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.

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

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

- We are parametrizing by \texttt{data}
- Can instantiate later with \texttt{int} or \texttt{real}, or role your own

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 7 / 20

HOL Terms for Shallow Part of Embedding

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

- Constants:
  - \texttt{definition k :: 'data state \Rightarrow 'data exp} where \texttt{\lambda s. c}

- Variables:
  - \texttt{definition rev_app :: var name \Rightarrow 'data exp} ('$(\)) where \texttt{\lambda s. s x}

We will add more when we specify a specific type of data

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 8 / 20

Boolean Expressions

- Can be complete about boolean
  - \texttt{type synonym 'data bool exp = 'data state \Rightarrow bool}

  \texttt{definition Bool :: bool \Rightarrow 'data bool exp} where \texttt{\lambda s. b}

  \texttt{definition true_b :: 'data bool exp} where \texttt{\lambda s. True}

  \texttt{definition false_b :: 'data bool exp} where \texttt{\lambda s. False}

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 9 / 20

Boolean Connectives

- We want the usual logical connectives no matter what type data has:
  - \texttt{definition and_b :: 'data bool exp \Rightarrow 'data bool exp \Rightarrow 'data bool exp} (infix \texttt{\&\&} 100) where \texttt{\lambda s. (a \&\& b s)}

  \texttt{definition or_b :: 'data bool exp \Rightarrow 'data bool exp \Rightarrow 'data bool exp} (infix \texttt{||} 100) where \texttt{\lambda s. (a || b s)}

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 10 / 20

Meaning of Satisfaction

- Need to be able to ask when a state satisfies, or \texttt{models} a proposition:

  \texttt{definition models :: 'data state \Rightarrow 'data bool exp \Rightarrow bool} (infix \texttt{\models} 90) where \texttt{(s \models b) \equiv b s}

  \texttt{definition bvalid :: 'data bool exp \Rightarrow bool} (* \texttt{\models} *) where \texttt{\models b \equiv (\forall s. b s)}

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 11 / 20

Reasoning about Propositions

Show the inference rules for Propositional Logic hold here:

\texttt{lemma bvalid_and_bI:}  
\texttt{\[ (\models P; (\models Q) \Rightarrow (\models (P \& Q))\]}

\texttt{lemma bvalid_and_bE [elim]}:  
\texttt{\[ (\models (P \& Q); (\models P; (\models Q) \Rightarrow R) \Rightarrow R\]}

\texttt{lemma bvalid_or_bI [intro]}:  
\texttt{\[ (\models P \Rightarrow (\models (P \Rightarrow Q))\]}

\texttt{lemma bvalid_or_bRI [intro]}:  
\texttt{\[ (\models Q \Rightarrow (\models (P \Rightarrow Q))\]}

Elsa L Gunter  
CS477 Formal Software Dev Methods  
February 28, 2018 12 / 20
Defining Hoare Logic Rules

How to Handle Substitution

Use the shallowness

\[
\text{definition substitute} :: \"(\text{data state} \Rightarrow \text{a}) \Rightarrow \text{var name} \Rightarrow \" (\\text{\"/\"}/\\text{-\"/\"}) [120,120,120]60)\]

where
\[
\text{p}[x\Rightarrow e] \equiv \lambda s . \text{p}(\lambda \nu . \text{if } v = x \text{ then } s(s) \text{ else } s(v))\]

Prove this satisfies all equations for substitution:

\[
\text{lemma name_var_subst: } \$x[x\Rightarrow e] = e\]
\[
\text{lemma diff_var_subst: } \[x \neq y \Rightarrow \$$y[x\Rightarrow e] = \$y\]
\[
\text{lemma plus_b_subst: } \left( (a \{+\} b)[x\Rightarrow e] = (a[x\Rightarrow e])\{+\}(b[x\Rightarrow e]) \right)\]
\[
\text{lemma less_b_subst: } \left( (a \{<\} b)[x\Rightarrow e] = (a[x\Rightarrow e])\{<\}(b[x\Rightarrow e]) \right)\]

HOL Type for Deep Part of Embedding

Using Shallow Part of Embedding

- Need to fix a type of data.
  
  \[\text{type synonym data} = \"\text{int}\"\]

- Need to lift constants, variables, arithmetic operators, and predicates to functions over states

- Already have constants (via \$k\) and variables (via \$\$\$\).

- Arithmetic operations:

\[
\text{definition plus}_b :: \"\text{exp} \Rightarrow \text{exp} \Rightarrow \text{data bool exp} \Rightarrow \text{data bool}\"
\]

Arithmetic relations:

\[
\text{definition less}_b :: \"\text{exp} \Rightarrow \text{exp} \Rightarrow \text{data bool exp} \Rightarrow \text{data bool}\"
\]

Boolean operators:

\[
\text{definition substitute :: \"(\text{data state} \Rightarrow \text{a}) \Rightarrow \text{var name} \Rightarrow \text{data exp} \Rightarrow \text{data exp}\" (infixl \"::=\" 61)}\]

\[
\text{Annotated Simple Imperative Language}
\]

Using Shallow Part of Embedding

- Arithmetic relations:

\[
\text{definition less}_b :: \"\text{exp} \Rightarrow \text{exp} \Rightarrow \text{data bool exp} \Rightarrow \text{data bool}\"
\]

- Boolean operators:

\[
\text{Example: } x < 0 \land y \neq z \text{ becomes}
\]

\[
\$\$'$'$x'$'$ [x] \times [k] \{\land\} \{\neg\} (\$\$'$'$y'$'$ [z] \times [k]) \]

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

\[
\text{(command)} ::= \text{(variable)} ::= \text{(term)} \\
\text{(command)} ::= \text{(command)} \text{; \ldots} \text{; (command)} \\
\text{if} \text{(datastatement)} \text{then (command)} \text{else (command)} \\
\text{while} \text{(datastatement)} \text{inv} \text{(datastatement)} \text{do (command)}
\]
Assignment Rule
\[ \frac{P}{P[e/x]} \]
\[ x := e \] \[ P \]

Rule of Consequence
\[ \frac{P \Rightarrow P'}{P'} \]
\[ C \] \[ \frac{Q' \Rightarrow Q}{Q} \]
\[ C \] \[ \frac{Q'}{Q} \]

Sequencing Rule
\[ \frac{P \quad Q \quad R}{P \quad C_1 \quad Q \quad C_2 \quad R} \]

If Then Else Rule
\[ \frac{P \land B \quad C_1 \quad Q \quad P \land \neg B \quad C_2 \quad Q}{P \quad \text{if } B \text{ then } C_1 \quad \text{else } C_2 \quad Q} \]

While Rule
\[ \frac{P \land B \quad C \quad P}{P \quad \text{while } B \text{ inv } P \text{ do } C \quad P \land \neg B} \]