backslash c}\ (c{\neq}\ell{\neq}\bar{c}) \qquad \frac{P \xrightarrow{\ell} P'}{A \xrightarrow{\ell} P'}\ (A \stackrel{def}{=} P)$$

$$\frac{P \xrightarrow{b\sharp_1} P',\ Q \stackrel{b?}{\not\longrightarrow}}{P|Q \xrightarrow{b\sharp_1} P'|Q} \qquad \frac{P \xrightarrow{b\sharp_1} P',\ Q \xrightarrow{b\sharp_2} Q'}{P|Q \xrightarrow{b\sharp} P'|Q'} \qquad \frac{P \stackrel{b?}{\not\longrightarrow},\ Q \xrightarrow{b\sharp_2} Q'}{P|Q \xrightarrow{b\sharp_2} P|Q'}$$

$\sharp_1 \circ \sharp_2 = \sharp \neq \_$ with

∘	!	?
!	_	!
?	!	?

Broadcast in CCS

- parallel composition associative, commutative?
- all operators are a congruence?

Case Study: AODV

Ad Hoc On-Demand Distance Vector Protocol

- routing protocol for wireless mesh networks (wireless networks without wired backbone)

- ad hoc (network is not static)
- on-Demand (routes are established when needed)
- distance (metric is hop count)

- developed 1997–2001 by Perkins, Beldig-Royer and Das (University of Cincinnati)

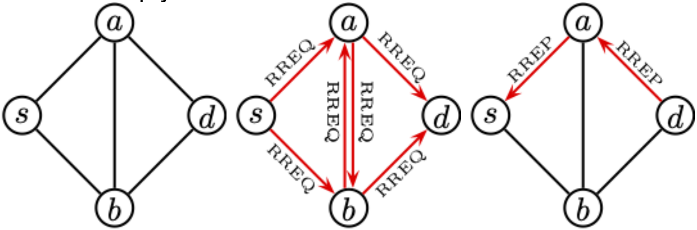
- one of the four protocols standardised by the IETF MANET working group (IEEE 802.11s)

Case Study: AODV

Main Mechanism

- if route is needed
  BROADCAST RREQ
- if node has information about a destination
  UNICAST RREP
- if unicast fails or link break is detected
  GROUPCAST RERR

- performance improvement via intermediate route reply



Case Study: AODV

Formal Specification Language (Process Algebra)

$X(exp_1, \ldots, \mathrm{exp}_n)$	process calls
$P + Q$	nondeterministic
$[\varphi]P$	if-construct (guard)
$[\![\mathtt{var} := exp]\!]P$	assignment followed
$\mathbf{broadcast}(ms).P$	broadcast
$\mathbf{groupcast}(dests, ms).P$	groupcast
$\mathbf{unicast}(dest, ms).P \blacktriangleright Q$	unicast
$\mathbf{send}(ms).P$	send
$\mathbf{receive}(\mathtt{msg}).P$	receive
$\mathbf{deliver}(data).P$	deliver

Case Study: AODV

Specification

```
+ [ (oip, rreqid) ∉ rreqs ]        /* the RREQ is new to this node */
  ⟦rt := update(rt,(oip,osn,kno,val,hops+1,sip,∅))⟧        /* update the route to oip in rt */
  ⟦rreqs := rreqs ∪ {(oip,rreqid)}⟧        /* update rreqs by adding (oip, rreqid) */
  (
    [ dip = ip ]        /* this node is the destination node */
      ⟦sn := max(sn,dsn)⟧        /* update the sqn of ip */
      /* unicast a RREP towards oip of the RREQ */
      unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) . AODV(ip,sn,rt,rreqs,store)
      ► /* If the transmission is unsuccessful, a RERR message is generated */
        ⟦dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}⟧
        ⟦rt := invalidate(rt,dests)⟧
        ⟦store := setRRF(store,dests)⟧
        ⟦pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}⟧
        ⟦dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}⟧
        groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
    + [ dip ≠ ip ]        /* this node is not the destination node */
      (
        [dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqnf(rt,dip) = kno ]        /* valid route to dip that is fresh enough */
          /* update rt by adding precursors */
          ⟦rt := addpreRT(rt,dip,{sip})⟧
          ⟦rt := addpreRT(rt,oip,{nhop(rt,dip)})⟧
          /* unicast a RREP towards the oip of the RREQ */
          unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
```

Case Study: AODV

Full specification of AODV (IETF Standard)

Specification details

- around 5 types and 30 functions
- around 120 lines of specification
  (in contrast to 40 pages English prose)

Properties of AODV

route correctness	✓	
loop freedom	✓	(for some interpretations)
route discovery	✗	
packet delivery	✗	

Final Oral Exam

- **6–10 November, 2021**
- 30 minutes oral examination
- read the guidelines (available via Wattle)
- send through the signed statement in time

GOOD LUCK

Feedback

Please provide feedback

- types of possible feedback
  - suggestions
  - improvements
- send feedback
  - SELT
  - to me (orally, written)

The ‘Final’ Slide

- Q/A sessions
  - Thursday, November 2 (11am-12pm),
  - topics: all questions you prepare
  - no questions, no session
- I hope you. . .
  - had some fun (I had),
    even despite the challenging times
  - learnt something useful

COMP3610/6361 done – what's next?

- COMP3630/6363 (S1 2024)
  Theory of Computation

- COMP4011/8011 (S2 2022)
  Special Topic: Software Verification using Proof Assistants

- Individual Projects/Honour's Theses/PhD projects . . .
  (potentially casual jobs)

Logic Summer School

December 04 – December 15, 2021

Lectures include

- Fundamentals of Metalogic
  (John Slaney, ANU)
- Defining and Reasoning About Programming Languages
  (Fabian Muehlboeck, ANU)
- Propositions and Types, Proofs and Programs
  (Ranald Clouston, ANU)
- Gödel's Theorem Without Tears
  (Dominik Kirst, Ben-Gurion University)
- Foundations for Type-Driven Probabilistic Modelling
  (Ohad Kammar, U Edinburgh)
- ...

Registration is A$150

`http://comp.anu.edu.au/lss`

— THE END —