Embedding logics in HOL

- Problem: How to define logic 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.

Deep embedding always possible, more work

- 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 embedding 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

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 `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 "∧") where "(a ∧ b) s ≡ λs. ((a s) ∧ (b s))"
  - `definition and_b ::"'data bool_exp ⇒ 'data bool_exp ⇒ 'data bool_exp" (infix "∨") where "(a ∨ b) s ≡ 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 "|="= λs. (Vs. 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_bI [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 ⇒ 'a"**

where

\[ p[x:= e] \equiv \lambda s \cdot p(\lambda v \cdot \text{if } v = x \text{ then } s(s) \text{ else } s(v)) \]

Prove this satisfies all equations for substitution:

1. **lemma name_var_subst:** \$x \cdot [x := e] \equiv e$
2. **lemma diff_var_subst:** \$x \cdot [x \neq y] \equiv \lambda y \cdot \text{if } y = x \text{ then } s(s) \text{ else } s(v)\$

HOL Type for Deep Part of Embedding

datatype command =

- AssignCom "var name" "'data exp" (infixl ":[:=]" 110)
- SeqCom "command" "command" (infixl ":" 109)
- CondCom "'data bool_exp" "command" "command"
  ("IF \ / THEN \ / ELSE \ / FI" [120,120,120]60)
- WhileCom "'data bool_exp" "command"
  ("WHILE \ / DO \ / OD" [120,120]60)

Defining Hoare Logic Rules

**inductive valid :: "'data bool_exp ⇒command ⇒'data bool_exp ⇒'data bool"**

**AssignAxiom:**

\[ \{P\} C \{Q\} \Rightarrow \{(P)\} (\{C\}) \{\{R\}\} \]

**SequenceRule:**

\[ \{P\} C \{Q\} \Rightarrow \{(P)\} C\{Q\} \{\{R\}\} \]

\[ \{(P)\} (\{C\}) \{\{R\}\} \]

**RuleOfConsequence:**

\[ \{P\} C \{Q\} \Rightarrow \{(P)\} C\{Q\} \{\{R\}\} \]

**IfThenElseRule:**

\[ \{IF B THEN C ELSE C' FI\} \{Q\} \Rightarrow \{P\} B \{Q\} \{R\} \]

**WhileRule:**

\[ \{P\} (WHILE B DO C OD) \{Q\} \Rightarrow \{P\} (\{C\}) \{Q\} \{\{R\}\} \]

Using Shallow Part of Embedding

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

Need to fix a type of data:

- Need to lift constants, variables, arithmetic operators, and predicates to functions over states
- Already have constants (via k) and variables (via $)
- Arithmetic operations:

\[ \lambda x \cdot [x = 0 \land y \neq z \Rightarrow x < 0 \land \forall t \in \mathbb{R}. \exists t' \in \mathbb{R}. \forall t'' \in \mathbb{R}. \forall t''' \in \mathbb{R}. t' = t'' = t''' = \ldots] \]

- Example: \$x \times x + (2 \times x + 1) \Rightarrow (x \times x + 1)$

Using Shallow Part of Embedding

- **Arithmetic relations:**
  - **definition less_b :: "exp ⇒exp ⇒'data bool_exp"**
    (infix "[<]" 140) where \( (a [\leq] b) \equiv (a \leq b) \)

- **Boolean operators:**

\[ x < 0 \land y \neq z \Rightarrow x < 0 \land \forall t \in \mathbb{R}. \exists t' \in \mathbb{R}. \forall t'' \in \mathbb{R}. \forall t''' \in \mathbb{R}. t' = t'' = t''' = \ldots] \]

DEMO
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

\[
\langle \text{command} \rangle ::= \langle \text{variable} \rangle ::= \langle \text{term} \rangle |
\langle \text{command} \rangle ; \ldots ; \langle \text{command} \rangle |
\text{if} \langle \text{datastatement} \rangle \text{then} \langle \text{command} \rangle \text{else} \langle \text{command} \rangle |
\text{while} \langle \text{datastatement} \rangle \text{inv} \langle \text{datastatement} \rangle \text{do} \langle \text{command} \rangle
\]

Hoare Logic for Annotated Programs

**Assignment Rule**

\[
\{ P[e/x] \} x := e \{ P \}
\]

**Rule of Consequence**

\[
P \Rightarrow P' \quad \{ P' \} C \{ Q' \} \quad Q' \Rightarrow Q
\]

**Sequencing Rule**

\[
\{ P \} C_1 \{ Q \} \quad \{ Q \} C_2 \{ R \}
\]

\[
\{ P \} C_1, C_2 \{ R \}
\]

**If Then Else Rule**

\[
\{ P \land B \} C_1 \{ Q \} \quad \{ P \land \neg B \} C_2 \{ Q' \}
\]

\[
\{ P \} \text{if} B \text{then} C_1 \text{else} C_2 \{ Q \}
\]

**While Rule**

\[
\{ P \land B \} C \{ P \}
\]

\[
\{ P \} \text{while} B \text{inv} P \text{do} C \{ P \land \neg B \}
\]