Concrete Semantics
with Isabelle/HOL

Peter Lammich (slides adopted from Tobias Nipkow)

Fakultät für Informatik
Technische Universität München

2018-12-2

Part II

Semantics

Chapter 7

IMP:
A Simple Imperative Language

1. IMP Commands

2. Big-Step Semantics

3. Small-Step Semantics

1. IMP Commands
2. Big-Step Semantics
3. Small-Step Semantics

Terminology

Statement: declaration of fact or claim

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Command: order to do something

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Command: order to do something

Study the book until you have understood it.

Terminology

Statement: declaration of fact or claim

Semantics is easy.

Command: order to do something

Study the book until you have understood it.

Expressions are *evaluated*, commands are *executed*

Commands

Concrete syntax:

$$\begin{array}{rcl} com & ::= & \texttt{SKIP} \\ & | & string \; \texttt{::=} \; aexp \\ & | & com \; \texttt{;;} \; com \\ & | & \texttt{IF} \; bexp \; \texttt{THEN} \; com \; \texttt{ELSE} \; com \\ & | & \texttt{WHILE} \; bexp \; \texttt{DO} \; com \end{array}$$

Commands

Abstract syntax:

$$
\begin{array}{lcl}
\textbf{datatype}\ com & = & \mathit{SKIP} \\
 & | & \mathit{Assign}\ string\ aexp \\
 & | & \mathit{Seq}\ com\ com \\
 & | & \mathit{If}\ bexp\ com\ com \\
 & | & \mathit{While}\ bexp\ com
\end{array}
$$

Com.thy

1. IMP Commands
2. Big-Step Semantics
3. Small-Step Semantics

Big-step semantics

Concrete syntax:

$$(com,\ initial\text{-}state) \Rightarrow final\text{-}state$$

Big-step semantics

Concrete syntax:

$$(com,\ initial\text{-}state) \Rightarrow final\text{-}state$$

Intended meaning of $(c,\ s) \Rightarrow t$:

Big-step semantics

Concrete syntax:

$$(com,\ initial\text{-}state) \Rightarrow final\text{-}state$$

Intended meaning of $(c,\ s) \Rightarrow t$:

Command c started in state s terminates in state t

Big-step semantics

Concrete syntax:

$$(com,\ initial\text{-}state) \Rightarrow final\text{-}state$$

Intended meaning of $(c,\ s) \Rightarrow t$:

Command c started in state s terminates in state t

"$\Rightarrow$" here not type!

Big-step rules

$$(SKIP,\ s) \Rightarrow\ s$$

Big-step rules

$$(SKIP,\ s) \Rightarrow\ s$$

$$(x ::=\ a,\ s) \Rightarrow\ s(x :=\ aval\ a\ s)$$

Big-step rules

$$(SKIP,\ s) \Rightarrow s$$

$$(x ::= a,\ s) \Rightarrow s(x := aval\ a\ s)$$

$$\frac{(c_1,\ s_1) \Rightarrow s_2 \qquad (c_2,\ s_2) \Rightarrow s_3}{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}$$

Big-step rules

$$\frac{bval\ b\ s \qquad (c_1,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

Big-step rules

$$\frac{bval\ b\ s \qquad (c_1,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

$$\frac{\neg\ bval\ b\ s \qquad (c_2,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}$$

Big-step rules

$$\frac{\neg\ bval\ b\ s}{(WHILE\ b\ DO\ c,\ s) \Rightarrow s}$$

Big-step rules

$$\frac{\neg\ bval\ b\ s}{(WHILE\ b\ DO\ c,\ s) \Rightarrow s}$$

$$\frac{bval\ b\ s_1 \quad (c,\ s_1) \Rightarrow s_2 \quad (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3}{(WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}$$

Examples: derivation trees

$$\frac{\vdots}{(''x'' ::= N\ 5;;\ ''y'' ::= V\ ''x'',\ s) \Rightarrow\ \ ?}$$

Examples: derivation trees

$$\frac{\vdots}{(''x'' ::= N\ 5;;\ ''y'' ::= V\ ''x'',\ s) \Rightarrow\ ?} \qquad \frac{\vdots}{(w,\ s_i) \Rightarrow\ ?}$$

where
$$\begin{array}{rcl} w & = & WHILE\ b\ DO\ c \\ b & = & NotEq\ (V\ ''x'')\ (N\ 2) \\ c & = & ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1) \\ s_i & = & s(''x'' := i) \end{array}$$

Examples: derivation trees

$$\frac{\vdots}{(''x'' ::= N\ 5;;\ ''y'' ::= V\ ''x'',\ s) \Rightarrow\ ?} \qquad \frac{\vdots}{(w,\ s_i) \Rightarrow\ ?}$$

where
$$\begin{array}{rcl} w & = & WHILE\ b\ DO\ c \\ b & = & NotEq\ (V\ ''x'')\ (N\ 2) \\ c & = & ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1) \\ s_i & = & s(''x'' := i) \end{array}$$

$NotEq\ a_1\ a_2 =$
$Not(And\ (Not(Less\ a_1\ a_2))\ (Not(Less\ a_2\ a_1)))$

Logically speaking

$$(c,\ s) \Rightarrow t$$

is just infix syntax for

$$big\_step\ (c,s)\ t$$

Logically speaking

$$(c,\ s) \Rightarrow t$$

is just infix syntax for

$$big\_step\ (c,s)\ t$$

where

$$big\_step :: com \times state \Rightarrow state \Rightarrow bool$$

is an inductively defined predicate.

Big_Step.thy

Semantics

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$?

Rule inversion

What can we deduce from

- $(SKIP, s) \Rightarrow t$? $t = s$

Rule inversion

What can we deduce from

- $(SKIP, s) \Rightarrow t$? $t = s$
- $(x ::= a, s) \Rightarrow t$?

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $\quad t = s$
- $(x ::= a,\ s) \Rightarrow t$? $\quad t = s(x := aval\ a\ s)$

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $\quad t = s$
- $(x ::= a,\ s) \Rightarrow t$? $\quad t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$?

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $\quad t = s$
- $(x ::= a,\ s) \Rightarrow t$? $\quad t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$?
  $\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3$

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $\quad t = s$
- $(x ::= a,\ s) \Rightarrow t$? $\quad t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$?
  $\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3$
- $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$?

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $\quad t = s$
- $(x ::= a,\ s) \Rightarrow t$? $\quad t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$?
  $\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3$
- $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$?
  $bval\ b\ s \wedge (c_1,\ s) \Rightarrow t\ \vee$
  $\neg\ bval\ b\ s \wedge (c_2,\ s) \Rightarrow t$

Rule inversion

What can we deduce from

- $(SKIP, s) \Rightarrow t$? $t = s$
- $(x ::= a, s) \Rightarrow t$? $t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2, s_1) \Rightarrow s_3$?
  $\exists\, s_2.\ (c_1, s_1) \Rightarrow s_2 \wedge (c_2, s_2) \Rightarrow s_3$
- $(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \Rightarrow t$?
  $bval\ b\ s \wedge (c_1, s) \Rightarrow t\ \vee$
  $\neg\ bval\ b\ s \wedge (c_2, s) \Rightarrow t$
- $(w, s) \Rightarrow t$ where $w = WHILE\ b\ DO\ c$?

Rule inversion

What can we deduce from

- $(SKIP,\ s) \Rightarrow t$? $t = s$
- $(x ::= a,\ s) \Rightarrow t$? $t = s(x := aval\ a\ s)$
- $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$?
  $\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3$
- $(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t$?
  $bval\ b\ s \wedge (c_1,\ s) \Rightarrow t\ \vee$
  $\neg\ bval\ b\ s \wedge (c_2,\ s) \Rightarrow t$
- $(w,\ s) \Rightarrow t$ where $w = WHILE\ b\ DO\ c$?
  $\neg\ bval\ b\ s \wedge t = s\ \vee$
  $bval\ b\ s \wedge (\exists\, s'.\ (c,\ s) \Rightarrow s' \wedge (w,\ s') \Rightarrow t)$

Automating rule inversion

Isabelle command **inductive_cases** produces theorems that perform rule inversions automatically.

We reformulate the inverted rules. Example:

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

We reformulate the inverted rules. Example:

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

is logically equivalent to

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3 \qquad \bigwedge s_2.\ [\![(c_1,\ s_1) \Rightarrow s_2;\ (c_2,\ s_2) \Rightarrow s_3]\!] \Longrightarrow P}{P}$$

We reformulate the inverted rules. Example:

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

is logically equivalent to

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3 \qquad \bigwedge s_2.\ [\![(c_1,\ s_1) \Rightarrow s_2;\ (c_2,\ s_2) \Rightarrow s_3]\!] \Longrightarrow P}{P}$$

Replaces assm $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$ by two assms $(c_1,\ s_1) \Rightarrow s_2$ and $(c_2,\ s_2) \Rightarrow s_3$

We reformulate the inverted rules. Example:

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

is logically equivalent to

$$\frac{\begin{array}{c}(c_1;;\ c_2,\ s_1) \Rightarrow s_3 \\ \bigwedge s_2.\ [\![(c_1,\ s_1) \Rightarrow s_2;\ (c_2,\ s_2) \Rightarrow s_3]\!] \Longrightarrow P\end{array}}{P}$$

Replaces assm $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$ by two assms $(c_1,\ s_1) \Rightarrow s_2$ and $(c_2,\ s_2) \Rightarrow s_3$ (with a new fixed s_2).

We reformulate the inverted rules. Example:

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}{\exists\, s_2.\ (c_1,\ s_1) \Rightarrow s_2 \wedge (c_2,\ s_2) \Rightarrow s_3}$$

is logically equivalent to

$$\frac{(c_1;;\ c_2,\ s_1) \Rightarrow s_3 \qquad \bigwedge s_2.\ [\![(c_1,\ s_1) \Rightarrow s_2;\ (c_2,\ s_2) \Rightarrow s_3]\!] \Longrightarrow P}{P}$$

Replaces assm $(c_1;;\ c_2,\ s_1) \Rightarrow s_3$ by two assms $(c_1,\ s_1) \Rightarrow s_2$ and $(c_2,\ s_2) \Rightarrow s_3$ (with a new fixed s_2).
No $\exists$ and $\wedge$!

The general format: *elimination rules*

$$\frac{asm \quad asm_1 \Longrightarrow P \quad \ldots \quad asm_n \Longrightarrow P}{P}$$

The general format: *elimination rules*

$$\frac{asm \quad asm_1 \Longrightarrow P \quad \dots \quad asm_n \Longrightarrow P}{P}$$

(possibly with $\bigwedge \overline{x}$ in front of the $asm_i \Longrightarrow P$)

The general format: *elimination rules*

$$\frac{asm \quad asm_1 \Longrightarrow P \quad \ldots \quad asm_n \Longrightarrow P}{P}$$

(possibly with $\bigwedge \overline{x}$ in front of the $asm_i \Longrightarrow P$)

Reading:

To prove a goal P with assumption asm,
prove all $asm_i \Longrightarrow P$

The general format: *elimination rules*

$$\frac{asm \quad asm_1 \Longrightarrow P \quad \ldots \quad asm_n \Longrightarrow P}{P}$$

(possibly with $\bigwedge \overline{x}$ in front of the $asm_i \Longrightarrow P$)

Reading:

> To prove a goal P with assumption asm,
> prove all $asm_i \Longrightarrow P$

Example:

$$\frac{F \lor G \quad F \Longrightarrow P \quad G \Longrightarrow P}{P}$$

$elim$ attribute

- Theorems with $elim$ attribute are used automatically by $blast$, $fastforce$ and $auto$

elim attribute

- Theorems with *elim* attribute are used automatically by *blast*, *fastforce* and *auto*
- Can also be added locally, eg *(blast elim: ...)*

elim attribute

- Theorems with *elim* attribute are used automatically by *blast*, *fastforce* and *auto*
- Can also be added locally, eg *(blast elim: ...)*
- Variant: *elim!* applies elim-rules eagerly.

Big_Step.thy

Rule inversion

Command equivalence

Two commands have the same input/output behaviour:

Command equivalence

Two commands have the same input/output behaviour:

$$c \sim c' \;\equiv\; (\forall\, s\ t.\ (c,s) \Rightarrow t \longleftrightarrow (c',s) \Rightarrow t)$$

Command equivalence

Two commands have the same input/output behaviour:

$$c \sim c' \;\equiv\; (\forall\, s\; t.\; (c,s) \Rightarrow t \longleftrightarrow (c',s) \Rightarrow t)$$

Example

$$w \sim w'$$

where $\;w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
$\qquad\; w' = \mathit{IF}\ b\ \mathit{THEN}\ c;;\ w\ \mathit{ELSE}\ \mathit{SKIP}$

Equivalence proof

$$(w,\ s) \Rightarrow\ t$$

Equivalence proof

$$(w,\ s) \Rightarrow t$$

$$\longleftrightarrow$$

$$\begin{gathered} bval\ b\ s \wedge (\exists\, s'.\ (c,\ s) \Rightarrow s' \wedge (w,\ s') \Rightarrow t) \\ \vee \\ \neg\ bval\ b\ s \wedge t = s \end{gathered}$$

Equivalence proof

$$(w,\ s) \Rightarrow t$$

$$\longleftrightarrow$$

$$bval\ b\ s \wedge (\exists\, s'.\ (c,\ s) \Rightarrow s' \wedge (w,\ s') \Rightarrow t)$$
$$\vee$$
$$\neg\ bval\ b\ s \wedge t = s$$

$$\longleftrightarrow$$

$$(w',\ s) \Rightarrow t$$

Equivalence proof

$$(w,\ s) \Rightarrow t$$

$$\longleftrightarrow$$

$$\begin{array}{c} bval\ b\ s \wedge (\exists\, s'.\ (c,\ s) \Rightarrow s' \wedge (w,\ s') \Rightarrow t) \\ \vee \\ \neg\ bval\ b\ s \wedge t = s \end{array}$$

$$\longleftrightarrow$$

$$(w',\ s) \Rightarrow t$$

Using the rules and rule inversions for $\Rightarrow$.

`Big_Step.thy`

Command equivalence

Execution is deterministic

Any two executions of the same command in the same start state lead to the same final state:

$$(c,\ s) \Rightarrow t \implies (c,\ s) \Rightarrow t' \implies t = t'$$

Execution is deterministic

Any two executions of the same command in the same start state lead to the same final state:

$$(c,\ s) \Rightarrow t \implies (c,\ s) \Rightarrow t' \implies t = t'$$

Proof by rule induction, for arbitrary t'.

Big_Step.thy

Execution is deterministic

The boon and bane of big steps

We cannot observe intermediate states/steps

The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

(c,s) does not terminate iff $\nexists\, t.\ (c,\ s) \Rightarrow\ t$?

The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

(c,s) does not terminate iff $\nexists\, t.\ (c,\ s) \Rightarrow\ t$?

Needs a formal notion of nontermination to prove it.

The boon and bane of big steps

We cannot observe intermediate states/steps

Example problem:

(c,s) does not terminate iff $\nexists\, t.\ (c,\ s) \Rightarrow\ t$?

Needs a formal notion of nontermination to prove it.
Could be wrong if we have forgotten a $\Rightarrow$ rule.

Big-step semantics cannot directly describe

- nonterminating computations,

Big-step semantics cannot directly describe

- nonterminating computations,
- parallel computations.

Big-step semantics cannot directly describe

- nonterminating computations,
- parallel computations.

We need a finer grained semantics!

1. IMP Commands

2. Big-Step Semantics

3. Small-Step Semantics

Small-step semantics

Concrete syntax:

$$(com, state) \rightarrow (com, state)$$

Small-step semantics

Concrete syntax:

$$(com,state) \to (com,state)$$

Intended meaning of $(c,\ s) \to (c',\ s')$:

Small-step semantics

Concrete syntax:

$$(\mathit{com},\mathit{state}) \to (\mathit{com},\mathit{state})$$

Intended meaning of $(c,\ s) \to (c',\ s')$:

> *The first step in the execution of c in state s leaves a "remainder" command c' to be executed in state s'.*

Small-step semantics

Concrete syntax:

$$(com, state) \to (com, state)$$

Intended meaning of $(c,\ s) \to (c',\ s')$:

> *The first step in the execution of c in state s*
> *leaves a "remainder" command c'*
> *to be executed in state s'.*

Execution as finite or infinite reduction:

$$(c_1, s_1) \to (c_2, s_2) \to (c_3, s_3) \to \ldots$$

Terminology

- A pair (c,s) is called a *configuration*.

Terminology

- A pair (c,s) is called a *configuration*.
- If $cs \to cs'$ we say that cs *reduces* to cs'.

Terminology

- A pair (c,s) is called a *configuration*.
- If $cs \to cs'$ we say that cs *reduces* to cs'.
- A configuration cs is *final* iff $\nexists\, cs'.\ cs \to cs'$

The intention:

$$(SKIP,\ s) \text{ is final}$$

The intention:

$$(SKIP,\ s) \text{ is final}$$

Why?

$SKIP$ is the empty program.

The intention:

$$(SKIP,\ s)\ \text{ is final}$$

Why?

$SKIP$ is the empty program. Nothing more to be done.

Small-step rules

$$(x{::=}a,\ s) \ \rightarrow$$

Small-step rules

$$(x ::= a,\ s) \ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

Small-step rules

$$(x ::= a,\ s) \ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

$$(SKIP;;\ c,\ s) \ \rightarrow$$

Small-step rules

$$(x ::= a,\ s)\ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

$$(SKIP;;\ c,\ s)\ \rightarrow\ (c,\ s)$$

Small-step rules

$$(x ::= a,\ s) \ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

$$(SKIP;;\ c,\ s) \ \rightarrow\ (c,\ s)$$

$$\frac{(c_1, s) \ \rightarrow\ (c_1', s')}{(c_1;; c_2, s) \ \rightarrow}$$

Small-step rules

$$(x ::= a,\ s) \ \rightarrow\ (SKIP,\ s(x := \ aval\ a\ s))$$

$$(SKIP;;\ c,\ s) \ \rightarrow\ (c,\ s)$$

$$\frac{(c_1, s) \ \rightarrow\ (c_1', s')}{(c_1;; c_2, s) \ \rightarrow\ (c_1';; c_2, s')}$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s)\ \rightarrow}$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_1, s)}$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_1, s)}$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_2, s)}$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_1, s)}$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_2, s)}$$

$$(WHILE\ b\ DO\ c,\ s) \ \rightarrow$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_1, s)}$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow\ (c_2, s)}$$

$$(WHILE\ b\ DO\ c,\ s) \ \rightarrow$$
$$(IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)$$

Small-step rules

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow \ (c_1, s)}$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \ \rightarrow \ (c_2, s)}$$

$$(WHILE\ b\ DO\ c,\ s) \ \rightarrow$$
$$(IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)$$

Fact $(SKIP, s)$ is a final configuration.

Small-step examples

$$(''z'' ::= V\ ''x'';;\ ''x'' ::= V\ ''y'';;\ ''y'' ::= V\ ''z'',\ s) \rightarrow$$

$$\ldots$$

where $s = <''x'' := 3,\ ''y'' := 7,\ ''z'' := 5>$.

Small-step examples

$$(''z'' ::= V\ ''x'';; ''x'' ::= V\ ''y'';; ''y'' ::= V\ ''z'',\ s) \rightarrow$$
$$\ldots$$

where $s = <''x'' := 3,\ ''y'' := 7,\ ''z'' := 5>$.

$$(w,\ s_0) \rightarrow \ldots$$

where
$$\begin{aligned} w &= \ WHILE\ b\ DO\ c \\ b &= \ Less\ (V\ ''x'')\ (N\ 1) \\ c &= \ ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1) \\ s_n &= \ <''x'' := n> \end{aligned}$$

`Small_Step.thy`

Semantics

Are big and small-step semantics equivalent?

From $\Rightarrow$ to $\rightarrow^*$

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP,\ t)$

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP, t)$

Proof by rule induction

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP, t)$

Proof by rule induction (of course on $cs \Rightarrow t$)

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP, t)$

Proof by rule induction (of course on $cs \Rightarrow t$)

In two cases a lemma is needed:

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP,\ t)$

Proof by rule induction (of course on $cs \Rightarrow t$)
In two cases a lemma is needed:

Lemma
$(c_1,\ s) \rightarrow* (c_1{}',\ s') \implies (c_1;;\ c_2,\ s) \rightarrow* (c_1{}';;\ c_2,\ s')$

From $\Rightarrow$ to $\rightarrow*$

Theorem $cs \Rightarrow t \implies cs \rightarrow* (SKIP,\ t)$

Proof by rule induction (of course on $cs \Rightarrow t$)
In two cases a lemma is needed:

Lemma
$(c_1,\ s) \rightarrow* (c_1{'},\ s') \implies (c_1;;\ c_2,\ s) \rightarrow* (c_1{'};;\ c_2,\ s')$

Proof by rule induction.

From $\rightarrow*$ to $\Rightarrow$

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP,\ t) \implies cs \Rightarrow t$

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP,\ t) \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow* (SKIP,\ t)$.

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP,\ t) \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow* (SKIP,\ t)$.
In the induction step a lemma is needed:

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP,\ t) \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow* (SKIP,\ t)$.
In the induction step a lemma is needed:

Lemma $cs \rightarrow cs' \implies cs' \Rightarrow t \implies cs \Rightarrow t$

From $\rightarrow*$ to $\Rightarrow$

Theorem $cs \rightarrow* (SKIP, t) \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow* (SKIP, t)$.
In the induction step a lemma is needed:

Lemma $cs \rightarrow cs' \implies cs' \Rightarrow t \implies cs \Rightarrow t$

Proof by rule induction on $cs \rightarrow cs'$.

Equivalence

Corollary $cs \Rightarrow t \;\longleftrightarrow\; cs \rightarrow\!* \;(SKIP,\; t)$

Small_Step.thy

Equivalence of big and small

Can execution stop prematurely?

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
    - $c_1 = SKIP$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
  - $c_1 \neq SKIP$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP,s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c,s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
  - $c_1 \neq SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1,\ s)$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
  - $c_1 \neq SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1,\ s)$ (by IH)

Can execution stop prematurely?

That is, are there any final configs except $(SKIP, s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c, s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
  - $c_1 \neq SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1,\ s)$ (by IH)
    $\Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$

Can execution stop prematurely?

That is, are there any final configs except $(SKIP,s)$?

Lemma $\mathit{final}\ (c,\ s) \Longrightarrow c = SKIP$

We prove the contrapositive

$c \neq SKIP \Longrightarrow \neg\ \mathit{final}(c,s)$

by induction on c.

- Case $c_1;;\ c_2$: by case distinction:
  - $c_1 = SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
  - $c_1 \neq SKIP \Longrightarrow \neg\ \mathit{final}\ (c_1,\ s)$ (by IH)
    $\Longrightarrow \neg\ \mathit{final}\ (c_1;;\ c_2,\ s)$
- Remaining cases: trivial or easy

By rule inversion: $(SKIP,\ s) \to ct \Longrightarrow False$

By rule inversion: $(SKIP,\ s) \to ct \Longrightarrow False$

Together:

Corollary $\mathit{final}\ (c,\ s) = (c = SKIP)$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow\ t) = (\exists\, cs'.\ cs \rightarrow\!*\ cs' \wedge \mathit{final}\ cs')$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow\ t) = (\exists\, cs'.\ cs \rightarrow\!*\ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow\ t)$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow\ t) = (\exists\, cs'.\ cs \rightarrow\!*\ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow\ t)$

$= \ (\exists\, t.\ cs \rightarrow\!*\ (\mathit{SKIP}, t))$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow\ t) = (\exists\, cs'.\ cs \rightarrow\!*\ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow\ t)$

$= \ (\exists\, t.\ cs \rightarrow\!*\ (SKIP,t))$

(by big = small)

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow t) = (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow t)$

$= (\exists\, t.\ cs \rightarrow\!* \ (\mathit{SKIP}, t))$

(by big = small)

$= (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow t) = (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow t)$

$= (\exists\, t.\ cs \rightarrow\!* \ (\mathit{SKIP}, t))$

(by big = small)

$= (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs')$

(by final = SKIP)

Infinite executions

$\Rightarrow$ yields final state iff $\rightarrow$ terminates

Lemma $(\exists\, t.\ cs \Rightarrow t) = (\exists\, cs'.\ cs \rightarrow\!\!*\ cs' \wedge \mathit{final}\ cs')$

Proof: $(\exists\, t.\ cs \Rightarrow t)$

$= (\exists\, t.\ cs \rightarrow\!\!*\ (SKIP,t))$

(by big = small)

$= (\exists\, cs'.\ cs \rightarrow\!\!*\ cs' \wedge \mathit{final}\ cs')$

(by final = SKIP)

Equivalent:

$\Rightarrow$ does not yield final state iff $\rightarrow$ does not terminate

May versus Must

$\rightarrow$ is deterministic:

May versus Must

$\rightarrow$ is deterministic:

Lemma $cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$

May versus Must

$\rightarrow$ is deterministic:

Lemma $cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$
(Proof by rule induction)

May versus Must

$\rightarrow$ is deterministic:

Lemma $cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$
(Proof by rule induction)

Therefore: no difference between

may terminate (there is a terminating $\rightarrow$ path)

must terminate (all $\rightarrow$ paths terminate)

May versus Must

$\rightarrow$ is deterministic:

Lemma $cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$
(Proof by rule induction)

Therefore: no difference between

may terminate (there is a terminating $\rightarrow$ path)

must terminate (all $\rightarrow$ paths terminate)

Therefore: $\Rightarrow$ correctly reflects termination behaviour.

May versus Must

$\rightarrow$ is deterministic:

Lemma $cs \rightarrow cs' \implies cs \rightarrow cs'' \implies cs'' = cs'$
(Proof by rule induction)

Therefore: no difference between

may terminate (there is a terminating $\rightarrow$ path)
must terminate (all $\rightarrow$ paths terminate)

Therefore: $\Rightarrow$ correctly reflects termination behaviour.

With nondeterminism: may have both $cs \Rightarrow t$ and a nonterminating reduction $cs \rightarrow cs' \rightarrow \ldots$

Chapter 8

Hoare Logic

4 Weakest Preconditions

5 Towards Simpler Verification of Programs

6 Example Verifications

4. Weakest Preconditions

5. Towards Simpler Verification of Programs

6. Example Verifications

4 Weakest Preconditions

Introduction

We have proved functional programs correct

We have proved functional programs correct

We have modeled semantics of imperative languages

We have proved functional programs correct

We have modeled semantics of imperative languages

But how do we prove imperative programs correct?

An example program:

$$
\begin{array}{l}
\textit{program exp} \; \{ \\
\quad a := 1 \\
\quad \textit{while } (0 < n) \textit{ do } \{ \\
\qquad a := a + a; \\
\qquad n := n - 1 \\
\quad \} \\
\}
\end{array}
$$

An example program:

$$
\begin{array}{l}
\textit{program exp } \{ \\
\quad a := 1 \\
\quad \textit{while } (0 < n) \textit{ do } \{ \\
\quad\quad a := a + a; \\
\quad\quad n := n - 1 \\
\quad \} \\
\}
\end{array}
$$

At the end of the execution, variable a should contain 2^n,

An example program:

$$
\begin{array}{l}
\textit{program exp } \{ \\
\quad a := 1 \\
\quad \textit{while } (0 < n) \textit{ do } \{ \\
\qquad a := a + a; \\
\qquad n := n - 1 \\
\quad \} \\
\}
\end{array}
$$

At the end of the execution, variable a should contain 2^n, where n is the original value of variable n!

An example program:

$$
\begin{array}{l}
\textit{program exp}\ \{ \\
\quad a := 1 \\
\quad \textit{while}\ (0{<}n)\ \textit{do}\ \{ \\
\qquad a := a + a; \\
\qquad n := n - 1 \\
\quad \} \\
\}
\end{array}
$$

At the end of the execution, variable a should contain 2^n, where n is the original value of variable n!
and $0 \leq n$!

In general: If *precondition* holds for initial state
then, program terminates, and
final state satisfies *postcondition*

In general: If *precondition* holds for initial state
then, program terminates, and
final state satisfies *postcondition*

Formally

In general: If *precondition* holds for initial state
then, program terminates, and
final state satisfies *postcondition*

Formally?

In general: If *precondition* holds for initial state
then, program terminates, and
final state satisfies *postcondition*

Formally?

$$P\ s \Longrightarrow \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t$$

In general: If *precondition* holds for initial state
then, program terminates, and
final state satisfies *postcondition*

Formally?

$$P\ s \Longrightarrow \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t$$

The RHS of this implication is called *weakest precondition*

$$wp\ c\ Q\ s \equiv \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t$$

In general: If *precondition* holds for initial state then, program terminates, and final state satisfies *postcondition*

Formally?

$$P\ s \Longrightarrow \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t$$

The RHS of this implication is called *weakest precondition*

$$wp\ c\ Q\ s \equiv \exists\, t.\ (c,\ s) \Rightarrow\ t \wedge\ Q\ t$$

Weakest condition on state, such that program c will satisfy postcondition Q.

Some obvious facts:

Some obvious facts:

Consequence rule:

$$\llbracket wp\ \ c\ \ P\ \ s;\ \bigwedge s.\ \ P\ \ s \Longrightarrow\ \ Q\ \ s\rrbracket \Longrightarrow\ \ wp\ \ c\ \ Q\ \ s$$

Some obvious facts:

Consequence rule:

$$\llbracket wp\ c\ P\ s; \bigwedge s.\ P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wp\ c\ Q\ s$$

wp of equivalent programs is equal

$$c \sim c' \Longrightarrow wp\ c = wp\ c'$$

Correctness of exp

Correctness of exp?

Correctness of exp?

$$0 \le s\ ''n'' \Longrightarrow wp\ exp\ \left(\lambda s'.\ s'\ ''a'' = 2^{\mathsf{nat}\ (s\ ''n'')}\right)\ s$$

Correctness of exp?

$$0 \le s\ ''n'' \Longrightarrow wp\ exp\ (\lambda s'.\ s'\ ''a'' = 2^{\mathsf{nat}\ (s\ ''n'')})\ s$$

$nat{::}int \Rightarrow nat$ required b/c $(\ \hat{}\ ){::}'a \Rightarrow nat \Rightarrow 'a$ only defined on nat.

Correctness of exp?

$$0 \leq s \; ''n'' \Longrightarrow wp \; exp \; (\lambda s'. \; s' \; ''a'' = 2^{\mathsf{nat}\;(\mathsf{s}\;''\mathsf{n}'')}) \; s$$

$nat{::}int \Rightarrow nat$ required b/c $(\hat{\ })::'a \Rightarrow nat \Rightarrow 'a$ only defined on nat.

In general: $P \; s \Longrightarrow wp \; c \; Q \; s$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s =$$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s = Q\ s$$

How to prove correctness of programs?

$P\ s \Longrightarrow wp\ c\ Q\ s$

$wp\ SKIP\ Q\ s = Q\ s$

$wp\ (x ::= a)\ Q\ s =$

How to prove correctness of programs?

$P\ s \Longrightarrow wp\ c\ Q\ s$

$wp\ SKIP\ Q\ s = Q\ s$
$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s = Q\ s$$
$$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$$
$$wp\ (c_1;;\ c_2)\ Q\ s =$$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s = Q\ s$$
$$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$$
$$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s = Q\ s$$
$$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$$
$$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$$
$$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$$
$$=$$

How to prove correctness of programs?

$$P\ s \Longrightarrow wp\ c\ Q\ s$$

$$wp\ SKIP\ Q\ s = Q\ s$$
$$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$$
$$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$$
$$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$$
$$= \textsf{if}\ bval\ b\ s\ \textsf{then}\ wp\ c_1\ Q\ s\ \textsf{else}\ wp\ c_2\ Q\ s$$

How to prove correctness of programs?

$P\ s \Longrightarrow wp\ c\ Q\ s$

$wp\ SKIP\ Q\ s = Q\ s$
$wp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$
$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$
$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$
$\quad = \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wp\ c_1\ Q\ s\ \textsf{\textit{else}}\ wp\ c_2\ Q\ s$

Reasoning along syntax of program!

That was easy!

That was easy! But what about While?

That was easy! But what about While?

$$wp\ (WHILE\ b\ DO\ c)\ Q\ s$$
$$=$$

That was easy! But what about $While$?

$wp\ (WHILE\ b\ DO\ c)\ Q\ s$
$=$*if* $bval\ b\ s$ *then* $wp\ c\ (wp\ (WHILE\ b\ DO\ c)\ Q)\ s$ *else*
$Q\ s$

That was easy! But what about $While$?

$wp\ (WHILE\ b\ DO\ c)\ Q\ s$
$=$*if* $bval\ b\ s$ *then* $wp\ c\ (wp\ (WHILE\ b\ DO\ c)\ Q)\ s$ *else*
$Q\ s$

Unfolding will continue forever!

That was easy! But what about $While$?

$wp\ (WHILE\ b\ DO\ c)\ Q\ s$
$=$*if* $bval\ b\ s$ *then* $wp\ c\ (wp\ (WHILE\ b\ DO\ c)\ Q)\ s$ *else* $Q\ s$

Unfolding will continue forever!

Obviously, need some inductive argument!

That was easy! But what about $While$?

$wp\ (WHILE\ b\ DO\ c)\ Q\ s$
$=$*if* $bval\ b\ s$ **then** $wp\ c\ (wp\ (WHILE\ b\ DO\ c)\ Q)\ s$ *else* $Q\ s$

Unfolding will continue forever!

Obviously, need some inductive argument!

But, let's get less ambitious (for first)

Weakest liberal precondition

$$wlp\ c\ Q\ s \equiv \forall\, t.\ (c,\ s) \Rightarrow\ t \longrightarrow\ Q\ t$$

Weakest liberal precondition

$$wlp\ c\ Q\ s \equiv \forall\, t.\ (c,\ s) \Rightarrow\ t \longrightarrow\ Q\ t$$

If c terminates on s, then new state satisfies Q

Weakest liberal precondition

$$wlp\ c\ Q\ s \equiv \forall\, t.\ (c,\ s) \Rightarrow\ t \longrightarrow\ Q\ t$$

If c terminates on s, then new state satisfies Q

Cannot reason about termination. This is called *partial correctness*.

Some obvious facts:

$$c \sim c' \Longrightarrow wlp\ c = wlp\ c'$$

$$\llbracket wlp\ c\ P\ s;\ \bigwedge s.\ P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wlp\ c\ Q\ s$$

Some obvious facts:

$$c \sim c' \Longrightarrow wlp\ c = wlp\ c'$$
$$\llbracket wlp\ c\ P\ s;\ \bigwedge s.\ P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wlp\ c\ Q\ s$$

Relation between wp and wlp

$$wp\ c\ Q\ s \Longrightarrow wlp\ c\ Q\ s$$
$$wlp\ c\ Q\ s \wedge (c,\ s) \Rightarrow t \Longrightarrow wp\ c\ Q\ s$$

Some obvious facts:

$$c \sim c' \Longrightarrow wlp\ c = wlp\ c'$$
$$[\![wlp\ c\ P\ s;\ \bigwedge s.\ P\ s \Longrightarrow Q\ s]\!] \Longrightarrow wlp\ c\ Q\ s$$

Relation between wp and wlp

$$wp\ c\ Q\ s \Longrightarrow wlp\ c\ Q\ s$$
$$wlp\ c\ Q\ s \wedge (c,\ s) \Rightarrow t \Longrightarrow wp\ c\ Q\ s$$

Unfold rules still hold:

$$wlp\ SKIP\ Q\ s = Q\ s$$
$$wlp\ (x ::= a)\ Q\ s = Q\ (s(x := aval\ a\ s))$$
$$wlp\ (c_1;;\ c_2)\ Q\ s = wlp\ c_1\ (wlp\ c_2\ Q)\ s$$
$$wlp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s =$$
$$(\textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wlp\ c_1\ Q\ s\ \textsf{\textit{else}}\ wlp\ c_2\ Q\ s)$$

$wlp\ (WHILE\ b\ DO\ c)\ Q\ s =$
$(\textit{if}\ bval\ b\ s\ \textsf{then}\ wlp\ c\ (wlp\ (WHILE\ b\ DO\ c)\ Q)\ s\ \textsf{else}$
$Q\ s)$

$wlp\ (WHILE\ b\ DO\ c)\ Q\ s =$
$(\textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wlp\ c\ (wlp\ (WHILE\ b\ DO\ c)\ Q)\ s\ \textsf{\textit{else}}$
$Q\ s)$

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wp\ c\ I\ s\ \textsf{\textit{else}}\ Q\ s$

$wlp\ (WHILE\ b\ DO\ c)\ Q\ s =$
$(\textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wlp\ c\ (wlp\ (WHILE\ b\ DO\ c)\ Q)\ s\ \textsf{\textit{else}}$
$Q\ s)$

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wp\ c\ I\ s\ \textsf{\textit{else}}\ Q\ s$

and I holds for start state.

$wlp\ (WHILE\ b\ DO\ c)\ Q\ s =$
$(\textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wlp\ c\ (wlp\ (WHILE\ b\ DO\ c)\ Q)\ s\ \textsf{\textit{else}}$
$Q\ s)$

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wp\ c\ I\ s\ \textsf{\textit{else}}\ Q\ s$

and I holds for start state.

Intuition: I holds initially, is preserved by iteration, and implies Q at end of loop.

$wlp\ (WHILE\ b\ DO\ c)\ Q\ s =$
$(\textit{if}\ bval\ b\ s\ \textit{then}\ wlp\ c\ (wlp\ (WHILE\ b\ DO\ c)\ Q)\ s\ \textit{else}$
$Q\ s)$

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow \textit{if}\ bval\ b\ s\ \textit{then}\ wp\ c\ I\ s\ \textit{else}\ Q\ s$

and I holds for start state.

Intuition: I holds initially, is preserved by iteration, and implies Q at end of loop. I is called *loop invariant*

While-rule for partial correctness

$$\llbracket I\ s_0;\ \bigwedge s.\ I\ s \Longrightarrow \textsf{\textit{if}}\ bval\ b\ s\ \textsf{\textit{then}}\ wlp\ c\ I\ s\ \textsf{\textit{else}}\ Q\ s\rrbracket$$
$$\Longrightarrow wlp\ (WHILE\ b\ DO\ c)\ Q\ s_0$$

`Wp_Demo.thy`

Weakest Precondition

Now we can start proving programs ...

Now we can start proving programs ...

$$P\ s \Longrightarrow wlp\ c\ Q\ s$$

Now we can start proving programs ...

$P\ s \Longrightarrow wlp\ c\ Q\ s$

If $c = WHILE\ \_\ DO\ \_$, provide invariant and apply while rule

Now we can start proving programs ...

$$P\ s \Longrightarrow wlp\ c\ Q\ s$$

If $c = WHILE\ \_\ DO\ \_$, provide invariant and apply while rule

Otherwise, use unfold rules.

Now we can start proving programs ...

$$P\ s \Longrightarrow wlp\ c\ Q\ s$$

If $c =\ WHILE\ \_\ DO\ \_$, provide invariant and apply while rule

Otherwise, use unfold rules.

Iterate, until all wlps gone!

wlp_if_eq and wlp_whileI' produce $if{-}then{-}else$

wlp_if_eq and wlp_whileI' produce $if{-}then{-}else$ which we have to split.

wlp_if_eq and wlp_whileI' produce $if\!-\!then\!-\!else$ which we have to split.

Combine rule with splitting!

Wp_Demo.thy

Proving Partial Correctness

But how about termination?

But how about termination?

An (ordering) relation $<$ is *well-founded*, iff every non-empty set has a minimal element.

But how about termination?

An (ordering) relation $<$ is *well-founded*, iff every non-empty set has a minimal element.

Equivalently: No infinite sequence with $x_1 > x_2 > \ldots$

But how about termination?

An (ordering) relation $<$ is *well-founded*, iff every non-empty set has a minimal element.

Equivalently: No infinite sequence with $x_1 > x_2 > \ldots$

Well-foundedness implies induction principle

But how about termination?

An (ordering) relation $<$ is *well-founded*, iff every non-empty set has a minimal element.

Equivalently: No infinite sequence with $x_1 > x_2 > \ldots$

Well-foundedness implies induction principle

$$\frac{wf\ r \qquad \bigwedge x.\ \dfrac{\forall\, y.\ (y,\ x) \in r \longrightarrow P\ y}{P\ x}}{P\ a}$$

Wellfounded_Demo.thy

For while loop: Find wf relation $<$ such that state decreases in each iteration

For while loop: Find wf relation $<$ such that state decreases in each iteration

$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ *then* $wp\ c\ (\lambda s'.\ I\ s' \wedge s' < s)\ s$ *else* $Q\ s$

For while loop: Find wf relation $<$ such that state decreases in each iteration

$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ **then** $wp\ c\ (\lambda s'.\ I\ s' \wedge s' < s)\ s$ *else* $Q\ s$

Then use wf-induction to prove:

$[\![wf\ R;\ I\ s_0;$
$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ **then** $wp\ c\ (\lambda s'.\ I\ s' \wedge (s',\ s) \in R)\ s$ *else* $Q\ s]\!]$
$\Longrightarrow wp\ (WHILE\ b\ DO\ c)\ Q\ s_0$

Or, equivalently

assumes WF: $wf\ R$
assumes $INIT$: $I\ s_0$
assumes $STEP$: $\bigwedge s.\ [\![\ I\ s;\ bval\ b\ s\ ]\!]$
$\Longrightarrow wp\ c\ (\lambda s'.\ I\ s' \wedge (s',s) \in R)\ s$
assumes $FINAL$: $\bigwedge s.\ [\![\ I\ s;\ \neg bval\ b\ s\ ]\!] \Longrightarrow Q\ s$
shows $wp\ (WHILE\ b\ DO\ c)\ Q\ s_0$

Or, equivalently

assumes WF: $wf\ R$
assumes $INIT$: $I\ s_0$
assumes $STEP$: $\bigwedge s.\ \llbracket\ I\ s;\ bval\ b\ s\ \rrbracket$
$\Longrightarrow wp\ c\ (\lambda s'.\ I\ s' \wedge (s',s){\in}R)\ s$
assumes $FINAL$: $\bigwedge s.\ \llbracket\ I\ s;\ \neg bval\ b\ s\ \rrbracket \Longrightarrow Q\ s$
shows $wp\ (WHILE\ b\ DO\ c)\ Q\ s_0$

Now we can prove total correctness ...

Wp_Demo.thy

Total Correctness

Let's make our VCG more usable

Let's make our VCG more usable

Add standard arithmetic operators to IMP

Let's make our VCG more usable

Add standard arithmetic operators to IMP

Add nice syntax for programs

Let's make our VCG more usable

Add standard arithmetic operators to IMP

Add nice syntax for programs

Make VCs more readable

Let's make our VCG more usable

Add standard arithmetic operators to IMP

Add nice syntax for programs

Make VCs more readable

Simplify specification of pre/postcondition, and invariants

Standard operators

We add generic syntax for any unary/binary operator

Standard operators

We add generic syntax for any unary/binary operator

$$Unop :: (int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp$$

Standard operators

We add generic syntax for any unary/binary operator

$$Unop :: (int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp$$
$$Binop :: (int \Rightarrow int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp \Rightarrow aexp$$

Standard operators

We add generic syntax for any unary/binary operator

$$Unop :: (int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp$$
$$Binop :: (int \Rightarrow int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp \Rightarrow aexp$$
$$Cmpop :: (int \Rightarrow int \Rightarrow bool) \Rightarrow aexp \Rightarrow aexp \Rightarrow bexp$$

Standard operators

We add generic syntax for any unary/binary operator

$$\begin{aligned}
&Unop :: (int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp \\
&Binop :: (int \Rightarrow int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp \Rightarrow aexp \\
&Cmpop :: (int \Rightarrow int \Rightarrow bool) \Rightarrow aexp \Rightarrow aexp \Rightarrow bexp \\
&BBinop :: (bool \Rightarrow bool \Rightarrow bool) \Rightarrow bexp \Rightarrow bexp \Rightarrow bexp
\end{aligned}$$

Standard operators

We add generic syntax for any unary/binary operator

$$Unop :: (int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp$$
$$Binop :: (int \Rightarrow int \Rightarrow int) \Rightarrow aexp \Rightarrow aexp \Rightarrow aexp$$
$$Cmpop :: (int \Rightarrow int \Rightarrow bool) \Rightarrow aexp \Rightarrow aexp \Rightarrow bexp$$
$$BBinop :: (bool \Rightarrow bool \Rightarrow bool) \Rightarrow bexp \Rightarrow bexp \Rightarrow bexp$$

For example:

$Cmpop\ (\leq)\ (Binop\ (+)\ (Unop\ uminus\ (V\ ''x''))\ (N$
$42))\ (N\ 50)$

IMP2/Introduction.thy

Adding more Operators

C-like syntax

Operators

C-like syntax

Operators

Arith: $+,-,*,/$ with usual binding

C-like syntax

Operators

Arith: $+, -, *, /$ with usual binding

Boolean: $\neg, \wedge, \vee$ and $=, \neq, \leq, <, >, \geq$

C-like syntax

Operators

Arith: $+,-,*,/$ with usual binding

Boolean: $\neg,\wedge,\vee$ and $=,\neq,\leq,<,>,\geq$

Commands

C-like syntax

Operators

Arith: $+, -, *, /$ with usual binding

Boolean: $\neg, \wedge, \vee$ and $=, \neq, \leq, <, >, \geq$

Commands

$skip$, $v = \ aexp$, $\{c\}$, $c_1; \ c_2$

C-like syntax

Operators

Arith: $+, -, *, /$ with usual binding

Boolean: $\neg, \wedge, \vee$ and $=, \neq, \leq, <, >, \geq$

Commands

$skip$, $v = aexp$, $\{c\}$, $c_1;\ c_2$

$if\ bexp\ then\ c_1\ [else\ c_2]$

C-like syntax

Operators

Arith: $+,-,*,/$ with usual binding

Boolean: $\neg,\wedge,\vee$ and $=,\neq,\leq,<,>,\geq$

Commands

$skip$, $v = aexp$, $\{c\}$, $c_1;\ c_2$

$if\ bexp\ then\ c_1\ [else\ c_2]$ else part is optional

C-like syntax

Operators

Arith: $+, -, *, /$ with usual binding

Boolean: $\neg, \wedge, \vee$ and $=, \neq, \leq, <, >, \geq$

Commands

$skip$, $v = aexp$, $\{c\}$, $c_1;\ c_2$

$if\ bexp\ then\ c_1\ [else\ c_2]$ else part is optional

$while\ (bexp)\ c$

`IMP2/Introduction.thy`

Program Syntax

More Readable VCs

Idea: Replace $s\ ''x''$ by (Isabelle) variable x.

More Readable VCs

Idea: Replace $s\ ''x''$ by (Isabelle) variable x.

Similar: $s_0\ ''x''$ by x_0.

More Readable VCs

Idea: Replace $s\ ''x''$ by (Isabelle) variable x.
Similar: $s_0\ ''x''$ by x_0.

If subgoal can still be proved for arbitrary (Isabelle) variable x, it can, in particular, be proved for $s\ ''x''$.

$$(\bigwedge x.\ P\ x) \Longrightarrow P\ (s\ ''x'')$$

IMP2/Introduction.thy

More Readable VCs

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c{\leq}n_0 \wedge a = c{*}c$ for
$s\ ''c'' \leq s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c{\leq}n_0 \wedge a = c{*}c$ for
$s\ ''c'' \leq s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

Which variables to interpret?

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c{\le}n_0 \wedge a = c{*}c$ for
$s\ ''c'' \le s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

Which variables to interpret? over which states?

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c \leq n_0 \wedge a = c * c$ for
$s\ ''c'' \leq s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

Which variables to interpret? over which states?

All variables that occur in the program!

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c{\le}n_0 \wedge a = c{*}c$ for
$s\ ''c'' \le s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

Which variables to interpret? over which states?

All variables that occur in the program!

Precondition: x interpreted as $s\ ''x''$

More Readable Annotations

Can we do similar trick for pre/postconditions and invariants?

E.g. write $c{\leq}n_0 \wedge a = c{*}c$ for
$s\ ''c'' \leq s_0\ ''n'' \wedge s\ ''a'' = s\ ''c'' * s\ ''c''$

Which variables to interpret? over which states?

All variables that occur in the program!

Precondition: x interpreted as $s\ ''x''$

Postcondition/Invariant: x as $s\ ''x''$, x_0 as $s_0\ ''x''$

IMP2/Introduction.thy

More Readable Annotations

4 Weakest Preconditions

5 Towards Simpler Verification of Programs

6 Example Verifications

Common Loop Patterns

We've seen a few loop's already:

Common Loop Patterns

We've seen a few loop's already:

$a{=}1;\ c{=}0;\ while\ (c{<}n)\ \{a{=}2{*}a;\ c{=}c{+}1\}$
Compute operation by iterating weaker operation

Common Loop Patterns

We've seen a few loop's already:

$a{=}1;\ c{=}0;\ while\ (c{<}n)\ \{a{=}2{*}a;\ c{=}c{+}1\}$
Compute operation by iterating weaker operation
e.g. $2^n = 2 * \ldots * 2$

Common Loop Patterns

We've seen a few loop's already:

$a{=}1;\ c{=}0;\ while\ (c{<}n)\ \{a{=}2{*}a;\ c{=}c{+}1\}$
Compute operation by iterating weaker operation
e.g. $2^n = 2 * \ldots * 2$

Use accumulator a and increment counter (count-up)

Common Loop Patterns

We've seen a few loop's already:

$a{=}1;\ c{=}0;\ while\ (c{<}n)\ \{a{=}2{*}a;\ c{=}c{+}1\}$
Compute operation by iterating weaker operation
e.g. $2^n = 2 * \ldots * 2$

Use accumulator a and increment counter (count-up)

Or decrement counter (e.g. n) (count down)

Common Loop Patterns

We've seen a few loop's already:

a=1; c=0; *while* (c<n) {a=2∗a; c=c+1}

Compute operation by iterating weaker operation

e.g. $2^n = 2 * \ldots * 2$

Use accumulator a and increment counter (count-up)

Or decrement counter (e.g. n) (count down)

Invariant: $a = 2\,\hat{}\,c \wedge \ldots$ (accumulator = f(iterations))

Common Loop Patterns

We've seen a few loop's already:

$a{=}1;\ c{=}0;\ \mathit{while}\ (c{<}n)\ \{a{=}2{*}a;\ c{=}c{+}1\}$

Compute operation by iterating weaker operation

e.g. $2^n = 2 * \ldots * 2$

Use accumulator a and increment counter (count-up)

Or decrement counter (e.g. n) (count down)

Invariant: $a = 2\,\hat{}\,c \wedge \ldots$ (accumulator = f(iterations))

Applications: $*$ by $+$, exp, Fibonacci, factorial, ...

IMP2/Examples.thy

Count-up, Count-Down

Approximate Naively

Invert monotonic function, by naively trying all values:

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ while\ (r{*}r{\le}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ while\ (r{*}r{\leq}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ \mathit{while}\ (r{*}r{\le}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ \mathit{while}\ (r{*}r{\leq}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ while\ (r{*}r{\leq}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Idea: Iterate until we overshoot by one. Then decrement.

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ while\ (r{*}r{\le}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Idea: Iterate until we overshoot by one. Then decrement.

Invariant:

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ while\ (r{*}r{\leq}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Idea: Iterate until we overshoot by one. Then decrement.

Invariant: ?

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ \mathit{while}\ (r{*}r{\leq}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Idea: Iterate until we overshoot by one. Then decrement.

Invariant: ? $(r{-}1)^2 \leq n \wedge \ldots$ ($r{-}1$ below or equal result)

Approximate Naively

Invert monotonic function, by naively trying all values:

$r{=}1;\ \mathit{while}\ (r{*}r{\le}n)\ \{r{=}r{+}1\};\ r{=}r{-}1$

What does this compute?square root, rounded down!

Idea: Iterate until we overshoot by one. Then decrement.

Invariant: ? $(r{-}1)^2 \le n \wedge \ldots$ ($r{-}1$ below or equal result)

Applications: sqrt, log, ...

`IMP2/Examples.thy`

Approximate from Below

Bisection

We can compute sqrt more efficiently.

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Invariant

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Invariant?

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Invariant? $l^2 \leq n < h^2 \wedge \ldots$ (range contains solution)

Bisection

We can compute sqrt more efficiently.

```
l=0; h=n+1;
while (l+1 < h)
  m = (l + h) / 2;
  if m*m ≤ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Invariant? $l^2 \leq n < h^2 \wedge \ldots$ (range contains solution)

This program is actually tricky to get right!

IMP2/Examples.thy

Bisection

6 Example Verifications

Loop Patterns

Euclid's Algorithm

Advanced Verification

Arrays

Data Refinement

Euclid Intro

Compute gcd of positive numbers a, b

Euclid Intro

Compute gcd of positive numbers a, b

Reminder: Divides: $(b\ dvd\ a) = (\exists\, k.\ a = b * k)$
Greatest Common Divisor: $gcd{::}int \Rightarrow int \Rightarrow int$ such that
$gcd\ a\ b\ dvd\ a$ and $gcd\ a\ b\ dvd\ b$ and
$[\![a \neq 0;\ b \neq 0;\ c\ dvd\ a;\ c\ dvd\ b]\!] \Longrightarrow c \leq gcd\ a\ b$

Euclid Variants

By subtraction. Using $gcd\ (m - n)\ n = gcd\ m\ n$

Euclid Variants

By subtraction. Using $gcd\ (m - n)\ n = gcd\ m\ n$

By modulo. Using: $gcd\ x\ y = gcd\ y\ (x\ mod\ y)$

`IMP2/Examples.thy`

Euclid

6 Example Verifications

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$ and only $a{,}i$ changed.

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$ and only $a{,}i$ changed.

Invariant: $a{=}2\,\hat{}\,i \wedge 0{\leq}i \wedge i{\leq}n$ and only $a{,}i$ changed.

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$ and only $a{,}i$ changed.

Invariant: $a{=}2\,\hat{}\,i \wedge 0{\leq}i \wedge i{\leq}n$ and only $a{,}i$ changed.

Only $a{,}i$ changed: $\forall\, x.\ x \notin \{''a'',\ ''i''\} \longrightarrow s\ x = s'\ x$

Modified Variables

Program: $a{=}1;\ i{=}0;\ while\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$ and only $a{,}i$ changed.

Invariant: $a{=}2\,\hat{}\,i \wedge 0{\leq}i \wedge i{\leq}n$ and only $a{,}i$ changed.

Only $a{,}i$ changed: $\forall\, x.\ x \notin \{''a'',\ ''i''\} \longrightarrow s\ x = s'\ x$

$modifies\ vars\ s_1\ s_2 = (\forall\, x.\ x \notin vars \longrightarrow s_1\ x = s_2\ x)$

Modified Variables

Program: $a{=}1;\ i{=}0;\ \mathit{while}\ (i{<}n)\ \{\ a{=}a{*}2;\ i{=}i{+}1\ \}$

Pre: $n{\geq}0$ Post: $a{=}2\,\hat{}\,n_0$ and only a,i changed.

Invariant: $a{=}2\,\hat{}\,i \wedge 0{\leq}i \wedge i{\leq}n$ and only a,i changed.

Only a,i changed: $\forall\, x.\ x \notin \{''a'',\ ''i''\} \longrightarrow s\ x = s'\ x$

$\mathit{modifies}\ \mathit{vars}\ s_1\ s_2 = (\forall\, x.\ x \notin \mathit{vars} \longrightarrow s_1\ x = s_2\ x)$

Program modifies at most variables it assigns to

$(c,\ s) \Rightarrow t \Longrightarrow \mathit{modifies}\ (\mathit{lhsv}\ c)\ t\ s$

Modified Variables

We can strengthen correctness statement (automatically)

$$wp\ c\ Q\ s \Longrightarrow wp\ c\ (\lambda s'.\ Q\ s' \wedge modifies\ (lhsv\ c)\ s'\ s)\ s$$

Modified Variables

We can strengthen correctness statement (automatically)

$$wp\ c\ Q\ s \Longrightarrow wp\ c\ (\lambda s'.\ Q\ s' \wedge modifies\ (lhsv\ c)\ s'\ s)\ s$$

For while-rule, we get

lemma wp_whileI_modset:
  fixes c
  defines $[simp]$: $modset \equiv lhsv\ c$
  assumes WF: $wf\ R$
  assumes $INIT$: $I\ \mathfrak{s}_0$
  assumes $STEP$: $\bigwedge\mathfrak{s}.\ [\![\ modifies\ modset\ \mathfrak{s}\ \mathfrak{s}_0;\ I\ \mathfrak{s};\ bval\ b\ \mathfrak{s}\ ]\!]$
$\Longrightarrow wp\ c\ (\lambda\mathfrak{s}'.\ I\ \mathfrak{s}' \wedge (\mathfrak{s}',\mathfrak{s}) \in R)\ \mathfrak{s}$
  assumes $FINAL$: $\bigwedge\mathfrak{s}.\ [\![\ modifies\ modset\ \mathfrak{s}\ \mathfrak{s}_0;\ I\ \mathfrak{s};\ \neg bval\ b\ \mathfrak{s}\ ]\!]$
$\Longrightarrow Q\ \mathfrak{s}$
  shows $wp\ (WHILE\ b\ DO\ c)\ Q\ \mathfrak{s}_0$

Modified Variables

The VCG will automatically rewrite with rule

$$\llbracket \mathit{modifies}\ vs\ s\ s';\ x \notin vs \rrbracket \Longrightarrow s\ x = s'\ x$$

Modified Variables

The VCG will automatically rewrite with rule

$$\llbracket modifies\ vs\ s\ s';\ x \notin vs \rrbracket \Longrightarrow s\ x = s'\ x$$

program_spec computes lhs-variables:

$$HT\_mods\ mods\ P\ c\ Q \equiv HT\ P\ c\ Q \wedge mods = lhsv\ c$$

`IMP2/Examples.thy`

Euclid – show modified sets

Modular Proofs

Consider program

$$
\begin{array}{l}
a{=}1; \\
\mathit{while}\ (m{>}0)\ \{ \\
\quad n{=}a;\ a = 1; \\
\quad \mathit{while}\ (n{>}0)\ \{ \\
\quad\quad a{=}2*a;\ n{=}n{-}1 \\
\quad \}; \\
\quad m{=}m{-}1 \\
\}
\end{array}
$$

What does this compute

Modular Proofs

Consider program

$$
\begin{array}{l}
a{=}1; \\
\mathit{while}\ (m{>}0)\ \{ \\
\quad n{=}a;\ a = 1; \\
\quad \mathit{while}\ (n{>}0)\ \{ \\
\quad\quad a{=}2*a;\ n{=}n{-}1 \\
\quad \}; \\
\quad m{=}m{-}1 \\
\}
\end{array}
$$

What does this compute?

Modular Proofs

Consider program

$$
\begin{array}{l}
a{=}1; \\
\textit{while}\ (m{>}0)\ \{ \\
\quad n{=}a;\ a = 1; \\
\quad \textit{while}\ (n{>}0)\ \{ \\
\quad\quad a{=}2*a;\ n{=}n{-}1 \\
\quad \}; \\
\quad m{=}m{-}1 \\
\}
\end{array}
$$

What does this compute?

Power-tower function: $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (m times)

Modular Proofs

Inner loop invariant: Would like to refer to n right before loop!

Modular Proofs

Inner loop invariant: Would like to refer to n right before loop!

In our simple VCG, we can't!

Modular Proofs

Inner loop invariant: Would like to refer to n right before loop!

In our simple VCG, we can't!

Still, we already have verified inner loop!

Modular Proofs

Inner loop invariant: Would like to refer to n right before loop!

In our simple VCG, we can't!

Still, we already have verified inner loop!

Idea: Split and verify separately!

Modular Proofs

```
a=1;
while (m>0) {
  n=a;
  inline exp_count_down;
  m=m-1
}
```

Modular Proofs

$a{=}1;$
$while\ (m{>}0)\ \{$
$\quad n{=}a;$
$\quad inline\ exp\_count\_down;$
$\quad m{=}m{-}1$
$\}$

Reuse existing proof of exp-count-down program!

Modular Proofs

Re-using proofs:

Modular Proofs

Re-using proofs:

$$[\![HT\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ Q\ s\ s' \Longrightarrow Q'\ s']\!] \Longrightarrow wp\ c\ Q'\ s$$

Modular Proofs

Re-using proofs:

$$\llbracket HT\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ Q\ s\ s' \Longrightarrow Q'\ s' \rrbracket \Longrightarrow wp\ c\ Q'\ s$$

with modified sets:

$$\llbracket HT\_mods\ mods\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ \llbracket modifies\ mods\ s'\ s;\ Q\ s\ s' \rrbracket \Longrightarrow Q'\ s' \rrbracket \Longrightarrow wp\ c\ Q'\ s$$

Modular Proofs

Re-using proofs:

$$\llbracket HT\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ Q\ s\ s' \Longrightarrow Q'\ s' \rrbracket \Longrightarrow wp\ c\ Q'\ s$$

with modified sets:

$$\llbracket HT\_mods\ mods\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ \llbracket modifies\ mods\ s'\ s; \\ Q\ s\ s' \rrbracket \Longrightarrow Q'\ s' \rrbracket \\ \Longrightarrow wp\ c\ Q'\ s$$

VCG will automatically use this rule.

Modular Proofs

Re-using proofs:

$$[\![HT\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ Q\ s\ s' \Longrightarrow Q'\ s']\!] \Longrightarrow wp\ c\ Q'\ s$$

with modified sets:

$$[\![HT\_mods\ mods\ P\ c\ Q;\ P\ s;\ \bigwedge s'.\ [\![modifies\ mods\ s'\ s;\ Q\ s\ s']\!] \Longrightarrow Q'\ s']\!] \Longrightarrow wp\ c\ Q'\ s$$

VCG will automatically use this rule.
If inlined program has been proved with **program_spec**

`IMP2/Examples.thy`

Power-Tower

6 Example Verifications

Arrays

Every variable is of type $int \Rightarrow int$.

Arrays

Every variable is of type $int \Rightarrow int$.

Arithmetic Expressions:

$Vidx :: char\ list \Rightarrow aexp \Rightarrow aexp$

$aval\ (Vidx\ x\ i)\ s = s\ x\ (aval\ i\ s)$

Arrays

Every variable is of type $int \Rightarrow int$.

Arithmetic Expressions:

$Vidx :: char\ list \Rightarrow aexp \Rightarrow aexp$

$aval\ (Vidx\ x\ i)\ s = s\ x\ (aval\ i\ s)$

Commands:

$AssignIdx :: char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$

$(x[i] ::= a,\ s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$

Arrays

Every variable is of type $int \Rightarrow int$.

Arithmetic Expressions:

$Vidx :: char\ list \Rightarrow aexp \Rightarrow aexp$

$aval\ (Vidx\ x\ i)\ s = s\ x\ (aval\ i\ s)$

Commands:

$AssignIdx :: char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$

$(x[i] ::= a,\ s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$

$ArrayCpy :: char\ list \Rightarrow char\ list \Rightarrow com$

$(x[] ::= y,\ s) \Rightarrow s(x := s\ y)$

Arrays

Every variable is of type $int \Rightarrow int$.

Arithmetic Expressions:

$Vidx :: char\ list \Rightarrow aexp \Rightarrow aexp$

$aval\ (Vidx\ x\ i)\ s = s\ x\ (aval\ i\ s)$

Commands:

$AssignIdx :: char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$

$(x[i] ::= a,\ s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$

$ArrayCpy :: char\ list \Rightarrow char\ list \Rightarrow com$

$(x[] ::= y,\ s) \Rightarrow s(x := s\ y)$

$ArrayClear :: char\ list \Rightarrow com$

$(CLEAR\ x[],\ s) \Rightarrow s(x := \lambda\_.\ 0)$

Arrays

By default, we use index 0.

Arrays

By default, we use index 0.

Abbreviations:
$V\ x = \ Vidx\ x\ (N\ 0)$

Arrays

By default, we use index 0.

Abbreviations:

$V\ x = \ Vidx\ x\ (N\ 0)$

$Assign\ x\ a = \ AssignIdx\ x\ (N\ 0)\ \ a$

Arrays

By default, we use index 0.

Abbreviations:

$V\ x = \ Vidx\ x\ (N\ 0)$

$Assign\ x\ a = \ AssignIdx\ x\ (N\ 0)\ \ a$

Arrays

By default, we use index 0.

Abbreviations:
$V\ x = \ Vidx\ x\ (N\ 0)$

$Assign\ x\ a = \ AssignIdx\ x\ (N\ 0)\ \ a$

VCG: Guess type from variable usage

Arrays

By default, we use index 0.

Abbreviations:

$V\ x = \ Vidx\ x\ (N\ 0)$

$Assign\ x\ a = \ AssignIdx\ x\ (N\ 0)\ \ a$

VCG: Guess type from variable usage
Only with index 0: Bind $VAR\ (s\ ''x''\ 0)\ (\lambda x.\ \ldots)$

Arrays

By default, we use index 0.

Abbreviations:
$V\ x = Vidx\ x\ (N\ 0)$

$Assign\ x\ a = AssignIdx\ x\ (N\ 0)\ a$

VCG: Guess type from variable usage
Only with index 0: Bind $VAR\ (s\ ''x''\ 0)\ (\lambda x.\ \ldots)$

Otherwise: Bind $VAR\ (s\ ''x'')\ (\lambda x.\ \ldots)$

Arrays

By default, we use index 0.

Abbreviations:

$V\ x = Vidx\ x\ (N\,0)$

$Assign\ x\ a = AssignIdx\ x\ (N\,0)\ a$

VCG: Guess type from variable usage

Only with index 0: Bind $VAR\ (s\ ''x''\,0)\ (\lambda x.\ \dots)$

Otherwise: Bind $VAR\ (s\ ''x'')\ (\lambda x.\ \dots)$

IMP2/Examples.thy

Array-Sum

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$ means?

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i{\in}\{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

$\forall\, i{\in}\{l..{<}h\}.\ \forall\, j{\in}\{l..{<}h\}.\ i{\le}j \longrightarrow a\ i \le a\ j$

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

$\forall\, i \in \{l..{<}h\}.\ \forall\, j \in \{l..{<}h\}.\ i \le j \longrightarrow a\ i \le a\ j$ means?

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

$\forall\, i \in \{l..{<}h\}.\ \forall\, j \in \{l..{<}h\}.\ i \leq j \longrightarrow a\ i \leq a\ j$ means?
Elements l to $<h$ are sorted

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i{\in}\{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

$\forall\, i{\in}\{l..{<}h\}.\ \forall\, j{\in}\{l..{<}h\}.\ i{\le}j \longrightarrow a\ i \le a\ j$ means?
Elements l to $<h$ are sorted

Reasoning about Arrays

Usually, use function $int \Rightarrow int$ directly.

Set interval notation:
$\{l..h\}$, $\{l..{<}h\}$, $\{l{<}..h\}$, $\{l{<}..{<}h\}$

Examples:
$\forall\, i \in \{0..{<}42\}.\ a\ i > 0$ means?
Elements 0 to 41 are positive

$\forall\, i \in \{l..{<}h\}.\ \forall\, j \in \{l..{<}h\}.\ i \leq j \longrightarrow a\ i \leq a\ j$ means?
Elements l to $<h$ are sorted

Theory $IMP2/IMP2\_Aux\_Lemmas$ provides useful lemmas and definitions

`IMP2/Examples.thy`

Sortedness Check

Binary Search Algorithm

Find element in sorted array. In time $O(\log n)$.

Binary Search Algorithm

Find element in sorted array. In time $O(\log n)$.

Idea: Halve interval in each step.

Binary Search Algorithm

Find element in sorted array. In time $O(\log n)$.

Idea: Halve interval in each step.

This algorithm is tricky to implement correctly!

Binary Search Algorithm

Find element in sorted array. In time $O(\log n)$.

Idea: Halve interval in each step.

This algorithm is tricky to implement correctly!

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky ...

— Donald Knuth

Binary Search Algorithm

Find element in sorted array. In time $O(\log n)$.
Idea: Halve interval in each step.

This algorithm is tricky to implement correctly!

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky ...

— Donald Knuth

Only 5 out of 20 surveyed textbooks had correct implementations

— Richard E. Pattis, 1988

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
↑								↑
l								h

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
$\uparrow$				$\uparrow$				$\uparrow$
l				m				h

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					$\uparrow$			$\uparrow$
					l			h

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					$\uparrow$ l	$\uparrow$ m		$\uparrow$ h

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					$\uparrow$	$\uparrow$		
					l	h		

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					$\uparrow$	$\uparrow$		
					m l	h		

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	**13**	17	19	
					$\uparrow$ l, h			

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					$\uparrow$			
					l, h			

$x = 13$

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Returns smallest i with $x \le a[i]$

Notes on Binary Search

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Notes on Binary Search

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Note: Our language has arbitrary large integers.

Notes on Binary Search

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Note: Our language has arbitrary large integers.

Otherwise, $m = (l + h)/2$ may overflow!

Notes on Binary Search

```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Note: Our language has arbitrary large integers.

Otherwise, $m = (l + h)/2$ may overflow!

Bug in Java Standard Library for > 9 years!

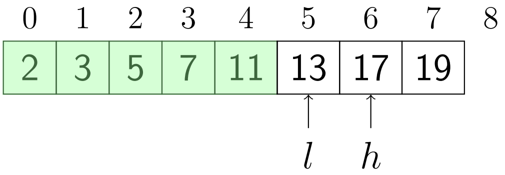
Proving Binary Search

0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					l	h		

$x = 13$

Invariant:

Proving Binary Search



$x = 13$

Invariant:

- $i < l \implies a[i] < x$ (strictly smaller than x)

Proving Binary Search

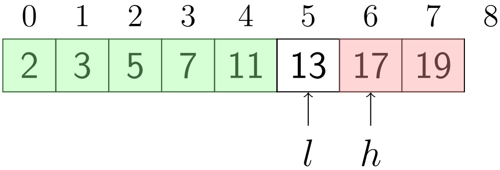
0	1	2	3	4	5	6	7	8
2	3	5	7	11	13	17	19	
					l	h		

$x = 13$

Invariant:

- $i < l \implies a[i] < x$ (strictly smaller than x)
- $i \geq h \implies x \leq a[i]$ (greater or equal to x)

Proving Binary Search



$x = 13$

Invariant:

- $i < l \implies a[i] < x$ (strictly smaller than x)
- $i \geq h \implies x \leq a[i]$ (greater or equal to x)
- and the usual bounds

IMP2/Examples.thy

Binary Search

Insertion Sort

```
j = l + 1;
while (j<h) {
   key = a[j];
   i = j-1;
   while (i>=l && a[i]>key) {
      a[i+1] = a[i];
      i=i-1
   };
   a[i+1] = key
   j=j+1
}
```

Idea: Build sorted array from start.

In each iteration, move next element to its position

Specifying Sorting Algorithms

Precondition: $l \leq h$

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted $ran\_sorted\ a\ l\ h$

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted $ran\_sorted\ a\ l\ h$
- Array contains same elements

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted $ran\_sorted\ a\ l\ h$
- Array contains same elements
  $mset\_ran\ a\ \{l..{<}h\} = mset\_ran\ a_0\ \{l..{<}h\}$

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted $ran\_sorted\ a\ l\ h$
- Array contains same elements
  $mset\_ran\ a\ \{l..<h\} = mset\_ran\ a_0\ \{l..<h\}$

Specifying Sorting Algorithms

Precondition: $l \leq h$

Postcondition:

- Array is sorted $ran\_sorted\ a\ l\ h$
- Array contains same elements
  $mset\_ran\ a\ \{l..<h\} = mset\_ran\ a_0\ \{l..<h\}$

where

$ran\_sorted\ a\ l\ h \equiv \forall\, i \in \{l..<h\}.\ \forall\, j \in \{l..<h\}.\ i \leq j \longrightarrow a\ i \leq a\ j$

$mset\_ran\ a\ r = (\sum i \in r.\ \{\#a\ i\#\})$

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

Multisets in Isabelle

imports $HOL\!-\!Library.Multiset$

$'a\ multiset$: Finite multiset

Multisets in Isabelle

imports $HOL\!-\!Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

$add\_mset\ x\ m$ — add element (cf. $insert$ on sets)

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

$add\_mset\ x\ m$ — add element (cf. $insert$ on sets)

$m_1 + m_2$ — union of multisets

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

$add\_mset\ x\ m$ — add element (cf. $insert$ on sets)

$m_1 + m_2$ — union of multisets

$a \in\#\ m$ — membership query

Multisets in Isabelle

imports $HOL{-}Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

$add\_mset\ x\ m$ — add element (cf. $insert$ on sets)

$m_1 + m_2$ — union of multisets

$a \in\#\ m$ — membership query

$\{\#a,\ b,\ c,\ c\#\}$ — Syntax for add_mset and $\{\#\}$

Multisets in Isabelle

imports $HOL\!-\!Library.Multiset$

$'a\ multiset$: Finite multiset

Some functions and syntax:

$\{\#\}$ — empty multiset

$add\_mset\ x\ m$ — add element (cf. $insert$ on sets)

$m_1 + m_2$ — union of multisets

$a \in\#\ m$ — membership query

$\{\#a,\ b,\ c,\ c\#\}$ — Syntax for add_mset and $\{\#\}$

$$mset\_ran\ a\ r = (\sum i{\in}r.\ \{\#a\ i\#\})$$

Multiset of elements at indexes in finite set r

Proving Insertion Sort

Separate proof for inner loop!

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop:

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$
ensures $ran\_sorted\ a\ l\ (j+1)$

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$

ensures $ran\_sorted\ a\ l\ (j+1)$ and

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$
ensures $ran\_sorted\ a\ l\ (j+1)$ and
ensures $mset\_ran\ a\ \{l..j\} = mset\_ran\ a_0\ \{l..j\}$

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$

ensures $ran\_sorted\ a\ l\ (j + 1)$ and

ensures $mset\_ran\ a\ \{l..j\} = mset\_ran\ a_0\ \{l..j\}$

Invariant of outer loop:

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$

ensures $ran\_sorted\ a\ l\ (j+1)$ and

ensures $mset\_ran\ a\ \{l..j\} = mset\_ran\ a_0\ \{l..j\}$

Invariant of outer loop:

$ran\_sorted\ a\ l\ j$

Proving Insertion Sort

Separate proof for inner loop!

```
j = l + 1;
while (j<h) {
  inline inner_loop;
  j=j+1
}
```

Specification of inner loop: ?

assumes $ran\_sorted\ a\ l\ j$

ensures $ran\_sorted\ a\ l\ (j+1)$ and

ensures $mset\_ran\ a\ \{l..j\} = mset\_ran\ a_0\ \{l..j\}$

Invariant of outer loop:

$ran\_sorted\ a\ l\ j$

$\wedge\ mset\_ran\ a\ \{l..{<}h\} = mset\_ran\ a_0\ \{l..{<}h\}$

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
   a[i+1] = a[i];
   i=i-1
};
a[i+1] = key
```

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
   a[i+1] = a[i];
   i=i-1
};
a[i+1] = key
```

Intuition:

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
  a[i+1] = a[i];
  i=i-1
};
a[i+1] = key
```

Intuition: ?

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
   a[i+1] = a[i];
   i=i-1
};
a[i+1] = key
```

Intuition: ?

$a[j]$ is moved backwards

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
   a[i+1] = a[i];
   i=i-1
};
a[i+1] = key
```

Intuition: ?

$a[j]$ is moved backwards until

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
  a[i+1] = a[i];
  i=i-1
};
a[i+1] = key
```

Intuition: ?
$a[j]$ is moved backwards until
previous element is $\leq a[j]$

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
  a[i+1] = a[i];
  i=i-1
};
a[i+1] = key
```

Intuition: ?

$a[j]$ is moved backwards until previous element is $\leq a[j]$ or

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
   a[i+1] = a[i];
   i=i-1
};
a[i+1] = key
```

Intuition: ?
$a[j]$ is moved backwards until previous element is $\leq a[j]$ or begin of array is reached

Insort: Inner Loop

```
key = a[j];
i = j-1;
while (i>=l && a[i]>key) {
  a[i+1] = a[i];
  i=i-1
};
a[i+1] = key
```

Intuition: ?

$a[j]$ is moved backwards until
previous element is $\leq a[j]$ or
begin of array is reached

Short: Move $a[j]$ backwards over greater elements.

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Let's specify this intuition!

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Let's specify this intuition!

It implies sortedness and mset-preservation

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Let's specify this intuition!

It implies sortedness and mset-preservation

But is closer to what algorithm does

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Let's specify this intuition!

It implies sortedness and mset-preservation

But is closer to what algorithm does

Invariants easier to find!

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$
ensures $i \in \{l - (1::'a)..<j\}$

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$
ensures $i \in \{l - (1::'a)..<j\}$
ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$
ensures $i \in \{l - (1::'a)..<j\}$
ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + (1::'b)) = key$ and

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$

ensures $i \in \{l - (1{::}'a)..{<}j\}$

ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + (1{::}'b)) = key$ and
$\forall\, k{\in}\{i + (2{::}'a)..j\}.\ a\ k = a_0\ (k - (1{::}'a))$

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$

ensures $i \in \{l - (1::'a)..<j\}$

ensures $\forall\, k\in\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + (1::'b)) = key$ and
$\forall\, k\in\{i + (2::'a)..j\}.\ a\ k = a_0\ (k - (1::'a))$

ensures $l \leq i \longrightarrow a\ i \leq key$ and

Insort: Inner Loop

Move $a[j]$ backwards over greater elements.

assumes $l < j$, $let\ key = a_0\ j$

ensures $i \in \{l - (1{::}'a)..{<}j\}$

ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + (1{::}'b)) = key$ and
$\forall\, k{\in}\{i + (2{::}'a)..j\}.\ a\ k = a_0\ (k - (1{::}'a))$

ensures $l \le i \longrightarrow a\ i \le key$ and
$\forall\, k{\in}\{i + (2{::}'a)..j\}.\ key < a\ k$

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	17	19	11
l						i	j

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	17	11	19
l					i		j

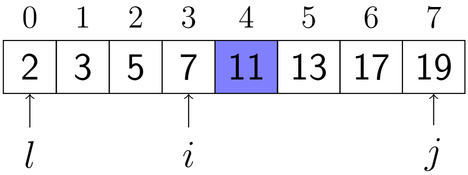
Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
$\uparrow$				$\uparrow$			$\uparrow$
l				i			j

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19
$\uparrow$ l			$\uparrow$ i				$\uparrow$ j

Insort: Finding Invariant



Consider intermediate situation

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
l				i			j

Consider intermediate situation

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
$\uparrow$				$\uparrow$			$\uparrow$
l				i			j

Consider intermediate situation

- indexes $\leq i$ unchanged: $\forall\, k \in \{l..i\}.\ a\ k = a_0\ k$

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
$\uparrow$				$\uparrow$			$\uparrow$
l				i			j

Consider intermediate situation

- indexes $\leq i$ unchanged: $\forall\, k \in \{l..i\}.\ a\ k = a_0\ k$
- indexes $\geq i{+}2$ correctly shifted
  $\forall\, k \in \{i + (2{::}'a)..j\}.\ a\ k = a_0\ (k - (1{::}'a))$

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
$\uparrow$				$\uparrow$			$\uparrow$
l				i			j

Consider intermediate situation

- indexes $\leq i$ unchanged: $\forall\, k \in \{l..i\}.\ a\ k = a_0\ k$
- indexes $\geq i+2$ correctly shifted
  $\forall\, k \in \{i + (2::'a)..j\}.\ a\ k = a_0\ (k - (1::'a))$
- and elements greater than key
  $\forall\, k \in \{i + (2::'a)..j\}.\ key < a\ k$

Insort: Finding Invariant

0	1	2	3	4	5	6	7
2	3	5	7	13	11	17	19
$\uparrow$				$\uparrow$			$\uparrow$
l				i			j

Consider intermediate situation

- indexes $\leq i$ unchanged: $\forall\, k \in \{l..i\}.\ a\ k = a_0\ k$
- indexes $\geq i+2$ correctly shifted
  $\forall\, k \in \{i + (2::'a)..j\}.\ a\ k = a_0\ (k - (1::'a))$
- and elements greater than key
  $\forall\, k \in \{i + (2::'a)..j\}.\ key < a\ k$
- $+$ the usual bounds: $l - (1::'a) \leq i \wedge i < j$

IMP2/Examples.thy

Insertion Sort

Summary so Far

Understand what program does!

Summary so Far

Understand what program does!

Split program into handy parts

Summary so Far

Understand what program does!

Split program into handy parts

Specify what parts do (independently of users)

Summary so Far

Understand what program does!

Split program into handy parts

Specify what parts do (independently of users)

Prove that this implies expectations of users

Summary so Far

Understand what program does!

Split program into handy parts

Specify what parts do (independently of users)

Prove that this implies expectations of users

Prove parts separately and assemble to bigger parts

Abstract View

Model $int \Rightarrow int$ not always appropriate

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\ [l..<h]$ as $int\ list$

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\ [l..{<}h]$ as $int\ list$

Idea: Do proof at level of understanding first

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\ [l..<h]$ as $int\ list$

Idea: Do proof at level of understanding first

then show that implementation is correct!

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\,[l..{<}h]$ as $int\ list$

Idea: Do proof at level of understanding first

then show that implementation is correct!

Instead of one proof, get two

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\ [l..<h]$ as $int\ list$

Idea: Do proof at level of understanding first

then show that implementation is correct!

Instead of one proof, get two ???

Abstract View

Model $int \Rightarrow int$ not always appropriate

E.g., list: Understand $a\ [l..{<}h]$ as $int\ list$

Idea: Do proof at level of understanding first

then show that implementation is correct!

Instead of one complex proof, get two simple proofs !

IMP2/Examples.thy

Filter, Merge, dedup