



University of
Sheffield

LECTURE 4: SEMANTICS AND ABSTRACTION

MODULAR PROOFS IN ISABELLE/HOL

CHELSEA EDMONDS | c.l.edmonds@sheffield.ac.uk

Midlands Graduate School 2025 |

University of Sheffield

COURSE OVERVIEW

A practical course on the effective use of the Isabelle/HOL proof assistant in mathematics and programming languages

Lectures:

- Introduction to Proof Assistants
- Formalising the basics in Isabelle/HOL
- Introduction to Isar, more types, Locales and Type-classes
- Case studies:
 - Formalising Mathematics: combinatorics & advanced locale reasoning patterns
 - **Semantics, Abstraction, PL: Formalising semantics, program properties, and introducing modularity/abstraction.**

Example Classes:

- Isabelle exercises based on the previous lecture
- Will be drawing from the existing Isabelle tutorials/Nipkow's Concrete Semantic Book, as well as custom exercises (e.g. for locales).

LECTURE 4 OVERVIEW

Modular proofs = an engineering-like approach to formalisation.

Yesterday: mathematical formalisations/case-study

TODAY:

- Program verification and proof assistants
- Review: operational semantics
- Formalising semantics and working with basic properties
- Examples of locales/modularity in program verification
 - Refinement +
 - Abstract reasoning
- Proof assistants in the wider-research landscape.



PROGRAM VERIFICATION & PROOF ASSISTANTS



OVERVIEW OF VERIFICATION RESEARCH

The development of several proof assistants was (and continues to be) motivated by program verification in many cases. Isabelle's AFP is split fairly evenly between "Computer Science" and "Mathematics"

Some historical/long running applications:

- Intel HOL-light (Floating Point verification): <https://www.cl.cam.ac.uk/~jrh13/papers/sfm.pdf>
- Sel4 (Isabelle): first formally verified operating system <https://sel4.systems/About/>

Currently

- Increasingly seen in industry (proof assistants are no longer just the domain of research!).
- Many projects focusing on various verification tasks, frameworks for other work etc:
 - AutoCorres (C verification), hoare/separation logic reasoning, algorithm verification etc.



SEMANTICS REVIEW



SEMANTICS INTRODUCTION

- We'll consider *operational* semantics, which can be given inductively:
 - Specifying syntax
 - Expression evaluation
 - Command Execution
- Typically, properties are proven using induction.
- We also often want to develop a type system for our semantics to reason on properties such as progress etc.

LET'S CONSIDER A BASIC SMALL-STEP SEMANTICS

- In the “Concrete Semantics” textbook (Nipkow & Klein, 2014), a basic “IMP” language is introduced. We’ll use this as our initial case study today:

$com ::= \text{SKIP} \mid string ::= aexp \mid com ;; com \mid \text{IF } bexp \text{ THEN } com \text{ ELSE } com \mid \text{WHILE } bexp \text{ DO } com$

$$\frac{}{(x ::= a, s) \rightarrow (SKIP, s(x := aval\ a\ s))} \text{Assign}$$

$$\frac{}{(SKIP ;; c_2, s) \rightarrow (c_2, s)} \text{Seq1} \quad \frac{(c_1, s) \rightarrow (c'_1, s')}{(c_1 ;; c_2, s) \rightarrow (c'_1 ;; c_2, s')} \text{Seq2}$$

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_1, s)} \text{IfTrue}$$

$$\frac{\neg bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_2, s)} \text{IfFalse}$$

$$\frac{}{(WHILE\ b\ DO\ c, s) \rightarrow (IF\ b\ THEN\ c ;; WHILE\ b\ DO\ c\ ELSE\ SKIP, s)} \text{While}$$

AND HERE'S AN EQUIVALENT BIG STEP SEMANTICS

- In the “Concrete Semantics” textbook (Nipkow & Klein, 2014), a basic “IMP” language is introduced. We’ll use this as our initial case study today:

$com ::= \text{SKIP} \mid string ::= aexp \mid com ;; com \mid \text{IF } bexp \text{ THEN } com \text{ ELSE } com \mid \text{WHILE } bexp \text{ DO } com$

$$\begin{array}{c}
 \frac{}{(SKIP, s) \Rightarrow s} \text{Skip} \qquad \frac{}{(x ::= a, s) \Rightarrow s(x := \text{aval } a \ s)} \text{Assign} \\
 \frac{(c_1, s_1) \Rightarrow s_2 \quad (c_2, s_2) \Rightarrow s_3}{(c_1 ;; c_2, s_1) \Rightarrow s_3} \text{Seq} \\
 \frac{bval \ b \ s \quad (c_1, s) \Rightarrow t}{(IF \ b \ THEN \ c_1 \ ELSE \ c_2, s) \Rightarrow t} \text{IfTrue} \\
 \frac{\neg bval \ b \ s \quad (c_2, s) \Rightarrow t}{(IF \ b \ THEN \ c_1 \ ELSE \ c_2, s) \Rightarrow t} \text{IfFalse} \\
 \frac{\neg bval \ b \ s}{(WHILE \ b \ DO \ c, s) \Rightarrow s} \text{WhileFalse} \\
 \frac{bval \ b \ s_1 \quad (c, s_1) \Rightarrow s_2 \quad (WHILE \ b \ DO \ c, s_2) \Rightarrow s_3}{(WHILE \ b \ DO \ c, s_1) \Rightarrow s_3} \text{WhileTrue}
 \end{array}$$



SEMANTICS IN ISABELLE



DATATYPES

- This is a basic command datatype in Isabelle (from the IMP language) with custom syntax

datatype

```
com = SKIP
    | Assign vname aexp      (<_ ::= _> [1000, 61] 61)
    | Seq    com  com        (<_ ;;/ _> [60, 61] 60)
    | If     bexp com  com    (<(IF _/ THEN _/ ELSE _)> [0, 0, 61] 61)
    | While  bexp com        (<(WHILE _/ DO _)> [0, 61] 61)
```

- We could either define elements of **com** abstractly or concretely:

Abstract

```
type_synonym aexp = "state  $\Rightarrow$  val"
type_synonym bexp = "state  $\Rightarrow$  bool"
```

Concrete

```
datatype bexp = Bc bool | Not bexp |
    And bexp bexp | Less aexp aexp

datatype aexp = N int | V vname | Plus aexp aexp
```

DATATYPES

- Say for example we wanted to also make our **com** definition more abstract.
- As Isabelle's datatypes allow for parameterisation, it is quite easy to do this!
- In the example below, parametrised **com** with two additional type parameters instead of using a more concrete *aexp* and *bexp*, noting *Assign* has also been generalised to an Atomic command.

```
datatype ('atom, 'test)com =  
  Done  
|Atom 'atom  
|Seq (leftSeq: "('atom, 'test)com") "('atom, 'test)com" ("_ $$ _" [61, 60] 60)  
|If 'test "('atom, 'test)com" "('atom, 'test)com" ("(if (_)/ {_}/ else/ {_})" [0, 0, 61  
|While 'test "('atom, 'test)com" ("(while (_)/ {_})" [0, 61] 61)
```

ASIDE: INDUCTIVE SETS AND PREDICATES

Inductive Set Approach

```
inductive_set  
Evens :: "nat set"  
where  
zeroIn: "0 ∈ Evens"  
| EvensI: "n ∈ Evens  $\implies$  Suc (Suc n) ∈ Evens"
```

Inductive Predicate

```
inductive evn :: "nat  $\Rightarrow$  bool" where  
zero: "evn 0" |  
step: "evn n  $\implies$  evn (Suc (Suc n))"
```

- After functions and datatypes, inductive definitions are one of the more valuable basic features of Isabelle
- They generate numerous useful facts (induct rules, cases etc).
- For semantics, we typically use the predicate style (as we also are often dealing with quite complex “triples”).

SMALL STEP DEFINITION

- Like other definitions, inductive definitions allow us to specify special syntax.

inductive

small_step :: "com * state \Rightarrow com * state \Rightarrow bool" (**infix** $\langle \rightarrow \rangle$ 55)

where

Assign: "(x ::= a, s) \rightarrow (SKIP, s(x := aval a s))" |

Seq1: "(SKIP;; c₂, s) \rightarrow (c₂, s)" |

Seq2: "(c₁, s) \rightarrow (c₁', s') \implies (c₁;; c₂, s) \rightarrow (c₁';; c₂, s')" |

IfTrue: "bval b s \implies (IF b THEN c₁ ELSE c₂, s) \rightarrow (c₁, s)" |

IfFalse: " \neg bval b s \implies (IF b THEN c₁ ELSE c₂, s) \rightarrow (c₂, s)" |

While: "(WHILE b DO c, s) \rightarrow
(IF b THEN c;; WHILE b DO c ELSE SKIP, s)"

AUTOMATION AND RULE INVERSION

- We can make induction rules more useful by “reformatting it”, such as splitting into pairs:

```
lemmas small_step_induct = small_step.induct[split_format(complete)]
```

- This adds the automatically generated “introduction” rules to the simp/intro sets (so tactics like `auto` will automatically use them).

```
declare small_step.intros[simp,intro]
```

- Rule inversion: We can also use “inductive cases” to get our rule inversion facts of our semantics for free!

```
inductive_cases SkipE[elim!]: "(SKIP, s) → ct"  
thm SkipE
```

☒ Proof state ☒ Auto hovering ☒ Auto

```
(SKIP, ?s) → ?ct ⇒ ?P
```

A SAMPLE PROOF: DETERMINISTIC

- We can use our inductive rules easily as normal, and for simple facts the proofs can be very fast!

```
lemma deterministic:
  "cs → cs'  $\implies$  cs → cs''  $\implies$  cs'' = cs'"
  apply (induction arbitrary: cs'' rule: small_step.induct)
  apply blast+
  done
```

- DEMO! More of the IMP theory



BUT WHAT ABOUT MODULARITY?



INTRODUCING A BASIC LOCALE

```
locale Step =  
fixes  
small_step :: "'com × 'state ⇒ 'com × 'state ⇒ bool" (infix ":->" 60) and  
final :: "'com × 'state ⇒ bool"  
begin
```

abbreviation

```
small_steps :: "'com × 'state ⇒ 'com × 'state ⇒ bool" (infix "→**" 55)  
where "x →** y == star small_step x y"
```

- A basic locale which represents a context that defines a “small step” semantics with a “final” operator (i.e. representing a program terminating)
- Other useful definitions and properties can now be defined/proven locally.

DEFINING ABSTRACT PROPERTIES

- Consider introducing a Hoare logic.
- We can abstractly define if a Hoare triple is valid without needing to know anything about the semantics is valid. And therefore other useful lemmas on this definition!

```
locale Step =  
  fixes  
  small_step :: "'com × 'state ⇒ 'com × 'state ⇒ bool" (infix "→" 55) and  
  final :: "'com × 'state ⇒ bool"  
begin
```

```
abbreviation  
  small_steps :: "'com × 'state ⇒ 'com × 'state ⇒ bool" (infix "→**" 55)  
  where "x →** y == star small_step x y"
```

```
definition  
  hoare_valid :: "('state ⇒ bool) ⇒ 'com ⇒ ('state ⇒ bool) ⇒ bool" ( $\models$  { (1_) } / ( _ ) / { (1_) } > 50) where  
  " $\models$  {P}c{Q} = ( $\forall s\ t\ c'. P\ s \wedge (c, s) \rightarrow^{**} (c', t) \wedge final\ (c', t) \longrightarrow Q\ t$ )"
```

end

Using locale
parameters

INHERITANCE

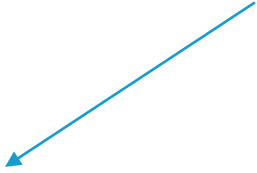
- We can use locales in all the same way we used them for mathematics, including inheritance.
- For example, we may want to abstractly reason on a semantics with a deterministic characteristic.

```
locale Step_Deterministic = Step +  
  assumes determ: " $((cs \rightarrow cs') \wedge (cs \rightarrow cs'')) \longrightarrow (cs'' = cs')$ "
```

INTERPRETING THE LOCALE (CONCRETE)

- Interpreting the **Step** locale with our concrete small-step semantics from earlier is trivial, as there are no assumptions!

```
interpretation Step small_step final .
```



- Similarly, we can use our deterministic lemma from earlier to establish an interpretation for our abstract deterministic locale

```
interpretation det: Step_Deterministic small_step final
  apply (unfold_locales)
  using deterministic by blast
```

OR “REFINING” THE LOCALE

- As is typical of program verification, we often want to gradually refine our specification, rather than jump straight to a concrete definition.
- This is an example of a locale which contains a concrete inductive definition of the semantics, that assumes the existence of evaluation functions for arithmetic and Boolean expressions

```
locale SeqParWhileLang =  
fixes evalA :: "'atom  $\Rightarrow$  'state  $\Rightarrow$  'state"  
and evalT :: "'test  $\Rightarrow$  'state  $\Rightarrow$  bool"  
and Skip :: 'atom and Not :: "'test  $\Rightarrow$  'test"  
assumes evalA_Skip_id[simp,intro!]: "evalA Skip = id"  
and evalT_Not[simp]: " $\wedge s.$  evalT (Not t) s = ( $\neg$  evalT t s)"
```

- We use **sublocale** to establish the relationship

```
sublocale SeqParWhileLang < Step where  
small_step = small_step and final = final .
```

REFINEMENT MORE GENERALLY

- A locale can also be thought of as a specification in the refinement context.
- This is particularly useful in verifying algorithms/data-structures.
- Example of a basic Queue specification (for research examples in data structures/algorithms, see work by Nipkow, Abdulaziz etc)

```
locale Queue =
  fixes empty :: "'q"
  fixes enq :: "'a ⇒ 'q ⇒ 'q"
  fixes first :: "'q ⇒ 'a"
  fixes deq :: "'q ⇒ 'q"
  fixes is_empty :: "'q ⇒ bool"
  fixes list :: "'q ⇒ 'a list"
  fixes invar :: "'q ⇒ bool"
  assumes list_empty: "list empty = []"
  assumes list_enq: "invar q ⇒ list(enq x q) = list q @ [x]"
  assumes list_deq: "invar q ⇒ list(deq q) = tl(list q)"
  assumes list_first: "invar q ⇒ ¬ list q = [] ⇒ first q = hd(list q)"
  assumes list_is_empty: "invar q ⇒ is_empty q = (list q = [])"
  assumes invar_empty: "invar empty"
  assumes invar_enq: "invar q ⇒ invar(enq x q)"
  assumes invar_deq: "invar q ⇒ invar(deq q)"
```



MORE ADVANCED CASE STUDIES



EXAMPLE 1: MORE ABSTRACT PROPERTIES

- In some recent joint work (w/ A. Popescu & J. Wright), we needed to abstractly reason on safety for Rely-Guarantee reasoning, and could then show our theorem held for any small step semantics, as well as interpret it for practical use.

```
context Step  
begin
```

Open abstract
context

```
inductive safe :: "nat  $\Rightarrow$  'com  $\times$  'state  $\Rightarrow$   
  ('state  $\Rightarrow$  'state  $\Rightarrow$  bool)  $\Rightarrow$   
  ('state  $\Rightarrow$  'state  $\Rightarrow$  bool)  $\Rightarrow$   
  ('state  $\Rightarrow$  bool)  $\Rightarrow$   
  bool" where
```

New local definition of
desired property

```
Zero: "safe 0 (c,s) R G Q"
```

```
|  
Suc: "( $\wedge$  c' s'. small_step (c,s) (c',s')  $\Rightarrow$  G s s'  $\wedge$  safe n (c',s') R G Q)  
   $\Rightarrow$   
  ( $\wedge$  s'. R s s'  $\Rightarrow$  safe n (c,s') R G Q)  
   $\Rightarrow$   
  (final (c,s)  $\Rightarrow$  Q s)  
   $\Rightarrow$   
  safe (Suc n) (c,s) R G Q"
```

Using locale
parameters

EXAMPLE 2: MODELLING ATTACKER LEVELS

- Information-flow security investigates if any information can leak from “high valued” variables to “low security” variables through the execution of a program
- Relative security focuses on checking if an enhanced (e.g. optimized) system, is secure with respect to the original (“vanilla”) system (i.e. if any leaks occur, they already occurred in the basic version).
- We can model the idea of “leaks” in different ways, depending on how “abstract” a property we want to reason on.
- For further details:
 - See the original conference paper here (B. Dongol, M. Griffin, A. Popescu, J. Wright, 2024): https://andreipopescu.uk/pdf/relative_security_CSF_2024.pdf
 - The AFP Entry here: https://www.isa-afp.org/entries/Relative_Security.html

EXAMPLE 2: MODELLING ATTACKER LEVELS

- What does it mean for a program to “leak?”
- We first model the system as a transition system (i.e. a set of states, initial states, and transition relation).
- We then introduce the abstract notion of a “leak”, and a predicate “leak via” which given two traces, indicates if they exhibit a leak
- But what is a leak? For this we need to be able to reason on an “attacker model”, i.e what are the secrets, possible observations, and actions (with respect to execution traces).
- Relative Security then requires reasoning on two instances of a system, with a more concrete notion of leaks.

EXAMPLE 2: SETTING UP A TRANSITION SYSTEM

- The base transition system locale

```
locale Transition_System =  
  fixes istate :: "'state  $\Rightarrow$  bool"  
  and validTrans :: "'trans  $\Rightarrow$  bool"  
  and srcOf :: "'trans  $\Rightarrow$  'state"  
  and tgtOf :: "'trans  $\Rightarrow$  'state"
```

- A transition system that includes the definition of finality as an assumption

```
locale System_Mod =  
  Simple_Transition_System istate validTrans  
  for istate :: "'state  $\Rightarrow$  bool"  
  and validTrans :: "'state  $\times$  'state  $\Rightarrow$  bool"  
  +  
  fixes final :: "'state  $\Rightarrow$  bool"  
  assumes final_def: "final s1  $\longleftrightarrow$  ( $\forall$  s2.  $\neg$  validTrans (s1,s2))"
```

EXAMPLE 2: MODELLING ATTACKER LEVELS

- Leakage Model: Assumes the existence of some function describing leaks

```
locale Leakage_Mod = System_Mod istate validTrans final
  for istate :: "'state  $\Rightarrow$  bool" and validTrans :: "'state  $\times$  'state  $\Rightarrow$  bool"
  and final :: "'state  $\Rightarrow$  bool"
+
fixes lleakVia :: "'state llist  $\Rightarrow$  'state llist  $\Rightarrow$  'leak  $\Rightarrow$  bool"
```

- Attacker Model: Specifies the leak via function using more precise functions on secrets, attackers and observers

```
locale Attacker_Mod = System_Mod istate validTrans final
  for istate :: "'state  $\Rightarrow$  bool" and validTrans :: "'state  $\times$  'state  $\Rightarrow$  bool"
  and final :: "'state  $\Rightarrow$  bool"
+
fixes S :: "'state llist  $\Rightarrow$  'secret llist"
and A :: "'state ltrace  $\Rightarrow$  'act llist"
and O :: "'state ltrace  $\Rightarrow$  'obs llist"
begin

fun lleakVia :: "'state llist  $\Rightarrow$  'state llist  $\Rightarrow$  'secret llist  $\times$  'secret llist  $\Rightarrow$  bool"
where
"lleakVia tr tr' (sl,sl') = (S tr = sl  $\wedge$  S tr' = sl'  $\wedge$  A tr = A tr'  $\wedge$  O tr  $\neq$  O tr')"
```

EXAMPLE 2: MODELLING ATTACKER LEVELS

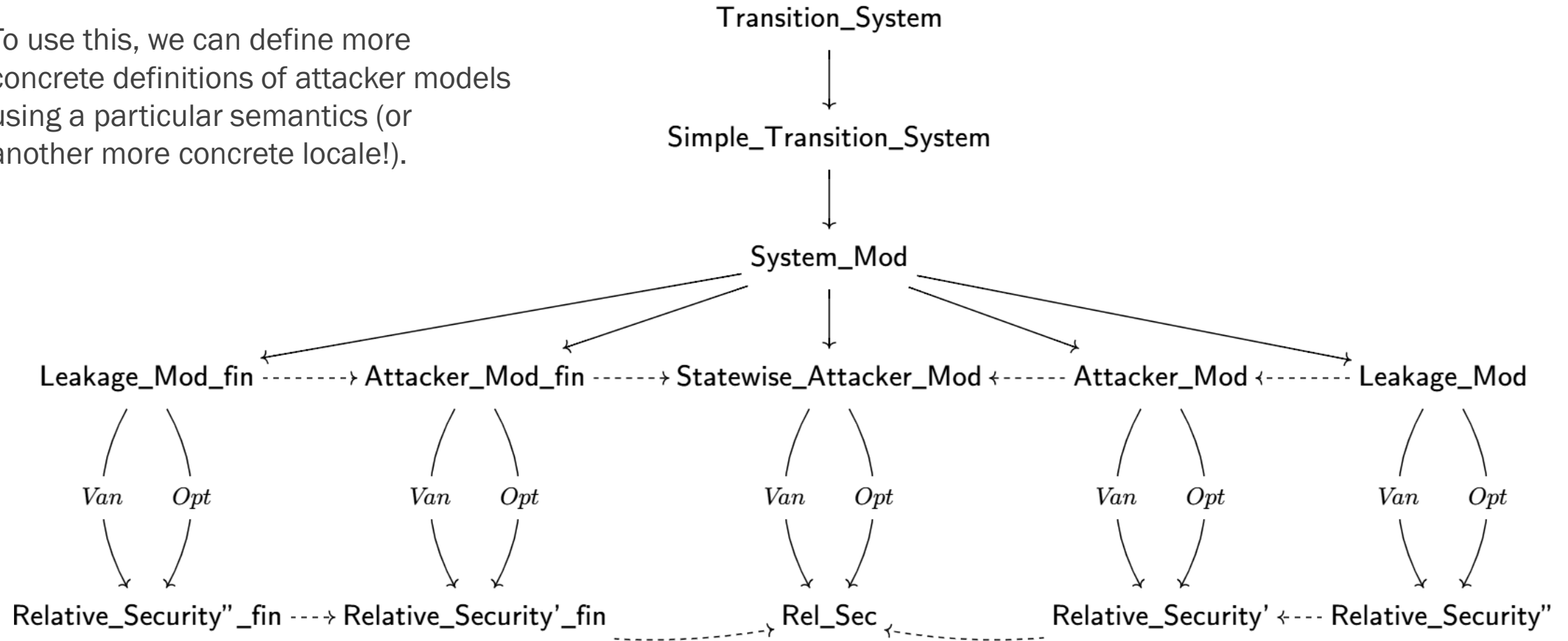
- Relative Security – uses two instances of attacker models.

```
local Rel_Sec =  
  Van: Statewise_Attacker_Mod istateV validTransV finalV isSecV getSecV isIntV getIntV  
  +  
  Opt: Statewise_Attacker_Mod istate0 validTrans0 final0 isSec0 getSec0 isInt0 getInt0  
  for validTransV :: "'stateV × 'stateV ⇒ bool"  
  and istateV :: "'stateV ⇒ bool" and finalV :: "'stateV ⇒ bool"  
  and isSecV :: "'stateV ⇒ bool" and getSecV :: "'stateV ⇒ 'secret"  
  and isIntV :: "'stateV ⇒ bool" and getIntV :: "'stateV ⇒ 'actV × 'obsV"  
  (* NB: we have the same notion of secret, but everything else can be different *)  
  and validTrans0 :: "'state0 × 'state0 ⇒ bool"  
  and istate0 :: "'state0 ⇒ bool" and final0 :: "'state0 ⇒ bool"  
  and isSec0 :: "'state0 ⇒ bool" and getSec0 :: "'state0 ⇒ 'secret"  
  and isInt0 :: "'state0 ⇒ bool" and getInt0 :: "'state0 ⇒ 'act0 × 'obs0"  
  and corrState :: "'stateV ⇒ 'state0 ⇒ bool"
```

- We restate all the parameters using **for** to keep our custom type names

RELATIVE SECURITY FINAL LOCALE INFRASTRUCTURE

To use this, we can define more concrete definitions of attacker models using a particular semantics (or another more concrete locale!).



DEMO AGAIN

CONCLUSION

Any feedback/questions/
thoughts? Feel free to get in
touch at:

c.l.edmonds@sheffield.ac.uk

We've covered

- A fast-paced introduction to the basics of Isabelle/HOL!
- An in-depth discussion of type classes and locales, including advanced reasoning patterns on locales.
- An introduction to reasoning on semantics in Isabelle/HOL
- Research case studies: formalized combinatorics, relative security.

And along the way:

- Some history (proof assistants, formalised maths, verification)
- Insight and links to current research in proof assistants/formal verification

CONCLUSION

Any feedback/questions/
thoughts? Feel free to get in
touch at:

c.l.edmonds@sheffield.ac.uk

Your Challenge:

- Try out formalising your own work in Isabelle (or any other proof assistant out there).
- Keep “software engineering” principles in mind:
 - Verification is only half the goal
 - Modular, reusable, and maintainable formal proof libraries can go much further!

More Resources:

- To continue the work we started on semantics today, see Nipkow and Klein’s book: <http://concrete-semantics.org>
- CPP/ITP are good starting points for formalisation focused research.
- See more links at start of lecture 1!