

We need to describe the syntax and semantics of our programming language.

```

COMMAND ::= skip
           | VARIABLE := EXPRESSION
           | COMMAND; COMMAND
           | if BOOLEAN then COMMAND else COMMAND

EXPRESSION ::= VARIABLE
              | CONSTANT
              | EXPRESSION + EXPRESSION
              | EXPRESSION - EXPRESSION

BOOLEAN ::= EXPRESSION = EXPRESSION
            | EXPRESSION < EXPRESSION
            | not BOOLEAN
            | BOOLEAN and BOOLEAN
            | BOOLEAN or BOOLEAN
    
```

```

val expression_AX = define_type {
  name = "expression_AX",
  type_spec = `expression = VAR of string
              | CONST of num
              | PLUS of expression * expression
              | MINUS of expression * expression`,
  fixities = [Prefix, Prefix, Infix 1000, Infix 1000]}

val boolean_AX = define_type {
  name = "boolean_AX",
  type_spec = `boolean = EQUAL of expression * expression
              | LESS of expression * expression
              | NOT of boolean
              | AND of boolean * boolean
              | OR of boolean * boolean`,
  fixities = [Infix 200, Infix 200, Prefix, Infix 100, Infix 100]}

val command_AX = define_type {
  name = "command_AX",
  type_spec = `command = SKIP
              | := of string * expression
              | ;; of command * command
              | IF of boolean * command * command`,
  fixities = [Prefix, Infix 20, Infix 10, Prefix]}
    
```

Semantics: Assigning Meaning to Programs

The state is a function from variables (strings) to values (numbers).

- Meaning of expressions: $\llbracket _ \rrbracket_{\mathcal{E}}: \text{expression} \rightarrow \text{state} \rightarrow \mathbb{N}$

$$\begin{aligned} \llbracket \text{VAR } v \rrbracket_{\mathcal{E}} \sigma &\stackrel{\text{def}}{=} \sigma v \\ \llbracket e_1 \text{ PLUS } e_2 \rrbracket_{\mathcal{E}} \sigma &\stackrel{\text{def}}{=} \llbracket e_1 \rrbracket_{\mathcal{E}} \sigma + \llbracket e_2 \rrbracket_{\mathcal{E}} \sigma \\ \llbracket e_1 \text{ MINUS } e_2 \rrbracket_{\mathcal{E}} \sigma &\stackrel{\text{def}}{=} \llbracket e_1 \rrbracket_{\mathcal{E}} \sigma - \llbracket e_2 \rrbracket_{\mathcal{E}} \sigma \\ \llbracket \text{CONST } c \rrbracket_{\mathcal{E}} \sigma &\stackrel{\text{def}}{=} c \end{aligned}$$

- Meaning of booleans: $\llbracket _ \rrbracket_{\mathcal{B}}: \text{boolean} \rightarrow \text{state} \rightarrow \mathbb{B}$

$$\begin{aligned} \llbracket e_1 \text{ EQUAL } e_2 \rrbracket_{\mathcal{B}} \sigma &\stackrel{\text{def}}{=} \llbracket e_1 \rrbracket_{\mathcal{E}} \sigma = \llbracket e_2 \rrbracket_{\mathcal{E}} \sigma \\ \llbracket \text{NOT } b \rrbracket_{\mathcal{B}} \sigma &\stackrel{\text{def}}{=} \neg \llbracket b \rrbracket_{\mathcal{B}} \sigma \\ \llbracket v_1 \text{ OR } v_2 \rrbracket_{\mathcal{B}} \sigma &\stackrel{\text{def}}{=} \llbracket v_1 \rrbracket_{\mathcal{B}} \sigma \vee \llbracket v_2 \rrbracket_{\mathcal{B}} \sigma \\ \llbracket e_1 \text{ LESS } e_2 \rrbracket_{\mathcal{B}} \sigma &\stackrel{\text{def}}{=} \llbracket e_1 \rrbracket_{\mathcal{E}} \sigma < \llbracket e_2 \rrbracket_{\mathcal{E}} \sigma \\ \llbracket b_1 \text{ AND } b_2 \rrbracket_{\mathcal{B}} \sigma &\stackrel{\text{def}}{=} \llbracket b_1 \rrbracket_{\mathcal{B}} \sigma \wedge \llbracket b_2 \rrbracket_{\mathcal{B}} \sigma \end{aligned}$$

- Meaning of commands: $\llbracket _ \rrbracket_{\mathcal{C}}: \text{command} \rightarrow \text{state} \rightarrow \text{state}$

$$\begin{aligned} \llbracket \text{SKIP} \rrbracket_{\mathcal{C}} \sigma &\stackrel{\text{def}}{=} \sigma \\ \llbracket c_1 ;; c_2 \rrbracket_{\mathcal{C}} \sigma &\stackrel{\text{def}}{=} \llbracket c_2 \rrbracket_{\mathcal{C}} (\llbracket c_1 \rrbracket_{\mathcal{C}} \sigma) \\ \llbracket v := e \rrbracket_{\mathcal{C}} \sigma &\stackrel{\text{def}}{=} (\lambda w. \text{if } w = v \text{ then } \llbracket e \rrbracket_{\mathcal{E}} \sigma \text{ else } \sigma w) \\ \llbracket \text{IF } b \text{ then } c_1 \text{ else } c_2 \rrbracket_{\mathcal{C}} \sigma &\stackrel{\text{def}}{=} \text{if } \llbracket b \rrbracket_{\mathcal{B}} \sigma \text{ then } \llbracket c_1 \rrbracket_{\mathcal{C}} \sigma \text{ else } \llbracket c_2 \rrbracket_{\mathcal{C}} \sigma \end{aligned}$$

Hoare Logic

A specialised proof systems for programs can make proofs easier:

The Hoare Triple: If C starts in a state satisfying P and terminates, it does so in a state satisfying Q .

$$\{P\} C \{Q\}$$

Hoare Rules:

$$\begin{aligned} &\frac{\vdash \forall \sigma. \{P\}_{\mathcal{C}} \sigma \supset \{Q\}_{\mathcal{C}} \sigma}{\vdash \{P\} \text{SKIP} \{Q\}} \\ &\frac{\vdash \forall \sigma. \{P\}_{\mathcal{C}} \sigma \supset \{Q[E/v]\}_{\mathcal{C}} \sigma}{\vdash \{P\} v := E \{Q\}} \\ &\frac{\vdash \{P\} C_1 \{R\} \quad \vdash \{R\} C_2 \{Q\}}{\vdash \{P\} C_1 ;; C_2 \{Q\}} \\ &\frac{\vdash \{P \text{ AND } B\} C_1 \{Q\} \quad \vdash \{P \text{ AND } (\text{NOT } B)\} C_2 \{Q\}}{\vdash \{P\} \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \{Q\}} \end{aligned}$$

Example Hoare Logic Proof

1 $\{(x \text{ EQUALS } X) \text{ AND } (y \text{ EQUALS } Y)\}$
 $t := x ;; x := y ;; y := t$
 $\{(x \text{ EQUALS } Y) \text{ AND } (y \text{ EQUALS } X)\}$

By ;; rule we get two subproofs:

1.1 $\{(x \text{ EQUALS } X) \text{ AND } (y \text{ EQUALS } Y)\}$
 $t := x ;; x := y$
 $\{(x \text{ EQUALS } Y) \text{ AND } (t \text{ EQUALS } X)\}$

1.2 $\{(x \text{ EQUALS } Y) \text{ AND } (t \text{ EQUALS } X)\}$
 $y := t$
 $\{(x \text{ EQUALS } Y) \text{ AND } (y \text{ EQUALS } X)\}$

By := rule we get one subproof

1.2.1 $\forall \sigma. \{(x \text{ EQUALS } Y) \text{ AND } (t \text{ EQUALS } X)\}_{\mathcal{B}} \sigma \supset$
 $\{(x \text{ EQUALS } Y) \text{ AND } (y \text{ EQUALS } X)\}_{\mathcal{B}} (\llbracket t/y \rrbracket_{\mathcal{B}} \sigma)$
 $= \forall \sigma. \{(x \text{ EQUALS } Y) \text{ AND } (t \text{ EQUALS } X)\}_{\mathcal{B}} \sigma \supset$
 $\{(x \text{ EQUALS } Y) \text{ AND } (t \text{ EQUALS } X)\}_{\mathcal{B}} \sigma$
 $= \top$

Hoare Logic into HOL

We must show that each Hoare logic rule follows from our semantics.

For example, for SKIP we must prove

$$\forall P Q. (\forall \sigma. \{P\}_{\mathcal{C}} \sigma \supset \{Q\}_{\mathcal{C}} \sigma) \supset \{P\} \text{SKIP} \{Q\}$$

Other rules are similar, but assignment is special:

$$\forall P Q v E. (\forall \sigma. \{P\}_{\mathcal{C}} \sigma \supset \{Q[E/v]\}_{\mathcal{C}} \sigma) \supset \{P\} v := E \{Q\}$$

We need to formalise syntactic substitution:

```

val SUBSTe_DEF = new_recursive_definition {
  name = "SUBSTe_DEF",
  fixity = Prefix,
  rec_axiom = expression_AX,
  def = Term `
    SUBSTe (VAR w) E v = (if w = v then E else (VAR w))
  ^ SUBSTe (CONST c) E v = CONST c
  ^ SUBSTe (e1 PLUS e2) E v = (SUBSTe e1 E v) PLUS (SUBSTe e2 E v)
  ^ SUBSTe (e1 MINUS e2) E v = (SUBSTe e1 E v) MINUS (SUBSTe e2 E v)
  `
    
```

More About Substitution

```

val SUBSTb_DEF = new_recursive_definition {
  name = "SUBSTb_DEF",
  fixity = Prefix,
  rec_axiom = boolean_AX,
  def = Term `
    SUBSTb (e1 EQUALS e2) E v = (SUBSTe e1 E v) EQUALS (SUBSTe e2 E v)
  ^ SUBSTb (e1 LESS e2) E v = (SUBSTe e1 E v) LESS (SUBSTe e2 E v)
  ^ SUBSTb (NOT e) E v = NOT (SUBSTb e E v)
  ^ SUBSTb (e1 AND e2) E v = (SUBSTb e1 E v) AND (SUBSTb e2 E v)
  ^ SUBSTb (e1 OR e2) E v = (SUBSTb e1 E v) OR (SUBSTb e2 E v)
  `
    
```

The relationship between substituting expressions and updating the state:

$$\text{update } v \text{ in } \sigma \stackrel{\text{def}}{=} (\lambda w. \text{if } w = v \text{ then } \sigma w)$$

- $\vdash \forall EX v \sigma. \llbracket E[X/v] \rrbracket_{\mathcal{E}} \sigma = \llbracket E \rrbracket_{\mathcal{E}} (\text{update } v (\llbracket X \rrbracket \sigma))$
- $\vdash \forall BX v \sigma. \llbracket B[X/v] \rrbracket_{\mathcal{B}} \sigma = \llbracket B \rrbracket_{\mathcal{B}} (\text{update } v (\llbracket X \rrbracket \sigma))$

We can now prove the Hoare assignment rule.

Shallow Versus Deep Embedding

We identified hardware descriptions with a subset of expressions of an existing type in HOL. This is called a *shallow embedding*.

For programs we defined a new type for the syntax of programs and then defined a meaning function to map that syntax into an existing type. This is a *deep embedding*.

- ✗ Deep embeddings are more work.
- ✗ Deep embeddings duplicate features of the logic.
- ✓ Deep embeddings allow reasoning at a syntactic level.
- ✓ Deep embeddings can prove general properties of programs, e.g.

$$\vdash \forall c_1 c_2 c_3. \llbracket c_1 ;; (c_2 ;; c_3) \rrbracket_{\mathcal{C}} = \llbracket (c_1 ;; c_2) ;; c_3 \rrbracket_{\mathcal{C}}$$