Content

- Intro & motivation, getting started with Isabelle
- Foundations & Principles
  - Lambda Calculus
  - Higher Order Logic, natural deduction
  - Term rewriting
- Proof & Specification Techniques
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - More recursion, Calculational reasoning
  - Hoare logic, proofs about programs
  - Locales, Presentation

Proofs about Programs

Now we know:
- What programs are: Syntax
- On what they work: State
- How they work: Semantics

So we can prove properties about programs

Example:
Show that example program from last lecture implements the factorial.

\[
\text{lemma} \quad (\text{factorial}, \sigma) \rightarrow \sigma' \quad \sigma' B = \text{fac} (\sigma A)
\]

(\text{where} \quad \text{fac} 0 = 0, \quad \text{fac} (\text{Suc} \ n) = (\text{Suc} \ n) \times \text{fac} \ n)
Too tedious

Induction needed for each loop
Is there something easier?

Slide 5

Floyd/Hoare

Idea: describe meaning of program by pre/post conditions

Examples:
- (True) $x := 2 \quad \{ x = 2 \}$
- ($y = 2$) $x := 21 \times y \quad \{ x = 42 \}$
- ($x = n$) IF $y < 0$ THEN $x := x + y$ ELSE $x := x - y$ \quad \{ $x = n - |y|$ \}
- ($A = n$) factorial \quad \{ $B = \text{fac } n$ \}

Proofs: have rules that directly work on such triples

Slide 6

Meaning of a Hoare-Triple

$\{ P \} \ c \ \{ Q \}$

What are the assertions $P$ and $Q$?
- Here: again functions from state to bool (shallow embedding of assertions)
- Other choice: syntax and semantics for assertions (deep embedding)

What does $\{ P \} \ c \ \{ Q \}$ mean?

Partial Correctness:
$\models \{ P \} \ c \ \{ Q \} \iff (\forall \sigma. P \sigma \implies \langle c, \sigma \rangle \implies \sigma' \implies Q \sigma')$

Total Correctness:
$\models \{ P \} \ c \ \{ Q \} \iff (\exists \sigma. P \sigma \implies \langle c, \sigma \rangle \implies \sigma' \land Q \sigma')$

This lecture: partial correctness only (easier)

Slide 7

Hoare Rules

$\begin{align*}
\{ P \} & \quad \text{SKIP} & \quad \{ P \} \\
\{ P \} & \quad x := e & \quad \{ P \}
\end{align*}$

$\begin{align*}
\{ P \} & \quad c_1 (R) & \quad \{ R \} & \quad c_2 (Q) \\
\{ P \} & \quad c_1; c_2 & \quad \{ Q \}
\end{align*}$

$\begin{align*}
\{ P \land b \} & \quad c_1 (Q) & \quad \{ P \land \neg b \} & \quad c_2 (Q) \\
\{ P \} & \quad \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2 & \quad \{ Q \}
\end{align*}$

$\begin{align*}
\{ P \land b \} & \quad e (P) & \quad \{ P \land \neg b \} & \quad e (Q) \\
\{ P \} & \quad \text{WHILE } b \ \text{DO } e & \quad \{ Q \}
\end{align*}$

$\begin{align*}
P & \implies P' & \quad \{ P' \} & \quad e (Q') & \quad Q' & \implies Q
\end{align*}$

Slide 8
Hoare Rules

⊢ {P} \text{SKIP} {P}

⊢ {\lambda \sigma. P (x := e \sigma)} \quad x := e \quad {P}

⊢ {P} c_1 (R) \quad \vdash {\{R\} c_2 (Q)}

⊢ {\lambda \sigma. P \sigma \land b \sigma} c_1 \quad {P}

⊢ {\lambda \sigma. P \sigma \land \neg b \sigma} c_2 \quad {Q \sigma}

⊢ {P} \quad \text{WHILE} b \text{ DO } c \text{ OD} \quad {Q}$

\[ \lambda \sigma. P \sigma \Rightarrow P' \sigma \quad \vdash {P'} c (Q') \quad \land \sigma. Q' \sigma \Rightarrow Q \sigma \]

\[ \vdash {P} \quad c \quad {Q} \]

Are the Rules Correct?

Soundness: \( \vdash {P} \quad c \quad {Q} \Rightarrow \vdash {P} \quad c \quad {Q} \)

Proof: by rule induction on \( \vdash {P} \quad c \quad {Q} \)

Demo: Hoare Logic in Isabelle

Nicer, but still kind of tedious

Hoare rule application seems boring & mechanical.

Automation?

Problem: While – need creativity to find right (invariant) \( P \)

Solution:

\( \Rightarrow \) annotate program with invariants

\( \Rightarrow \) then, Hoare rules can be applied automatically

Example:

\[ \{ M = 0 \land N = 0 \} \quad \text{WHILE} M \neq a \text{ INV} \{ N = M + b \} \text{ DO } N := N + b; M := M + 1 \text{ OD} \quad \{ N = a * b \} \]

Weakest Preconditions

\( \text{pre} \quad c \quad Q \Rightarrow \text{weakest} \quad P \text{ such that} \quad \{P\} \quad c \quad \{Q\} \)

With annotated invariants, easy to get:

\( \text{pre} \quad \text{SKIP} \quad Q \quad = \quad Q \)

\( \text{pre} \quad (x := a) \quad Q \quad = \quad \lambda \sigma. Q (x := a \sigma) \)

\( \text{pre} \quad (c_1; c_2) \quad Q \quad = \quad \text{pre} \quad c_1 \quad (\text{pre} \quad c_2 \quad Q) \)

\( \text{pre} \quad (\text{IF} \ b \text{ THEN} \ c_1 \text{ ELSE} \ c_2) \quad Q \quad = \quad \lambda \sigma. (b \quad \rightarrow \quad \text{pre} \quad c_1 \quad Q \sigma) \quad \land \quad (\neg b \quad \rightarrow \quad \text{pre} \quad c_2 \quad Q \sigma) \)

\( \text{pre} \quad (\text{WHILE} \ b \text{ INV} \ I \text{ DO} \ c \text{ OD}) \quad Q \quad = \quad I \)
Verification Conditions

\{(pre c Q) ∈ \{Q\}\} only true under certain conditions

These are called verification conditions \(vc\ c\ Q\):  
\(vc\ SKIP\ Q\) = True 
\(vc\ (x := a)\ Q\) = True 
\(vc\ (c_1; c_2)\ Q\) = \(vc\ c_2\ Q \land (vc\ c_1\ (pre c_2\ Q))\) 
\(vc\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\) = \(vc\ c_1\ Q \land vc\ c_2\ Q\) 
\(vc\ (WHILE\ b\ INV\ I\ DO\ c\ OD)\ Q\) = \(∀\ σ. Iσ \land bσ \rightarrow (pre\ c\ I σ) \land (νσ. Iσ \land ¬bσ \rightarrow Q σ) \land vc\ c\ I\) 
\(vc\ c\ Q \land (pre\ c\ Q \Rightarrow P) \Rightarrow (P)\ c\ Q\)

Syntax Tricks

- \(x := λσ. 1\) instead of \(x := 1\) sucks
- \(\{λσ. x = n\}\) instead of \(x = n\) sucks as well

Problem: program variables are functions, not values

Solution: distinguish program variables syntactically

Choices:
- declare program variables with each Hoare triple
- nice, usual syntax
- works well if you state full program and only use \(vcg\)
- separate program variables from Hoare triple (use extensible records), indicate usage as function syntactically
- more syntactic overhead
- program pieces compose nicely

Records in Isabelle

Records are a tuples with named components

Example:

\(record\ A = a :: nat\ b :: int\)

- Selectors: \(\{a :: A \Rightarrow nat, b :: A \Rightarrow int, a \cdot r = Suc 0\}\)
- Constructors: \(\{\| a = Suc 0, b = −1\}\)
- Update: \(v\{a := Suc 0\}\)

Records are extensible:

\(record\ B = A +\ c :: nat\ list\)

\(\{\| a = Suc 0, b = −1, c = [0, 0]\}\)

Arrays

Depending on language, model arrays as functions:

- Array access = function application:
  \(a[i] = \ a\ i\)
- Array update = function update:
  \(a[i] := v = \ a\ := a(i := v)\)

Use lists to express length:

- Array access = nth:
  \(a[i] = \ a\ i\)
- Array update = list update:
  \(a[i] := v = \ a\ := a(i := v)\)
- Array length = list length:
  \(a.length = \ length\ a\)

7

8
Pointers

Choice 1

datatype ref = Ref int | Null

|types| heap = int → val |

|datatype| val = Int int | Bool bool | Struct x int int bool |

⇒ hp :: heap, p :: ref
⇒ Pointer access: *p = the_int (hp (the_addr p))
⇒ Pointer update: *p := v = hp := hp ((the_addr p) := v)

⇒ a bit klunky
⇒ gets even worse with structs
⇒ lots of value extraction (the_int) in spec and program

Slide 17

Pointers

Choice 2 (Burstall '72, Bornat '00)

struct with next pointer and element

datatype ref = Ref int | Null

|types| next hp = int → ref |

|types| elem hp = int → int |

⇒ next :: next hp, elem :: elem hp, p :: ref
⇒ Pointer access: p → next = next (the_addr p)
⇒ Pointer update: p → next := v = next := next ((the_addr p) := v)

⇒ a separate heap for each struct field
⇒ buys you p → next ≠ p → elem automatically (aliasing)
⇒ still assumes type safe language

Slide 18

We have seen today ...

⇒ Hoare logic rules
⇒ Soundness of Hoare logic
⇒ Verification conditions
⇒ Example program proofs
⇒ Arrays, pointers

Slide 20