# CS477 Formal Software Dev Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides based in part on previous lectures
by Mahesh Vishwanathan, and by Gul Agha

February 28, 2018



## Embedding logics in HOL

- Problem: How to define logics and their meaning in HOL?
- Two approaches: *deep* or *shallow*
- Shallow: use propositions of HOL as propositions of defined logic
- Example of shallow: Propositional Logic in HOL (just restrict the terms)
  - Can't always have such a simple inclusion
  - Reasoning easiest in "defined" logic when possible
  - Can't reason *about* defined logic this way, only in it.



## Embedding logics in HOL

- Alternative - Deep:
  - Terms and propositions: elements in data types,
  - Assignment: function from variables (names) to values
  - "Satisfies": function of assignment and proposition to booleans
  - Can always be done
  - More work to define, more work to use than shallow embedding
  - More powerful, can reason about defined logic as well as in it
- Can combine two approaches



## What is the Meaning of a Hoare Triple?

- Hoare triple $\{P\}\ C\ \{Q\}$ means that
  - if $C$ is run in a state $S$ satisfying $P$, and $C$ terminates
  - then $C$ will end in a state $S'$ satisfying $Q$
- Implies states $S$ and $S'$ are (can be viewed as) assignments of variables to values
- States are abstracted as functions from variables to values
- States are modeled as functions from variables to values



## How to Define Hoare Logic in HOL?

- Deep embeeding always possible, more work
- Is shallow possible?
- Two parts: Code and conditions
- Shallowest possible:
  - Code *is* function from states to states
  - Expression *is* function from states to values
  - Boolean expression *is* function from states to booleans
  - Conditions *are* function from states to booleans, since boolean expressions occur in conditions
- Problem: Can't do case analysis on general type of functions from states to states
- Can't do case analysis or induction on code
- Solution: go a bit deeper



## Embedding Hoare Logic in HOL

- Recursive data type for Code (think BNF Grammar)
- Keep expressions, boolean expressions almost as before
- Expressions: functions from states to values
- Boolean expressions: functions from states to booleans
- Conditions: function from states to booleans (i.e. boolean expressions)
- **Note**: Constants, variables are expressions, so are functions from states to values
- What functions are they?

## HOL Types for Shallow Part of Embedding

```
type_synonym var_name = "string"
type_synonym 'data state = "var_name ⇒'data"
type_synonym 'data exp = "'data state ⇒'data"
```

- We are parametrizing by `'data`
- Can instantiate later with `int` of `real`, or role your own



## HOL Terms for Shallow Part of Embedding

Need to lift constants, variables, boolean and arithmetic operators to functions over states:

- Constants:
  ```
  definition k :: "'data ⇒'data exp" where
  "k c ≡λs. c"
  ```
- Variables:
  ```
  definition rev_app :: "var_name ⇒'data exp" ("($)")
  where "$ x ≡λs. s x"
  ```
- We will add more when we specify a specific type of data



## Boolean Expressions

- Can be complete about boolean
  ```
  type_synonym 'data bool_exp = "'data state ⇒bool"

  definition Bool :: "bool ⇒'data bool_exp" where
  "Bool b s = b"

  definition true_b:: "'data bool_exp" where
  "true_b ≡λs. True"

  definition false_b:: "'data bool_exp" where
  "false_b ≡λs. False"
  ```



## Boolean Connectives

- We want the usual logical connectives no matter what type data has:
  ```
  definition and_b
  ::"'data bool_exp ⇒'data bool_exp ⇒'data bool_exp"
  (infix "[∧]" 100) where
  "(a [∧] b) ≡ λs. ((a s) ∧(b s))"

  definition and_b
  ::"'data bool_exp ⇒'data bool_exp ⇒'data bool_exp"
  (infix "[∨]" 100) where
  "(a [∨] b) ≡ λs. ((a s) ∨(b s))"
  ```



## Meaning of Satisfaction

- Need to be able to ask when a state satisfies, or models a proposition:

```
definition models :: "'data state ⇒'data bool_exp ⇒bool"
(infix "⊨" 90)
where
"(s⊨b) ≡b s"

definition bvalid :: "'data bool_exp ⇒bool" ("|⊨")
where
" |⊨b ≡(∀s. b s)"
```



## Reasoning about Propositions

Show the inference rules for Propositional Logic hold here:

```
lemma bvalid_and_bI:
"⟦ |⊨P; |⊨Q⟧ ⟹ |⊨(P [∧] Q)"

lemma bvalid_and_bE [elim]:
"⟦ |⊨(P [∧] Q); ⟦ |⊨P; |⊨Q⟧ ⟹R⟧ ⟹R"

lemma bvalid_or_bLI [intro]: " |⊨P ⟹ |⊨(P [∨] Q)"

lemma bvalid_or_bRI [intro]: " |⊨Q ⟹ |⊨(P [∨] Q)"
```

## How to Handle Substitution

Use the shallowness

```
definition substitute :: "('data state ⇒ 'a) ⇒ var_name ⇒ '
      ("_/[_/⇐_ /]" [120,120,120]60)
 where
"p[x⇐ e] ≡ λ s. p(λ v. if v = x then e(s) else s(v))"
```

Prove this satisfies all equations for substitution:

```
lemma same_var_subst: "$x[x⇐ e] = e"
lemma diff_var_subst: "⟦x ≠ y⟧ ⟹ $y[x⇐ e] = $y"
lemma plus_e_subst:
 "(a [+] b)[x⇐ e] = (a[x⇐ e])[+](b[x⇐ e])"
lemma less_b_subst:
 "(a [<] b)[x⇐ e] = (a[x⇐ e])[<](b[x⇐ e])"
```



## HOL Type for Deep Part of Embedding

```
datatype command =
   AssignCom "var_name" "'data exp"         (infix "::=" 61)
 | SeqCom "command" "command"          (infixl ";;" 60)
 | CondCom "'data bool_exp" "command" "command"
       ("IF _/ THEN _/ ELSE _/ FI" [0,0,0]60)
 | WhileCom "'data bool_exp" "command"
       ("WHILE _/ DO _/ OD" [0,0]60)
```



## Defining Hoare Logic Rules

```
inductive valid :: "'data bool_exp ⇒command ⇒'data bool_exp
⇒'data bool"
("{{_}}_{{_}}" [120,120,120]60)where
AssignmentAxiom:
"{{(P[x⇐e])}}(x::=e) {{P}}" |
SequenceRule:
"⟦{{P}}C {{Q}}; {{Q}}C' {{R}}⟧
⟹{{P}}(C;C'){{R}}" |
RuleOfConsequence:
"⟦|⊨(P [⟶] P') ; {{P'}}C{{Q'}}; |⊨(Q' [⟶] Q) ⟧
⟹{{P}}C{{Q}}" |
IfThenElseRule:
"⟦{{(P [∧] B)}}C{{Q}}; {{(P[∧]([¬]B))}}C'{{Q}}⟧
⟹{{P}}(IF B THEN C ELSE C' FI){{Q}}" |
WhileRule:
"⟦{{(P [∧] B)}}C{{P}}⟧
⟹{{P}}(WHILE B DO C OD){{(P [∧] ([¬]B))}}"
```



## Using Shallow Part of Embedding

- Need to fix a type of data.
- Will fix it as int:

  ```
  type_synonym data = "int"
  ```
- Need to lift constants, variables, arithmetic operators, and predicates to functions over states
- Already have constants (via k) and variables (via $).
- Arithmetic operations:

  ```
  definition plus_e :: "exp ⇒exp ⇒exp" (infixl "[+]" 150)
  where "(p [+] q) ≡ λs. (p s + (q s))"
  ```

Example: $x \times x + (2 \times x + 1)$ becomes

```
"$''x'' [×] $''x'' [+] k 2 [×] $''x'' [+] k 1)"
```



## Using Shallow Part of Embedding

- Arithmetic relations:

  ```
  definition less_b :: "exp ⇒exp ⇒'data bool_exp"
  (infix "[<]" 140) where "(a [<] b)s ≡(a s) < (b s)"
  ```
- Boolean operators:

Example: $x < 0 \wedge y \neq z$ becomes

```
"$''x'' [<] k 0 [∧] [¬]($''y'' [=] $''z'')"
```



## Annotated Simple Imperative Language

- We will give verification conditions for an annotated version of our simple imperative language
- Add a presumed invariant to each while loop

$\langle command \rangle$ ::= $\langle variable \rangle$ := $\langle term \rangle$
| $\langle command \rangle$; ...; $\langle command \rangle$
| *if* $\langle' datastatement \rangle$ *then* $\langle command \rangle$ *else* $\langle command \rangle$
| *while* $\langle' datastatement \rangle$ *inv* $\langle' datastatement \rangle$ *do* $\langle command \rangle$

## Hoare Logic for Annotated Programs

Assingment Rule

$$\frac{}{\{\!|P[e/x]|\!\}\ x\ :=\ e\ \{\!|P|\!\}}$$

Rule of Consequence

$$\frac{P \Rightarrow P' \quad \{\!|P'|\!\}\ C\ \{\!|Q'|\!\} \quad Q' \Rightarrow Q}{\{\!|P|\!\}\ C\ \{\!|Q|\!\}}$$

Sequencing Rule

$$\frac{\{\!|P|\!\}\ C_1\ \{\!|Q|\!\} \quad \{\!|Q|\!\}\ C_2\ \{\!|R|\!\}}{\{\!|P|\!\}\ C_1;\ C_2\ \{\!|R|\!\}}$$

If Then Else Rule

$$\frac{\{\!|P \wedge B|\!\}\ C_1\ \{\!|Q|\!\} \quad \{\!|P \wedge \neg B|\!\}\ C_2\ \{\!|Q|\!\}}{\{\!|P|\!\}\ \mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2\ \{\!|Q|\!\}}$$

While Rule

$$\frac{\{\!|P \wedge B|\!\}\ C\ \{\!|P|\!\}}{\{\!|P|\!\}\ \mathit{while}\ B\ \mathit{inv}\ P\ \mathit{do}\ C\ \{\!|P \wedge \neg B|\!\}}$$



DEMO