NICTA Advanced Course

Theorem Proving

Principles, Techniques, Applications

\{P\} \ldots \{Q\}
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
LAST TIME

→ Recdef
→ More induction
→ Well founded orders
→ Well founded recursion
→ Calculations: also/finally
→ [trans]-rules
A CRASH COURSE IN SEMANTICS
IMP - a small Imperative Language

Commands:

datatype com = SKIP

| Assign loc aexp (_ := _) |
| Semi com com (_; _) |
| Cond bexp com com (IF _ THEN _ ELSE _) |
| While bexp com (WHILE _ DO _ OD) |
IMP - A SMALL IMPERATIVE LANGUAGE

Commands:

**datatype** com = SKIP

| Assign loc aexp (loc := aexp) |
| Semi com com (com; com) |
| Cond bexp com com (IF bexp THEN com ELSE com) |
| While bexp com (WHILE bexp DO com OD) |

**types** loc = string

**types** state = loc ⇒ nat
Commands:

**datatype** com = SKIP
| Assign loc aexp (\_ := \_)
| Semi com com (\_; \_)
| Cond bexp com com (IF \_ THEN \_ ELSE \_)
| While bexp com (WHILE \_ DO \_ OD)

**types** loc = string

**types** state = loc ⇒ nat

**types** aexp = state ⇒ nat

**types** bexp = state ⇒ bool
Example Program

Usual syntax:

\[ B := 1; \]
\[ \text{WHILE } A \neq 0 \text{ DO} \]
\[ B := B \times A; \]
\[ A := A - 1 \]
\[ \text{OD} \]
Example Program

Usual syntax:

\[
B := 1; \\
\text{WHILE } A \neq 0 \text{ DO} \\
\quad B := B \times A; \\
\quad A := A - 1 \\
\text{OD}
\]

Expressions are functions from state to bool or nat:

\[
B := (\lambda \sigma. 1); \\
\text{WHILE } (\lambda \sigma. \sigma A \neq 0) \text{ DO} \\
\quad B := (\lambda \sigma. \sigma B \times \sigma A); \\
\quad A := (\lambda \sigma. \sigma A - 1) \\
\text{OD}
\]
So far we have defined:
So far we have defined:

- Syntax of commands and expressions
So far we have defined:

- **Syntax** of commands and expressions
- **State** of programs (function from variables to values)

Now we need:
WHAT DOES IT DO?

So far we have defined:

- **Syntax** of commands and expressions
- **State** of programs (function from variables to values)

**Now we need:** the meaning (semantics) of programs

**How to define execution of a program?**
So far we have defined:

- **Syntax** of commands and expressions
- **State** of programs (function from variables to values)

Now we need: the meaning (semantics) of programs

How to define execution of a program?

- A wide field of its own (visit a semantics course!)
WHAT DOES IT DO?

So far we have defined:

- **Syntax** of commands and expressions
- **State** of programs (function from variables to values)

Now we need: the meaning (semantics) of programs

How to define execution of a program?

- A wide field of its own (visit a semantics course!)
- Some choices:
  - Operational (inductive relations, big step, small step)
  - Denotational (programs as functions on states, state transformers)
  - Axiomatic (pre-/post conditions, Hoare logic)
\[ \langle \text{SKIP}, \sigma \rangle \longrightarrow \sigma \]
\[
\langle \text{SKIP}, \sigma \rangle \longrightarrow \sigma
\]

\[
\langle x := e, \sigma \rangle \longrightarrow
\]
Structural Operational Semantics

\[
\langle \text{SKIP}, \sigma \rangle \rightarrow \sigma
\]

\[
e \sigma = v \quad \Rightarrow \quad \langle x := e, \sigma \rangle \rightarrow \sigma[x \leftarrow v]
\]

\[
\langle c_1; c_2, \sigma \rangle \rightarrow \sigma''
\]
**Structural Operational Semantics**

\[
\langle \text{SKIP}, \sigma \rangle \rightarrow \sigma
\]

\[
e \sigma = v
\]

\[
\langle x := e, \sigma \rangle \rightarrow \sigma[x \leftarrow v]
\]

\[
\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma''
\]

\[
\langle c_1 ; c_2, \sigma \rangle \rightarrow \sigma''
\]

\[
b \sigma = \text{True}
\]

\[
\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \rightarrow \sigma'
\]
STRUCTURAL OPERATIONAL SEMANTICS

\[ \langle \text{SKIP}, \sigma \rangle \rightarrow \sigma \]

\[ e \sigma = v \]
\[ \langle x := e, \sigma \rangle \rightarrow \sigma[x \leftarrow v] \]

\[ \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma'' \]
\[ \langle c_1; c_2, \sigma \rangle \rightarrow \sigma'' \]

\[ b \sigma = \text{True} \quad \langle c_1, \sigma \rangle \rightarrow \sigma' \]
\[ \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \rightarrow \sigma' \]

\[ b \sigma = \text{False} \]
\[ \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \rightarrow \sigma' \]
\[
\begin{align*}
\langle \text{SKIP, } \sigma \rangle & \longrightarrow \sigma \\
\frac{e \sigma = v}{\langle x := e, \sigma \rangle \longrightarrow \sigma[x \leftarrow v]} \\
\frac{\langle c_1, \sigma \rangle \longrightarrow \sigma' \quad \langle c_2, \sigma' \rangle \longrightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \longrightarrow \sigma''} \\
\frac{b \sigma = \text{True}}{\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \longrightarrow \sigma'} \\
\frac{b \sigma = \text{False}}{\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \longrightarrow \sigma'}
\end{align*}
\]
\[ \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow \]
\[
\frac{b \sigma = \text{False}}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow \sigma}
\]
\begin{align*}
\text{if } b \sigma = \text{False} \\
\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle & \rightarrow \sigma \\
\text{if } b \sigma = \text{True} \\
\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle & \rightarrow 
\end{align*}
\[
\begin{align*}
\text{if } b \sigma = \text{False} & \quad \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow \sigma \\
\text{if } b \sigma = \text{True} & \quad \langle c, \sigma \rangle \rightarrow \sigma' \quad \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow
\end{align*}
\]
\[
\begin{align*}
&b \sigma = \text{False} \\
&\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
&b \sigma = \text{True} \\
&\langle c, \sigma \rangle \rightarrow \sigma' \\
&\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma' \rangle \rightarrow \sigma'' \\
&\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \rightarrow \sigma''
\end{align*}
\]
DEMO: THE DEFINITIONS IN ISABELLE
Now we know:

➔ What programs are: Syntax
➔ On what they work: State
➔ How they work: Semantics

Example: Show that example program from slide 6 implements the factorial.

\[
\text{factorial} \; 0 = 0 \quad \text{and} \quad \text{factorial} \; (\text{Suc} \; n) = (\text{Suc} \; n) \times \text{factorial} \; n
\]
Now we know:

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

So we can prove properties about programs
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 slide 6 implements the factorial.

\[
\text{lemma } \langle \text{factorial}, \sigma \rangle \longrightarrow \sigma' \implies \sigma' B = \text{fac } (\sigma A)
\]

(Where \( \text{fac } 0 = 0 \), \( \text{fac } (\text{Suc } n) = (\text{Suc } n) \times \text{fac } n \))
Demo: Example Proof
Induction needed for each loop
Induction needed for each loop

Is there something easier?
Idea: describe meaning of program by pre/post conditions

Examples:
Idea: describe meaning of program by pre/post conditions

Examples:

\{ \text{True} \} \quad x := 2 \quad \{ x = 2 \}
Idea: describe meaning of program by pre/post conditions

Examples:
\{\text{True}\} \quad x := 2 \quad \{x = 2\}
\{y = 2\} \quad x := 21 * y \quad \{x = 42\}
**Idea:** describe meaning of program by pre/post conditions

**Examples:**

\[
\begin{align*}
\{ \text{True} \} & \quad x := 2 \quad \{ x = 2 \} \\
\{ y = 2 \} & \quad x := 21 \times y \quad \{ x = 42 \}
\end{align*}
\]

\[
\begin{align*}
\{ x = n \} & \quad \text{IF } y < 0 \text{ THEN } x := x + y \text{ ELSE } x := x - y \quad \{ x = n - |y| \}
\end{align*}
\]
Idea: describe meaning of program by pre/post conditions

Examples:

\{True\} \quad x := 2 \quad \{x = 2\} \\
\{y = 2\} \quad x := 21 \times y \quad \{x = 42\} \\
\{x = n\} \quad IF y < 0 THEN x := x + y ELSE x := x - y \quad \{x = n - |y|\} \\
\{A = n\} \quad \text{factorial} \quad \{B = \text{fac } n\}
Idea: describe meaning of program by pre/post conditions

Examples:
\{\text{True}\} \quad x := 2 \quad \{x = 2\}
\{y = 2\} \quad x := 21 \ast y \quad \{x = 42\}

\{x = n\} \quad \text{IF } y < 0 \text{ THEN } x := x + y \text{ ELSE } x := x - y \quad \{x = n - |y|\}

\{A = n\} \quad \text{factorial} \quad \{B = \text{fac } n\}

Proofs: have rules that directly work on such triples
MEANING OF A HOARE-TRIPLE

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

What are the assertions $P$ and $Q$?
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)
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?
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:
\[
\models \{P\} \ c \ \{Q\} \equiv (\forall \sigma \ \sigma'. \ P \ \sigma \land \langle c, \sigma \rangle \rightarrow \sigma' \rightarrow Q \ \sigma')
\]
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:
\[\models \{P\} \ c \ \{Q\} \equiv (\forall \sigma \sigma'. \ P \sigma \land \langle c, \sigma \rangle \rightarrow \sigma' \rightarrow Q \sigma')\]

Total Correctness:
\[\models \{P\} \ c \ \{Q\} \equiv (\forall \sigma. \ P \sigma \rightarrow \exists \sigma'. \langle c, \sigma \rangle \rightarrow \sigma' \land Q \sigma')\]
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:
$$\models \{P\} \ c \ \{Q\} \equiv (\forall \sigma \ \sigma'. \ P \ \sigma \land \langle c, \sigma \rangle \rightarrow \sigma' \Longrightarrow Q \ \sigma')$$

Total Correctness:
$$\models \{P\} \ c \ \{Q\} \equiv (\forall \sigma. \ P \ \sigma \Longrightarrow \exists \sigma'. \ \langle c, \sigma \rangle \rightarrow \sigma' \land Q \ \sigma')$$

This lecture: partial correctness only (easier)
HOARE RULES

\{P\} \quad \text{SKIP} \quad \{P\}
**HOARE RULES**

\[
\begin{align*}
\{P\} & \text{ SKIP } \{P\} \\
\{P[x \leftarrow e]\} & x := e \{P\}
\end{align*}
\]
**HOARE RULES**

\[
\begin{align*}
\{P\} & \text{ SKIP } \{P\} & \{P[x \leftarrow e]\} & \text{ } x := e & \{P\} \\
\{P\} & c_1 \{R\} & \{R\} & c_2 \{Q\} & \{P\} \\
& c_1; c_2 & \{Q\}
\end{align*}
\]
\[
\begin{align*}
\{P\} & \quad \text{SKIP} & \quad \{P\} \\
\{P[x \leftrightarrow e]\} & \quad x := e & \quad \{P\} \\
\{P\} & \quad c_1 & \quad \{R\} \\
\{R\} & \quad c_2 & \quad \{Q\} \\
\{P\} & \quad c_1; c_2 & \quad \{Q\} \\
\{P\} & \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 & \quad \{Q\}
\end{align*}
\]
HOARE RULES

\[
\begin{align*}
\{P\} & \text{SKIP} & \{P\} \\
\{P[x \leftarrow e]\} & x := e & \{P\} \\
\{P\} & c_1 & \{R\} \\
\{R\} & c_2 & \{Q\} \\
\{P\} & c_1; c_2 & \{Q\} \\
\{P \land b\} & c_1 & \{Q\} \\
\{P\} & \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 & \{Q\}
\end{align*}
\]
**Hoare Rules**

\[
\begin{array}{l}
\{P\} \text{ SKIP} \{P\} \quad \{P[x \leftarrow e]\} \quad x := e \quad \{P\} \\
\{P\} \ c_1 \ {R} \quad {R} \ c_2 \ {Q} \\
\{P\} \ c_1; c_2 \ {Q} \\
\{P \land b\} \ c_1 \ {Q} \quad {P \land \neg b} \ c_2 \ {Q} \\
\{P\} \quad \text{IF } b \text{ THEN } c_1 \ ELSE \ c_2 \ {Q}
\end{array}
\]
**Hoare Rules**

\[
\begin{align*}
\{P\} \text{ SKIP} \{P\} \\
\{P\} c_1 \{R\} \quad \{R\} c_2 \{Q\} \\
\{P\} \quad c_1; c_2 \quad \{Q\} \\
\{P \land b\} c_1 \{Q\} \quad \{P \land \neg b\} c_2 \{Q\} \\
\{P\} \quad \text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \quad \{Q\} \\
\{P \land b\} c \{P\} \quad P \land \neg b \implies Q \\
\{P\} \quad \text{WHILE} \ b \ \text{DO} \ c \ \text{OD} \quad \{Q\}
\end{align*}
\]
\begin{align*}
\{P\} & \quad \text{SKIP} \quad \{P\} \\
\{P\} & \quad c_1 \quad \{R\} \quad \{R\} \quad c_2 \quad \{Q\} \\
\{P\} & \quad c_1; c_2 \quad \{Q\} \\
\{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 \text{ ELSE } c_2 \quad \{Q\} \\
\{P \land b\} & \quad c \quad \{P\} \quad \quad P \land \neg b \quad \implies \quad Q \\
\{P\} & \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\} \\
\{P'\} & \quad c \quad \{Q'\} \\
\{P\} & \quad c \quad \{Q\} 
\end{align*}
**Hoare Rules**

\[
\begin{align*}
\{P\} & \text{ SKIP } \{P\} \\
\{P\} \ c_1 \ \{R\} & \quad \{R\} \ c_2 \ \{Q\} \\
\{P\} & \quad c_1; \ c_2 \ \{Q\} \\
\{P \land b\} \ c_1 \ \{Q\} & \quad \{P \land \neg b\} \ c_2 \ \{Q\} \\
\{P\} & \quad \text{IF } b \text{ THEN } c_1 \ \text{ELSE } c_2 \ \{Q\} \\
\{P \land b\} \ c \ \{P\} & \quad P \land \neg b \implies Q \\
\{P\} & \quad \text{WHILE } b \text{ DO } c \ \text{OD} \ \{Q\} \\
\{P\} & \quad c \ \{Q\}
\end{align*}
\]
**Hoare Rules**

\[ \vdash \{P\} \text{ SKIP} \{P\} \]

\[ \vdash \{P\} c_1 \{R\} \vdash \{R\} c_2 \{Q\} \]

\[ \vdash \{P\} \quad \vdash \{P\} c_1; c_2 \quad \{Q\} \]

\[ \vdash \{\lambda \sigma. P \sigma \land b \sigma\} c_1 \{R\} \quad \vdash \{\lambda \sigma. P \sigma \land \neg b \sigma\} c_2 \{Q\} \]

\[ \vdash \{P\} \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \{Q\} \]

\[ \vdash \{\lambda \sigma. P \sigma \land b \sigma\} c \{P\} \quad \wedge \sigma. P \sigma \land \neg b \sigma \implies Q \sigma \]

\[ \vdash \{P\} \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\} \]

\[ \wedge \sigma. P \sigma \implies P' \sigma \quad \vdash \{P'\} c \{Q'\} \quad \wedge \sigma. Q' \sigma \implies Q \sigma \]

\[ \vdash \{P\} \quad c \quad \{Q\} \]
ARE THE RULES CORRECT?

Soundness: \[ \vdash \{P\} \ c \ \{Q\} \implies \models \{P\} \ c \ \{Q\} \]
ARE THE RULES CORRECT?

**Soundness:** $\vdash \{P\} \ c \ {Q} \implies\models \{P\} \ c \ {Q}$

**Proof:** by rule induction on $\vdash \{P\} \ c \ {Q}$
ARE THE RULES CORRECT?

Soundness: \( \vdash \{ P \} c \{ Q \} \implies \models \{ P \} c \{ Q \} \)

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

Demo: Hoare Logic in Isabelle
Hoare rule application seems boring & mechanical.

Automation?
NicER, but still kind of tedious

Hoare rule application seems boring & mechanical.

Automation?

**Problem:** While – need creativity to find right (invariant) \( P \)
NICER, BUT STILL KIND OF TEDIOUS

Hoare rule application seems boring & mechanical.

Automation?

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

**Solution:**

- annotate program with invariants
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
**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:**

\[
\begin{align*}
\{M = 0 \land N = 0\} \\
\text{WHILE } M \neq a \text{ INV } \{N = M \times b\} \text{ DO } N := N + b; M := M + 1 \text{ OD} \\
\{N = a \times b\}
\end{align*}
\]
**Weakest Preconditions**

\[
\text{pre } c \ Q = \text{weakest } P \text{ such that } \{P\} \ c \ \{Q\}
\]

With annotated invariants, easy to get:
Weakest Preconditions

\[ \text{pre } c \ Q = \text{weakest } P \text{ such that } \{P\} \ c \ \{Q\} \]

With annotated invariants, easy to get:

\[ \text{pre SKIP } Q \]

\[ = \ Q \]
**Weakest Preconditions**

\[ \text{pre } c \ Q = \text{weakest } P \text{ such that } \{P\} \ c \ \{Q\} \]

With annotated invariants, easy to get:

\[
\begin{align*}
\text{pre SKIP } Q &= Q \\
\text{pre } (x := a) \ Q &= \lambda \sigma. \ Q(\sigma(x := a\sigma))
\end{align*}
\]
W e a k e s t  P r e c o n d i t i o n s

\[ \text{pre } c \ Q = \text{weakest } P \text{ such that } \{P\} \ c \ \{Q\} \]

With annotated invariants, easy to get:

\[
\begin{align*}
\text{pre } \text{SKIP } Q & = Q \\
\text{pre } (x := a) \ Q & = \lambda \sigma. \ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1 ; c_2) \ Q & = \text{pre } c_1 \ (\text{pre } c_2 \ Q)
\end{align*}
\]
**Weakest Preconditions**

\[
\text{pre } c Q = \text{weakest } P \text{ such that } \{P\} c \{Q\}
\]

With annotated invariants, easy to get:

\[
\begin{align*}
\text{pre SKIP } Q & = Q \\
\text{pre } (x := a) \ Q & = \lambda \sigma. \ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1; c_2) \ Q & = \text{pre } c_1 \ (\text{pre } c_2 \ Q) \\
\text{pre } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q & = \lambda \sigma. \ (b \rightarrow \text{pre } c_1 \ Q \ \sigma) \land \neg b \rightarrow \text{pre } c_2 \ Q \ \sigma)
\end{align*}
\]
**Weakest Preconditions**

\[ \text{pre } c \ Q = \text{weakest } P \text{ such that } \{ P \} \ c \ { Q} \]

With annotated invariants, easy to get:

- \( \text{pre } \text{SKIP } Q \) = \( Q \)
- \( \text{pre } (x := a) \ Q \) = \( \lambda \sigma. \ Q(\sigma(x := a\sigma)) \)
- \( \text{pre } (c_1; c_2) \ Q \) = \( \text{pre } c_1 \ (\text{pre } c_2 \ Q) \)
- \( \text{pre } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q \) = \( \lambda \sigma. \ (b \rightarrow \text{pre } c_1 \ Q \ \sigma) \land \neg b \rightarrow \text{pre } c_2 \ Q \ \sigma) \)
- \( \text{pre } (\text{WHILE } b \ \text{INV } I \ \text{DO } c \ \text{OD}) \ Q \) = \( I \)
Verification Conditions

\{ \text{pre } c \ Q \} \ c \ \{ Q \} \ \text{only true under certain conditions}
VERIFICATION CONDITIONS

\{ \text{pre } c \ Q \} \ c \ \{ Q \} \text{ only true under certain conditions}

These are called verification conditions $\text{vc } c \ Q$:

$\text{vc SKIP } Q \quad = \quad \text{True}$
These are called **verification conditions** \( \text{vc } c \ Q \):

- \( \text{vc } \text{SKIP } Q \) = True
- \( \text{vc } (x := a) \ Q \) = True
**Verification Conditions**

\[ \{ \text{pre } c \ Q \} \ c \ \{ Q \} \text{ only true under certain conditions} \]

These are called **verification conditions** \( vc \ c \ Q \):  

- \( vc \ \text{SKIP } Q \) \hspace{1cm} = \text{ True} 
- \( vc \ (x := a) \ Q \) \hspace{1cm} = \text{ True} 
- \( vc \ (c_1; c_2) \ Q \) \hspace{1cm} = \( vc \ c_2 \ Q \land (vc \ c_1 \ (\text{pre } c_2 \ Q)) \)
These are called verification conditions $\text{vc } c \ Q$: 

- $\text{vc } \text{SKIP} \ Q = \text{True}$
- $\text{vc } (x := a) \ Q = \text{True}$
- $\text{vc } (c_1; c_2) \ Q = \text{vc } c_2 \ Q \land (\text{vc } c_1 (\text{pre } c_2 \ Q))$
- $\text{vc } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q = \text{vc } c_1 \ Q \land \text{vc } c_2 \ Q$
Verification Conditions

\{\text{pre } c \ Q\} \ c \ \{Q\} \text{ only true under certain conditions}

These are called verification conditions \(\text{vc } c \ Q\):

- \(\text{vc } \text{SKIP } Q\) \hspace{1cm} = \text{True}
- \(\text{vc } (x := a) \ Q\) \hspace{1cm} = \text{True}
- \(\text{vc } (c_1 ; c_2) \ Q\) \hspace{1cm} = \(\text{vc } c_2 \ Q \land (\text{vc } c_1 \ (\text{pre } c_2 \ Q))\)
- \(\text{vc } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q\) \hspace{1cm} = \(\text{vc } c_1 \ Q \land \text{vc } c_2 \ Q\)
- \(\text{vc } (\text{WHILE } b \ \text{INV } I \ \text{DO } c \ \text{OD}) \ Q\) \hspace{1cm} = (\forall \sigma. \ I \sigma \land b \sigma \rightarrow \text{pre } c \ I \ \sigma) \land (\forall \sigma. \ I \sigma \land \neg b \sigma \rightarrow Q \ \sigma) \land \text{vc } c \ I
VERIFICATION CONDITIONS

\{ \text{pre } c \ Q \} \ c \ \{ \ Q \} \text{ only true under certain conditions}

These are called verification conditions \( \text{vc } c \ Q \):

\[
\begin{align*}
\text{vc } \text{SKIP } Q & = \text{ True} \\
\text{vc } (x := a) \ Q & = \text{ True} \\
\text{vc } (c_1; c_2) \ Q & = \text{ vc } c_2 \ Q \land (\text{vc } c_1 \ (\text{pre } c_2 \ Q)) \\
\text{vc } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q & = \text{ vc } c_1 \ Q \land \text{vc } c_2 \ Q \\
\text{vc } (\text{WHILE } b \ \text{INV } I \ \text{DO } c \ \text{OD}) \ Q & = (\forall \sigma. \ I\sigma \land b\sigma \rightarrow \text{pre } c \ I \ \sigma) \land \\
& \quad (\forall \sigma. \ I\sigma \land \neg b\sigma \rightarrow Q \ \sigma) \land \text{vc } c \ I \\
\text{vc } c \ Q \land (\text{pre } c \ Q \rightarrow P) & \rightarrow \{ P \} \ c \ \{ Q \}
\end{align*}
\]
Syntax Tricks

\[ x := \lambda \sigma. 1 \quad \text{instead of} \quad x := 1 \text{ sucks} \]

\[ \{ \lambda \sigma. \sigma \ x = n \} \quad \text{instead of} \quad \{ x = n \} \text{ sucks as well} \]
SYNTAX TRICKS

→  \( x := \lambda\sigma. 1 \)  instead of  \( x := 1 \) sucks

→  \( \{\lambda\sigma. \sigma \; x = n\} \)  instead of  \( \{x = n\} \) sucks as well

**Problem:** program variables are functions, not values


**Syntax Tricks**

- $x := \lambda \sigma. 1$ instead of $x := 1$ sucks
- $\{ \lambda \sigma. \sigma x = n \}$ instead of $\{x = n\}$ sucks as well

**Problem:** Program variables are functions, not values

**Solution:** Distinguish program variables syntactically

---

**Syntax Tricks**


**Syntax Tricks**

→ $x := \lambda \sigma. 1$ instead of $x := 1$ sucks

→ $\{\lambda \sigma. \sigma x = n\}$ 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
SYNTAX TRICKS

→ $x := \lambda \sigma. 1$ instead of $x := 1$ sucks
→ $\{\lambda \sigma. \sigma x = n\}$ 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
SYNTAX TRICKS

→ \( x := \lambda \sigma. 1 \) instead of \( x := 1 \) sucks
→ \( \{ \lambda \sigma. \sigma x = n \} \) 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
Syntax Tricks

\[ x := \lambda \sigma. 1 \] instead of \[ x := 1 \] sucks
\[ \{ \lambda \sigma. \sigma x = n \} \] instead of \[ \{ x = n \} \] sucks as well

Problem: program variables are functions, not values

Solution: distinguish program variables syntactically

Choices:
\[ \rightarrow \] declare program variables with each Hoare triple
- nice, usual syntax
- works well if you state full program and only use vcg

\[ \rightarrow \] 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
Records are a tuples with named components

Example:

```plaintext
record A = a :: nat 
   b :: int
```
Records are a tuples with named components

Example:

```
record A = a :: nat
           b :: int

⇒ Selectors: a :: A ⇒ nat,  b :: A ⇒ int,  a r = Suc 0
```
Records are tuples with named components

Example:

```plaintext
record A =  a :: nat
          b :: int

Selectors:  a :: A ⇒ nat,  b :: A ⇒ int,  a r = Suc 0
Constructors:  (| a = Suc 0,  b = −1 |)
```
Records in Isabelle

Records are a tuples with named components

Example:

```plaintext
record A =  
  a :: nat
  b :: int
```

➔ Selectors:  
  a :: A ⇒ nat,  
  b :: A ⇒ int,  
  a r = Suc 0

➔ Constructors:  
  (\ a = Suc 0,  
  b = -1 |

➔ Update:  
  r(\ a := Suc 0 )
```
Records are a tuples with named components

Example:

```
record A =  a :: nat
          b :: int
```

→ Selectors:  a :: A ⇒ nat,  b :: A ⇒ int,  a r = Suc 0
→ Constructors:  (| a = Suc 0, b = -1 |
→ Update:  r(| a ::= Suc 0 |)

Records are extensible:

```
record B = A +
          c :: nat list
```
Records are a tuples with named components

Example:

\[
\text{record } A = \begin{aligned}
  &a :: \text{nat} \\
  &b :: \text{int}
\end{aligned}
\]

- Selectors: \( a :: A \Rightarrow \text{nat}, \ b :: A \Rightarrow \text{int}, \ a \ r = \text{Suc} \ 0 \)
- Constructors: \( (\mid a = \text{Suc} \ 0, \ b = -1 \mid) \)
- Update: \( r(\mid a ::= \text{Suc} \ 0 \mid) \)

Records are extensible:

\[
\text{record } B = A + \\
  c :: \text{nat list}
\]

\( (\mid a = \text{Suc} \ 0, \ b = -1, \ c = [0, 0] \mid) \)
Available now in Isabelle:

- procedures
Available now in Isabelle:

- procedures
- with blocks and local variables
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
- arrays
Available now in Isablle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
- arrays
- pointers
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
- arrays
- pointers

We’re working at:

- nondeterminism
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
- arrays
- pointers

We’re working at:

- nondeterminism
- probability
Available now in Isabelle:

- procedures
- with blocks and local variables
- and (mutual) recursion
- exceptions
- arrays
- pointers

We’re working at:

- nondeterminism
- probability
- object orientation
WE HAVE SEEN TODAY ...

- Syntax and semantics of IMP
- Hoare logic rules
- Soundness of Hoare logic
- Verification conditions
- Example program proofs


EXERCISES

→ Write a program in IMP that calculates quotient and reminder of
  \( x \in \mathbb{N} \) and \( y \in \mathbb{N} \)

→ Find the right invariant for its while loop.

→ Show its correctness in Isabelle:

\[
\vdash \{\text{True}\} \quad \text{program} \quad \{ \; Q \ast y + R = x \land R < y \; \}
\]

→ Write an IMP program that sorts arrays (lists) by insertion sort.

→ Formulate and show its correctness in Isabelle.