Concrete Semantics
with Isabelle/HOL

Peter Lammich (slides adopted from Tobias Nipkow)

Fakultät für Informatik
Technische Universität München

2019-1-7

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:

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

Commands

Abstract syntax:

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

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:

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

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:

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

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

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

$$\frac{\begin{array}{c} bval\ b\ s_1 \\ (c,\ s_1) \Rightarrow s_2 \qquad (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3 \end{array}}{(WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}$$

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

Examples: derivation trees

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

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

$$\mathit{NotEq}\ a_1\ a_2 =$$
$$\mathit{Not}(\mathit{And}\ (\mathit{Not}(\mathit{Less}\ a_1\ a_2))\ (\mathit{Not}(\mathit{Less}\ a_2\ a_1)))$$

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

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

Rule inversion

What can we deduce from

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

Rule inversion

What can we deduce from

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

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

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

Rule inversion

What can we deduce from

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

Rule inversion

What can we deduce from

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

Rule inversion

What can we deduce from

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

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:

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

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*

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

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*

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

(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*

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

(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:

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

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 $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$

$w' = \mathit{IF}\ b\ \mathit{THEN}\ c;;\ w\ \mathit{ELSE}\ \mathit{SKIP}$

Equivalence proof

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

Equivalence proof

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

$$\longleftrightarrow$$

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

$$\vee$$

$$\neg\ bval\ b\ s \wedge t = s$$

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

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

$$\longleftrightarrow$$

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

$$\longleftrightarrow$$

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

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:

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

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'.*

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 $cs \rightarrow cs'$ we say that cs *reduces* to cs'.

Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`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:

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

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

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

(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:
$$\begin{aligned} & (\exists\, t.\ cs \Rightarrow t) \\ = \ & (\exists\, t.\ cs \rightarrow\!* \ (\mathit{SKIP}, t)) \\ & \quad \text{(by big = small)} \\ = \ & (\exists\, cs'.\ cs \rightarrow\!* \ cs' \wedge \mathit{final}\ cs') \end{aligned}$$

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

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

(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

7 Advanced Verification

4 Weakest Preconditions

5 Towards Simpler Verification of Programs

6 Example Verifications

7 Advanced Verification

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:

```
$program\ exp\ \{$
  $a := 1$
  $while\ (0{<}n)\ do\ \{$
    $a := a + a;$
    $n := n - 1$
  $\}$
$\}$
```

An example program:

```
$program\ exp$ {
  $a := 1$
  $while\ (0<n)\ do$ {
    $a := a + a;$
    $n := n - 1$
  }
}
```

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

An example program:

```
$program\ exp$ {
  $a := 1$
  $while\ (0 < n)\ do$ {
    $a := a + a;$
    $n := n - 1$
  }
}
```

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

An example program:

```
$program\ exp$ {
  $a := 1$
  $while\ (0<n)\ do$ {
    $a := a + a;$
    $n := n - 1$
  }
}
```

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?

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

Correctness of exp?

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

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

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.

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?

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

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

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

$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$

$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$

$\quad = \textsf{\textit{if}}\ bval\ b\ s\ \textsf{then}\ wp\ c_1\ Q\ s\ \textsf{else}\ wp\ c_2\ Q\ s$

How to prove correctness of programs?

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

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

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

$wp\ (c_1;;\ c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$

$wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$
$\quad = \textsf{if}\ bval\ b\ s\ \textsf{then}\ wp\ c_1\ Q\ s\ \textsf{else}\ wp\ c_2\ Q\ s$

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:

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

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:

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

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

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

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ *then* $wp\ c\ I\ s$ *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.

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

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ *then* $wp\ c\ I\ s$ *else* $Q\ s$

and I holds for start state.

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

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

Let's try to find predicate I, such that

$\bigwedge s.\ I\ s \Longrightarrow$ *if* $bval\ b\ s$ *then* $wp\ c\ I\ s$ *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 $c = \mathit{WHILE}\ \_\ \mathit{DO}\ \_$, 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 $if\text{—}then\text{—}else$ 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: $\mathit{wf}\ R$
assumes INIT: $I\ s_0$
assumes STEP: $\bigwedge s.\ [\![\ I\ s;\ \mathit{bval}\ b\ s\ ]\!]$
$\Longrightarrow \mathit{wp}\ c\ (\lambda s'.\ I\ s' \wedge (s',s){\in}R)\ s$
assumes FINAL: $\bigwedge s.\ [\![\ I\ s;\ \neg \mathit{bval}\ b\ s\ ]\!] \Longrightarrow Q\ s$
shows $\mathit{wp}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ Q\ s_0$

Or, equivalently

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

Now we can prove total correctness ...

`Wp_Demo.thy`

Total Correctness

lemma $ASSUME\_\Theta\_alt$:
$ASSUME\_\Theta\ \pi\ f_0\ s_0\ R\ \Theta = (\forall (f,(P,c,Q)) \in \Theta.\ HT'\ \pi\ (\lambda s.\ (f\ s,\ f_0\ s_0) \in R \wedge\ P\ s)\ c\ Q)$

unfolding $ASSUME\_\Theta\_def\ HT'set\_r\_def$ **..**

4 Weakest Preconditions

5 Towards Simpler Verification of Programs

6 Example Verifications

7 Advanced Verification

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 $c{\le}n_0 \land a = c{*}c$ for
$s\ ''c'' \le s_0\ ''n'' \land s\ ''a'' = s\ ''c'' * s\ ''c''$

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 $c {\le} n_0 \land a = c{*}c$ for
$s\ ''c'' \le s_0\ ''n'' \land s\ ''a'' = s\ ''c'' * s\ ''c''$

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

7 Advanced Verification

6 Example Verifications
Loop Patterns
Euclid's Algorithm

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:

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

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)

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

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))

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:

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

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?

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!

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.

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: ?

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:

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

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)

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 $\leq$ 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 $\leq$ 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 $\leq$ n then l=m else h=m
;
r=l
```

Idea: Half range in each step

Invariant? $l^2 \leq n < h^2 \land \ldots$ (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 $\leq$ 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

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

4 Weakest Preconditions

5 Towards Simpler Verification of Programs

6 Example Verifications

7 Advanced Verification

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

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

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

Program modifies at most variables it assigns to

$\pi\!:\ (c,\ s) \Rightarrow\ t \Longrightarrow\ modifies\ (lhsv\ \pi\ \ c)\ t\ s$

Modified Variables

We can strengthen correctness statement (automatically)

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

Modified Variables

We can strengthen correctness statement (automatically)

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

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:

$HT\_mods\ \pi\ mods\ P\ c\ Q \equiv HT\ \pi\ P\ c\ (\lambda s_0\ s.\ modifies\ mods\ s\ s_0 \wedge Q\ s_0\ s)$

IMP2/Examples.thy

Euclid – show modified sets

Modular Proofs

Consider program

```
$a$=1;
$while$ ($m$>0) {
  $n$=$a$; $a$ = 1;
  $while$ ($n$>0) {
    $a$=2*$a$; $n$=$n$−1
  };
  $m$=$m$−1
}
```

What does this compute

Modular Proofs

Consider program

```
$a$=1;
while ($m$>0) {
  $n$=$a$; $a$ = 1;
  while ($n$>0) {
    $a$=2*$a$; $n$=$n$−1
  };
  $m$=$m$−1
}
```

What does this compute?

Modular Proofs

Consider program

```
$a$=1;
while ($m$>0) {
  $n$=$a$; $a$ = 1;
  while ($n$>0) {
    $a$=2∗$a$; $n$=$n$−1
  };
  $m$=$m$−1
}
```

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) {
  $n$=$a$;
  $inline$ $exp\_count\_down$;
  $m$=$m$−1
}
```

Reuse existing proof of exp-count-down program!

Modular Proofs

Re-using proofs:

Modular Proofs

Re-using proofs:

$$\llbracket HT\,\pi\ P\ c\ Q;\ \bigwedge s.\ P'\ s \Longrightarrow P\ s;\ \bigwedge s_0\ s.\ \llbracket P\ s_0;\ P'\ s_0;\ Q\ s_0\ s\rrbracket \Longrightarrow Q'\ s_0\ s\rrbracket$$
$$\Longrightarrow HT\,\pi\ P'\ c\ Q'$$

Modular Proofs

Re-using proofs:

$$\llbracket HT\ \pi\ \ P\ c\ Q;\ \bigwedge s.\ P'\ s \Longrightarrow P\ s;\ \bigwedge s_0\ s.\ \llbracket P\ s_0;\ P'\ s_0;\ Q\ s_0\ s\rrbracket \Longrightarrow Q'\ s_0\ s\rrbracket \Longrightarrow HT\ \pi\ P'\ c\ Q'$$

with modified sets:

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

Modular Proofs

Re-using proofs:

$$\llbracket HT\ \pi\ \ P\ c\ \ Q;\ \bigwedge s.\ \ P'\ s \Longrightarrow P\ s;\ \bigwedge s_0\ \ s.\ \llbracket P\ s_0;\ \ P'\ s_0;\ \ Q\ s_0\ \ s\rrbracket \Longrightarrow Q'\ s_0\ \ s\rrbracket \Longrightarrow HT\ \pi\ \ P'\ c\ \ Q'$$

with modified sets:

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

VCG will automatically use this rule.

Modular Proofs

Re-using proofs:

$$\llbracket HT\ \pi\ \ P\ \ c\ \ Q;\ \bigwedge s.\ \ P'\ s \Longrightarrow P\ \ s;\ \bigwedge s_0\ \ s.\ \llbracket P\ \ s_0;\ \ P'\ \ s_0;\ \ Q\ \ s_0\ \ s\rrbracket \Longrightarrow Q'\ s_0\ \ s\rrbracket \Longrightarrow HT\ \pi\ \ P'\ \ c\ \ Q'$$

with modified sets:

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

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

`IMP2/Examples.thy`

Power-Tower

7 Advanced Verification
Arrays
Data Refinement
Local Variables
Recursion

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:

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

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

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:

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

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

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

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

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:

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

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

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

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

$ArrayClear :: char\ list \Rightarrow com$

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

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 $VAR\ (s\ ''x''\ 0)\ (\lambda x.\ \dots)$

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 $\ VAR\ (s\ ''x'')\ (\lambda x.\ \dots)$

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

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

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

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

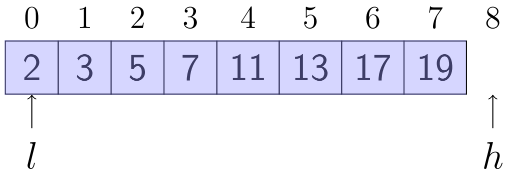
Binary Search Algorithm

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
}
```

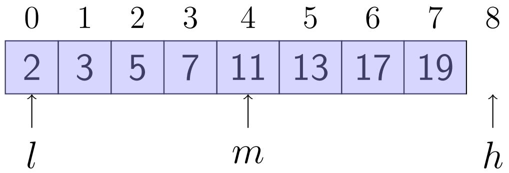
Binary Search Algorithm



$x = 13$

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

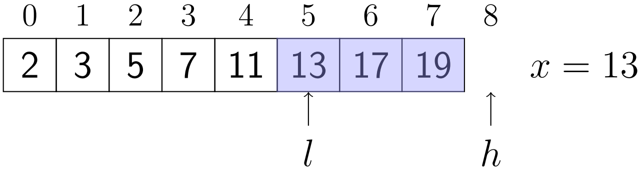
Binary Search Algorithm



$x = 13$

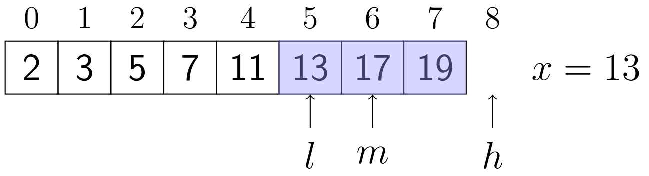
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm



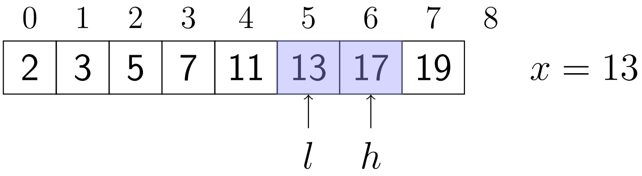
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm



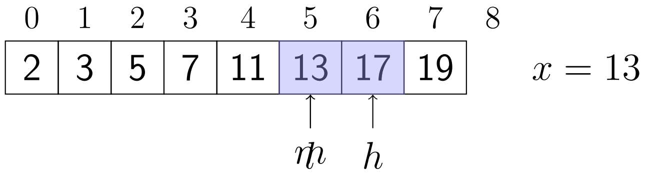
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm



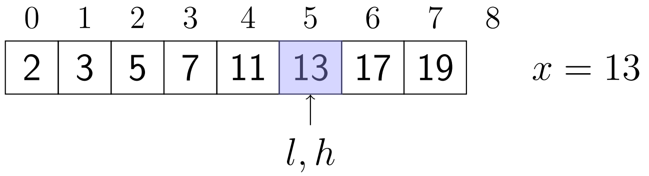
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm



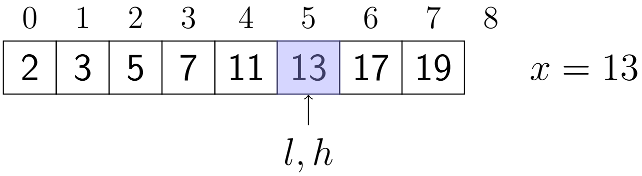
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Binary Search Algorithm



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

Binary Search Algorithm



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

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

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

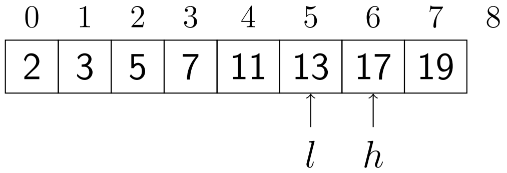
```
while (l < h) {
    m = (l + h) / 2;
    if (a[m] < x) l = m + 1
    else h = m
}
```

Note: Our language has arbitrary large integers.

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

Bug in Java Standard Library for > 9 years!

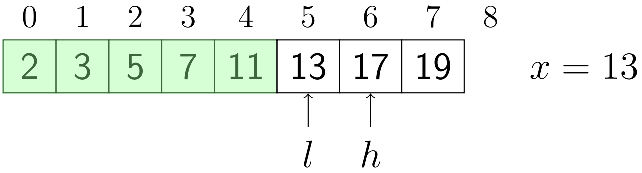
Proving Binary Search



$x = 13$

Invariant:

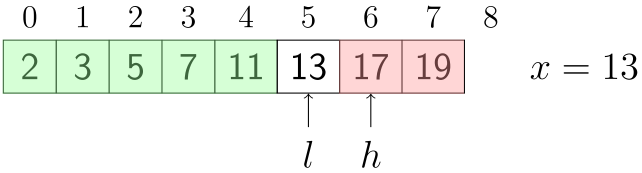
Proving Binary Search



Invariant:

- $i{<}l \Longrightarrow a[i] < x$ (strictly smaller than x)

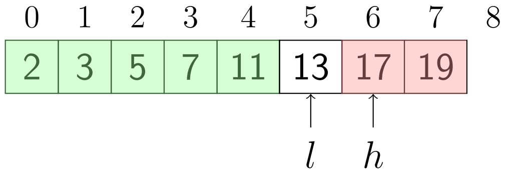
Proving Binary Search



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: $l{\le}h$

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: $l \le h$

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

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

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

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 $HOL\text{--}Library.Multiset$

$'a\ multiset$: 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 $i \in \{l - 1..{<}j\}$

Insort: Inner Loop

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

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

ensures $i \in \{l - 1..{<}j\}$

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 $i \in \{l - 1..{<}j\}$

ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + 1) = key$ and

Insort: Inner Loop

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

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

ensures $i \in \{l - 1..{<}j\}$

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

Insort: Inner Loop

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

assumes $l < j$, $let\ key = a_0\ j$
ensures $i \in \{l - 1..<j\}$
ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + 1) = key$ and
$\forall\, k{\in}\{i + 2..j\}.\ a\ k = a_0\ (k - 1)$
ensures $l \le i \longrightarrow a\ i \le key$ and

Insort: Inner Loop

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

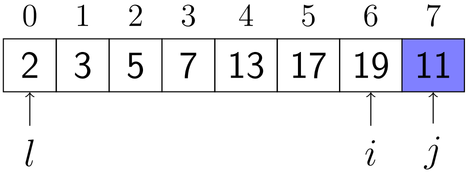
assumes $l < j$, $let\ key = a_0\ j$

ensures $i \in \{l - 1..<j\}$

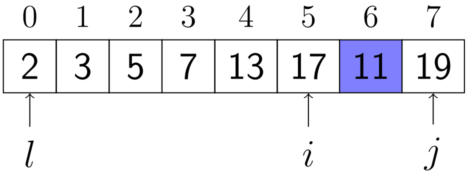
ensures $\forall\, k{\in}\{l..i\}.\ a\ k = a_0\ k$ and
$a\ (i + 1) = key$ and
$\forall\, k{\in}\{i + 2..j\}.\ a\ k = a_0\ (k - 1)$

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

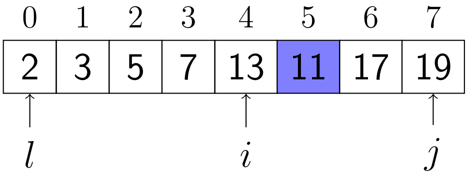
Insort: Finding Invariant



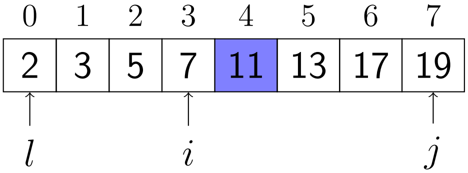
Insort: Finding Invariant



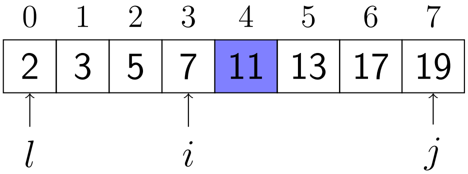
Insort: Finding Invariant



Insort: Finding Invariant

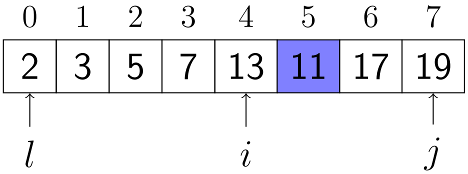


Insort: Finding Invariant



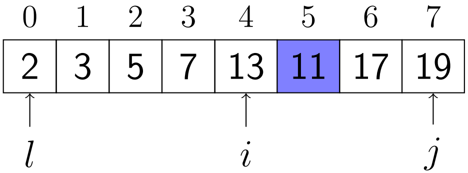
Consider intermediate situation

Insort: Finding Invariant



Consider intermediate situation

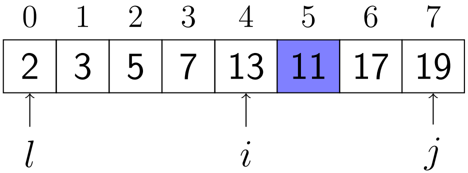
Insort: Finding Invariant



Consider intermediate situation

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

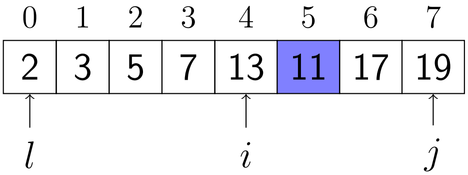
Insort: Finding Invariant



Consider intermediate situation

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

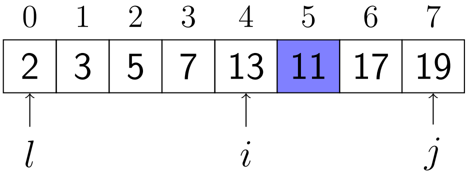
Insort: Finding Invariant



Consider intermediate situation

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

Insort: Finding Invariant



Consider intermediate situation

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

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

7 Advanced Verification
Arrays
Data Refinement
Local Variables
Recursion

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

7 Advanced Verification
Arrays
Data Refinement
Local Variables
Recursion

Local Variables

Introduce local variables

Local Variables

Introduce local variables

Why?

Local Variables

Introduce local variables

Why? Better modularity.

Local Variables

Introduce local variables

Why? Better modularity.

Don't worry about name-clashes with subroutine's auxiliary variables

Local Variables

Partition variable names into local and global names

Local Variables

Partition variable names into local and global names

is_global — Variable name starts with ”G”

Local Variables

Partition variable names into local and global names

is_global — Variable name starts with "G"

```
fun $is\_global :: vname \Rightarrow bool$ where
  $is\_global\ [] \longleftrightarrow True$
| $is\_global\ (CHR\ ''G''\#\_) \longleftrightarrow True$
| $is\_global\ \_ \longleftrightarrow False$
```

Local Variables

Partition variable names into local and global names

is_global — Variable name starts with "G"

fun $is\_global :: vname \Rightarrow bool$ **where**
$is\_global\ [] \longleftrightarrow True$
$|\ is\_global\ (CHR\ ''G''\#\_) \longleftrightarrow True$
$|\ is\_global\ \_ \longleftrightarrow False$

$is\_local\ a = \neg is\_global\ a$

State Combination

$<s_1|s_2>$ – State with locals from s_1, globals from s_2

State Combination

$<s_1|s_2>$ – State with locals from s_1, globals from s_2

$<s|t>\ n = ($*if* $is\_local\ n$ *then* $s\ n$ *else* $t\ n)$

State Combination

$<s_1|s_2>$ – State with locals from s_1, globals from s_2

$<s|t>\ n = ($*if* $is\_local\ n$ ***then*** $s\ n$ ***else*** $t\ n)$

Some rules: $<s|s> = s$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = (\textit{if}\ is\_local\ n\ \textit{then}\ s\ n\ \textit{else}\ t\ n)$

Some rules: $<\!s|s\!> = s$

$<\!s|<\!s'|t\!>\!>$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = ($*if* $is\_local\ n$ ***then*** $s\ n$ ***else*** $t\ n)$

Some rules: $<\!s|s\!> = s$
$<\!s|<\!s'|t\!>\!> =$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = ($*if* $is\_local\ n$ ***then*** $s\ n$ ***else*** $t\ n)$

Some rules: $<\!s|s\!> = s$
$<\!s|<\!s'|t\!>\!> = <\!s|t\!>$
$<\!<\!s|t'\!>|t\!> = <\!s|t\!>$

State Combination

$<s_1|s_2>$ – State with locals from s_1, globals from s_2

$<s|t>\ n = (\textit{if}\ is\_local\ n\ \textit{then}\ s\ n\ \textit{else}\ t\ n)$

Some rules: $<s|s> = s$
$<s|<s'|t>> = <s|t>$
$<<s|t'>|t> = <s|t>$
$is\_local\ x \Longrightarrow <s|t>\ x =$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = (\textsf{\textit{if}}\ is\_local\ n\ \textsf{\textit{then}}\ s\ n\ \textsf{\textit{else}}\ t\ n)$

Some rules: $<\!s|s\!> = s$
$<\!s|<\!s'|t\!>\!> = <\!s|t\!>$
$<\!<\!s|t'\!>|t\!> = <\!s|t\!>$
$is\_local\ x \Longrightarrow <\!s|t\!>\ x = s\ x$
$is\_global\ x \Longrightarrow <\!s|t\!>\ x = t\ x$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = (\textit{if}\ is\_local\ n\ \textit{then}\ s\ n\ \textit{else}\ t\ n)$

Some rules: $<\!s|s\!> = s$

$<\!s|<\!s'|t\!>\!> = <\!s|t\!>$

$<\!<\!s|t'\!>|t\!> = <\!s|t\!>$

$is\_local\ x \Longrightarrow <\!s|t\!>\ x = s\ x$

$is\_global\ x \Longrightarrow <\!s|t\!>\ x = t\ x$

$is\_local\ x \Longrightarrow <\!s|t\!>(x := v) =$

State Combination

$<s_1|s_2>$ – State with locals from s_1, globals from s_2

$<s|t>\ n = ($*if* $is\_local\ n$ *then* $s\ n$ *else* $t\ n)$

Some rules: $<s|s> = s$
$<s|<s'|t>> = <s|t>$
$<<s|t'>|t> = <s|t>$
$is\_local\ x \Longrightarrow <s|t>\ x = s\ x$
$is\_global\ x \Longrightarrow <s|t>\ x = t\ x$
$is\_local\ x \Longrightarrow <s|t>(x := v) = <s(x := v)|t>$

State Combination

$<\!s_1|s_2\!>$ – State with locals from s_1, globals from s_2

$<\!s|t\!>\ n = (\textit{if } is\_local\ n \textit{ then } s\ n \textit{ else } t\ n)$

Some rules: $<\!s|s\!> = s$

$<\!s|<\!s'|t\!>\!> = <\!s|t\!>$

$<<\!s|t'\!>|t\!> = <\!s|t\!>$

$is\_local\ x \Longrightarrow <\!s|t\!>\ x = s\ x$

$is\_global\ x \Longrightarrow <\!s|t\!>\ x = t\ x$

$is\_local\ x \Longrightarrow <\!s|t\!>(x := v) = <\!s(x := v)|t\!>$

$is\_global\ x \Longrightarrow <\!s|t\!>(x := v) = <\!s|t(x := v)\!>$

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Semantics:

$$\pi\text{: } (c,\ <<>|s>) \Rightarrow s' \Longrightarrow \pi\text{: } (SCOPE\ c,\ s) \Rightarrow <s|s'>$$

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Semantics:

$$\pi\text{: } (c,\ <<>|s>) \Rightarrow s' \Longrightarrow \pi\text{: } (SCOPE\ c,\ s) \Rightarrow <s|s'>$$

Unfold rule: $wp\ \pi\ (SCOPE\ c)\ Q\ s$

$=$

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Semantics:

$$\pi\text{: } (c,\ {<}{<}{>}|s{>}) \Rightarrow s' \Longrightarrow \pi\text{: } (SCOPE\ c,\ s) \Rightarrow {<}s|s'{>}$$

Unfold rule: $wp\ \pi\ (SCOPE\ c)\ Q\ s$
$= ?$

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Semantics:

$$\pi\text{: } (c,\ <<>|s>) \Rightarrow s' \Longrightarrow \pi\text{: } (SCOPE\ c,\ s) \Rightarrow <s|s'>$$

Unfold rule: $wp\ \pi\ (SCOPE\ c)\ Q\ s$

$=$

Scope Command

$SCOPE\ c$ — Execute c with fresh set of local variables. Restore original local variables afterwards

Semantics:

$$\pi\text{: } (c,\ {<}{<}{>}|s{>}) \Rightarrow s' \Longrightarrow \pi\text{: } (SCOPE\ c,\ s) \Rightarrow {<}s|s'{>}$$

Unfold rule: $wp\ \pi\ (SCOPE\ c)\ Q\ s$
$\quad = wp\ \pi\ \ c\ (\lambda s'.\ Q\ {<}s|s'{>})\ {<}{<}{>}|s{>}$

Parameter Passing

Pass information over scope boundaries by globals

Parameter Passing

Pass information over scope boundaries by globals

Non-recursive procedure call: $r = f(a_1, \ldots, a_n)$

Parameter Passing

Pass information over scope boundaries by globals

Non-recursive procedure call: $r = f(a_1,\ldots,a_n)$
$G_1 = a_1;\ \ldots;\ G_n = a_n;\ \mathit{inline}\ f;\ r{=}G$

Parameter Passing

Pass information over scope boundaries by globals

Non-recursive procedure call: $r = f(a_1,\ldots,a_n)$
$G_1 = a_1;\ \ldots;\ G_n = a_n;\ \mathit{inline}\ f;\ r{=}G$

Procedure: $f(p_1,\ldots,p_n)\ \{\ \mathit{body};\ \mathit{return}\ x\ \}$

Parameter Passing

Pass information over scope boundaries by globals

Non-recursive procedure call: $r = f(a_1,\ldots,a_n)$
$G_1 = a_1;\ \ldots;\ G_n = a_n;\ inline\ f;\ r{=}G$

Procedure: $f(p_1,\ldots,p_n)\ \{\ body;\ return\ x\ \}$
$scope\ \{\ p_1 = G_1;\ \ldots;\ p_n = G_n;\ body;\ G{=}x\ \}$

Wrapping Specification

Given specification of body: $HT\ P\ body\ Q$
and parameters $p_1,\ldots,p_n$ and return variable x

Wrapping Specification

Given specification of body: $HT\ P\ body\ Q$
and parameters $p_1,\ldots,p_n$ and return variable x

How to derive specification for procedure?
$HT\ P'\ (scope\ \{\ p_1 = G_1;\ \ldots;\ p_n = G_n;\ body;\ G{=}x\ \})\ Q'$

Wrapping Specification

Given specification of body: $HT\ P\ body\ Q$
and parameters p_1,...,p_n and return variable x

How to derive specification for procedure?
$HT\ P'\ (scope\ \{\ p_1\ =\ G_1;\ \ldots;\ p_n\ =\ G_n;\ body;\ G{=}x\ \})\ Q'$

Recall:

$HT\ \pi\ P\ c\ Q \equiv \forall\, s_0.\ P\ s_0 \longrightarrow wp\ \pi\ c\ (Q\ s_0)\ s_0$

Prologue

$$HT\ \pi\ \ P\ body\ \ Q \Longrightarrow$$
$$HT\ \pi\ \ (wp\ \pi\ \ prologue\ P)\ (prologue;;\ body)$$
$$(\lambda s_0\ \ s.\ wp\ \pi\ \ prologue\ (\lambda s_0.\ Q\ s_0\ \ s)\ s_0)$$

Intuition: Weakest precondition to enforce P after prologue

Epilogue

$$\llbracket HT\ \pi\ \ P\ body\ \ Q;\ \forall\, s.\ \exists\, t.\ \pi\colon\ (epilogue,\ s) \Rightarrow\ t\rrbracket$$
$$\Longrightarrow HT\ \pi\ \ P\ (body;;\ epilogue)\ (\lambda s_0.\ sp\ \pi\ \ (Q\ \ s_0)\ epilogue)$$

Intuition: Strongest postcondition we get from Q after epilogue

Strongest Postconditions

$$sp\ \pi\ \ P\ \ c\ \ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$$

Strongest Postconditions

$$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$$

Selected rules:

$$sp\ \pi\ P\ (x[] ::=\ y)\ t$$

Strongest Postconditions

$$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$$

Selected rules:

$$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow$$

Strongest Postconditions

$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\text{:}\ (c,\ s) \Rightarrow\ t$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

Strongest Postconditions

$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t$

Strongest Postconditions

$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow$

Strongest Postconditions

$$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi{:}\ (c,\ s) \Rightarrow\ t$$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow\ t\ x\ =\ t\ y \wedge\ (\exists\, vx.\ P\ (t(x\ :=$
$vx,\ y\ :=\ t\ x)))$

Strongest Postconditions

$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\colon (c,\ s) \Rightarrow\ t$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow\ t\ x\ =\ t\ y \wedge\ (\exists\, vx.\ P\ (t(x\ :=$
$vx,\ y\ :=\ t\ x)))$

$sp\ \pi\ P\ (c_1;;\ c_2)\ t$

Strongest Postconditions

$$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\text{: } (c,\ s) \Rightarrow\ t$$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow t\ x\ =\ t\ y \wedge (\exists\, vx.\ P\ (t(x\ :=$
$vx,\ y\ :=\ t\ x)))$

$sp\ \pi\ P\ (c_1;;\ c_2)\ t \longleftrightarrow$

Strongest Postconditions

$sp\ \pi\ P\ c\ t \equiv \exists\, s.\ P\ s \wedge \pi\text{: } (c,\ s) \Rightarrow\ t$

Selected rules:

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow \exists\, vx.$ *let* $s\ =\ t(x\ :=\ vx)$ *in* $t\ x$
$=\ s\ y \wedge\ P\ s$

$sp\ \pi\ P\ (x[]\ ::=\ y)\ t \longleftrightarrow t\ x = t\ y \wedge (\exists\, vx.\ P\ (t(x\ :=$
$vx,\ y\ :=\ t\ x)))$

$sp\ \pi\ P\ (c_1;;\ c_2)\ t \longleftrightarrow sp\ \pi\ (sp\ \pi\ P\ c_1)\ c_2\ t$

Wrapping Specification

$$HT\ P'\ (scope\ \{\ p_1 = G_1;\ \ldots;\ p_n = G_n;\ body;\ G{=}x\ \})\ Q'$$

Wrapping Specification

$$HT\ P'\ (scope\ \{\ p_1 = G_1;\ \ldots;\ p_n = G_n;\ body;\ G{=}x\ \})\ Q'$$

Derive specification for

Wrapping Specification

$HT\ P'\ (scope\ \{\ p_1\ =\ G_1;\ \ldots;\ p_n\ =\ G_n;\ body;\ G{=}x\ \})\ Q'$

Derive specification for

Parameter assignments: $HT\,\pi\ P\ c\ Q \Longrightarrow$
$HT\,\pi\ (\lambda s.\ P\ (s(x := s\ y)))\ (x[]\ ::=\ y;;\ c)\ (\lambda s_0.\ Q\ (s_0(x := s_0\ y)))$

Wrapping Specification

$HT\ P'\ (scope\ \{\ p_1\ =\ G_1;\ \ldots;\ p_n\ =\ G_n;\ body;\ G{=}x\ \})\ Q'$

Derive specification for

Parameter assignments: $HT\ \pi\ P\ c\ Q \Longrightarrow$
$HT\ \pi\ (\lambda s.\ P\ (s(x := s\ y)))\ (x[]\ ::=\ y;;\ c)\ (\lambda s_0.\ Q\ (s_0(x := s_0\ y)))$

Return value assignment: $HT\ \pi\ P\ c\ Q \Longrightarrow$
$HT\ \pi\ P\ (c;;\ x[]\ ::=\ y)\ (\lambda s_0\ s.\ \exists\, vx.\ Q\ s_0\ (s(x := vx,\ y := s\ x)))$

Wrapping Specification

$HT\ P'\ (scope\ \{\ p_1\ =\ G_1;\ \ldots;\ p_n\ =\ G_n;\ body;\ G{=}x\ \})\ Q'$

Derive specification for

Parameter assignments: $HT\,\pi\ P\ c\ Q \Longrightarrow$
$HT\,\pi\ (\lambda s.\ P\ (s(x := s\ y)))\ (x[]\ ::=\ y;;\ c)\ (\lambda s_0.\ Q\ (s_0(x := s_0\ y)))$

Return value assignment: $HT\,\pi\ P\ c\ Q \Longrightarrow$
$HT\,\pi\ P\ (c;;\ x[]\ ::=\ y)\ (\lambda s_0\ s.\ \exists\, vx.\ Q\ s_0\ (s(x := vx,\ y := s\ x)))$

Scope: $HT\,\pi\ P\ c\ Q \Longrightarrow$
$HT\,\pi\ (\lambda s.\ P\ {<}{<}{>}|s{>})\ (SCOPE\ c)\ (\lambda s_0\ s.\ \exists\, l.\ Q\ {<}{<}{>}|s_0{>}$
${<}l|s{>})$

`IMP2/Examples.thy`

Merge as Procedure

7 Advanced Verification
Arrays
Data Refinement
Local Variables
Recursion

Recursive Procedures

Program is map $pname \rightharpoonup com$

Recursive Procedures

Program is map $pname \rightharpoonup com$

Procedure call command $PCall :: char\ list \Rightarrow com$

Recursive Procedures

Program is map $pname \rightharpoonup com$

Procedure call command $PCall{::}char\ list \Rightarrow com$

Big-Step semantics: $\pi\text{:}\ (c,\ s) \Rightarrow t$
parameterized with program π

Recursive Procedures

Program is map $pname \rightharpoonup com$

Procedure call command $PCall :: char\ list \Rightarrow com$

Big-Step semantics: $\pi\text{: } (c,\ s) \Rightarrow t$
parameterized with program π

$$\llbracket \pi\ p = Some\ c;\ \pi\text{: } (c,\ s) \Rightarrow t \rrbracket \Longrightarrow \pi\text{: } (PCall\ p,\ s) \Rightarrow t$$

Recursive Procedures

Program is map $pname \rightharpoonup com$

Procedure call command $PCall :: char\ list \Rightarrow com$

Big-Step semantics: $\pi \colon (c,\ s) \Rightarrow t$
parameterized with program π

$$[\![\pi\ p = Some\ c;\ \pi\colon (c,\ s) \Rightarrow t]\!] \Longrightarrow \pi\colon (PCall\ p,\ s) \Rightarrow t$$

Note: Gets stuck if procedure does not exist!

Recursive Procedures

Program is map $pname \rightharpoonup com$

Procedure call command $PCall :: char\ list \Rightarrow com$

Big-Step semantics: $\pi \colon (c,\ s) \Rightarrow t$
parameterized with program π

$$\llbracket \pi\ p = Some\ c;\ \pi \colon (c,\ s) \Rightarrow t \rrbracket \Longrightarrow \pi \colon (PCall\ p,\ s) \Rightarrow t$$

Note: Gets stuck if procedure does not exist!
No problem when proving total correctness

Proof Rules for Recursion

Unfolding: $\pi\ p = Some\ c \Longrightarrow wp\ \pi\ (PCall\ p)\ Q\ s = wp\ \pi\ c\ Q\ s$

Proof Rules for Recursion

Unfolding: $\pi\ p = Some\ c \Longrightarrow wp\ \pi\ (PCall\ p)\ Q\ s = wp\ \pi\ c\ Q\ s$

Idea: Well-Founded induction on state

Proof Rules for Recursion

Unfolding: $\pi\ p = Some\ c \Longrightarrow wp\ \pi\ (PCall\ p)\ Q\ s = wp\ \pi\ c\ Q\ s$

Idea: Well-Founded induction on state

assumes $wf\ R$

$\bigwedge s.\ [\![HT\ \pi\ (\lambda s'.\ (s',s) \in R \wedge P\ s')\ (PCall\ p)\ Q;\ P\ s\ ]\!]$
$\Longrightarrow wp\ \pi\ (PCall\ p)\ (Q\ s)\ s$

shows $HT\ \pi\ P\ (PCall\ p)\ Q$

Proof Rules for Recursion

Unfolding: $\pi\ p = Some\ c \Longrightarrow wp\ \pi\ (PCall\ p)\ Q\ s = wp\ \pi\ c\ Q\ s$

Idea: Well-Founded induction on state

assumes $wf\ R$
$\bigwedge s.\ [\![HT\ \pi\ (\lambda s'.\ (s',s) \in R \wedge P\ s')\ (PCall\ p)\ Q;\ P\ s\ ]\!]$
$\Longrightarrow wp\ \pi\ (PCall\ p)\ (Q\ s)\ s$
shows $HT\ \pi\ P\ (PCall\ p)\ Q$

Show specification for state s, assuming it holds for smaller states s'.

Mutual Recursion

Same idea, but for sets of specifications.

Mutual Recursion

Same idea, but for sets of specifications.

$HT'set\ \pi\ \Theta \equiv \forall\,(n,\ P,\ c,\ Q){\in}\Theta.\ HT'\ \pi\ \ P\ \ c\ \ Q$
All Hoare-Triples in Θ valid. Annotation n ignored!

Mutual Recursion

Same idea, but for sets of specifications.

$HT'set\ \pi\ \Theta \equiv \forall\,(n,\ P,\ c,\ Q){\in}\Theta.\ HT'\ \pi\ P\ c\ Q$
All Hoare-Triples in Θ valid. Annotation n ignored!

$ASSUME\_\Theta\ \pi\ f_0\ s_0\ R\ \Theta =$
$(\forall\,(f,\ P,\ c,\ Q){\in}\Theta.\ HT'\ \pi\ (\lambda s.\ (f\ s,\ f_0\ s_0) \in R \wedge P\ s)\ c\ Q)$
Hoare-triples valid for states less than $f_0\ s_0$. Annotation is variant.

Mutual Recursion

Same idea, but for sets of specifications.

$HT'set\ \pi\ \Theta \equiv \forall\,(n,\ P,\ c,\ Q){\in}\Theta.\ HT'\ \pi\ P\ c\ Q$
All Hoare-Triples in Θ valid. Annotation n ignored!

$ASSUME\_\Theta\ \pi\ f_0\ s_0\ R\ \Theta =$
$(\forall\,(f,\ P,\ c,\ Q){\in}\Theta.\ HT'\ \pi\ (\lambda s.\ (f\ s,\ f_0\ s_0) \in R \wedge P\ s)\ c\ Q)$
Hoare-triples valid for states less than $f_0\ s_0$. Annotation is variant.

$PROVE\_\Theta\ \pi\ f_0\ s_0\ \Theta \equiv$
$\forall\,P\ c\ Q.\ (f_0,\ P,\ c,\ Q) \in \Theta \wedge P\ s_0 \longrightarrow wp\ \pi\ (c\ s_0)\ (Q\ s_0)\ s_0$
Hoare-triples valid for fixed variant f_0 and state s_0.

Mutual Recursion

lemma $vcg\_HT'setI$:
assumes $wf\ R$
assumes RL: $\bigwedge f_0\ s_0.\ [\![\ ASSUME\_\Theta\ \pi\ f_0\ s_0\ R\ \Theta\ ]\!] \Longrightarrow PROVE\_\Theta\ \pi\ f_0\ s_0\ \Theta$
shows $HT'set\ \pi\ \Theta$

Fix variant and state,
assume that Hoare-triples hold for smaller states
prove that Hoare-triples hold for this state.

Mutual Recursion

lemma $vcg\_HT'setI$:
assumes $wf\ R$
assumes RL: $\bigwedge f_0\ s_0.\ [\![\ ASSUME\_\Theta\ \pi\ f_0\ s_0\ R\ \Theta\ ]\!] \Longrightarrow PROVE\_\Theta\ \pi\ f_0\ s_0\ \Theta$
shows $HT'set\ \pi\ \Theta$

Fix variant and state,
assume that Hoare-triples hold for smaller states
prove that Hoare-triples hold for this state.

$[\![\pi\ p = Some\ c;\ HT\_mods\ \pi\ mods\ P\ c\ Q]\!] \Longrightarrow HT\_mods\ \pi\ mods\ P\ (PCall\ p)\ Q$

Maps Hoare-Triples to procedure calls

Local Procedures

Idea: Recursive procedure names only valid locally!

Local Procedures

Idea: Recursive procedure names only valid locally!

No need to worry about name clashes!

Local Procedures

Idea: Recursive procedure names only valid locally!

No need to worry about name clashes!

π': $(c,\ s) \Rightarrow\ t \Longrightarrow \pi$: $(PScope\ \pi'\ c,\ s) \Rightarrow\ t$

Call procedure with local procedure environment

Local Procedures

Idea: Recursive procedure names only valid locally!

No need to worry about name clashes!

π': $(c,\ s) \Rightarrow t \Longrightarrow \pi$: $(PScope\ \pi'\ c,\ s) \Rightarrow t$

Call procedure with local procedure environment

$HT\_mods\ \pi\ mods\ P\ (PCall\ p)\ Q \Longrightarrow HT\_mods\ \pi'\ mods\ P\ (PScope\ \pi\ (PCall\ p))\ Q$

Wrap current procedure environment

Specification of Mutual Recursive Procedures

The IMP2 tools take care of

- wf-relation. Default $less\_than$

Specification of Mutual Recursive Procedures

The IMP2 tools take care of

- wf-relation. Default $less\_than$
- parameters and return values.

Specification of Mutual Recursive Procedures

The IMP2 tools take care of

- wf-relation. Default $less\_than$
- parameters and return values.
- variants: expression over parameters.

Specification of Mutual Recursive Procedures

The IMP2 tools take care of

- wf-relation. Default $less\_than$
- parameters and return values.
- variants: expression over parameters.
- localization of procedure environment.

IMP2/Examples.thy

Ackermann, Odd/Even, Merge Sort

Completeness

Consider program with $HT\,\pi\ P\ c\ Q$

Completeness

Consider program with $HT\,\pi\ P\ c\ Q$

Can we always find annotations to get provable VCs?

Completeness

Consider program with $HT\,\pi\ P\ c\ Q$

Can we always find annotations to get provable VCs?

Only consider while-rule here

Partial Correctness

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

Partial Correctness

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

What invariant shall we use?

Partial Correctness

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

What invariant shall we use?

$wlp\ \pi\ c\ Q$!

Total Correctness

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

Invariant: $wp\ \pi\ c\ Q$

Total Correctness

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

Invariant: $wp\ \pi\ c\ Q$

Variant?

Total Correctness

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

Invariant: $wp\ \pi\ c\ Q$

Variant?

Number of iterations until termination!

IMP2/Examples.thy

Completeness of While-Rule

Conclusions

IMP2: Verification of simple programs in Isabelle/HOL
  while-language, arrays, local-vars, recursive procedures
  Tools: concrete syntax for programs and specs, VCG

Conclusions

IMP2: Verification of simple programs in Isabelle/HOL
  while-language, arrays, local-vars, recursive procedures
  Tools: concrete syntax for programs and specs, VCG

Not supported:
  types (char, float, records), pointers, concurrency, ...
  Tools: ghost variables, compiler, ...

Conclusions

IMP2: Verification of simple programs in Isabelle/HOL
 while-language, arrays, local-vars, recursive procedures
 Tools: concrete syntax for programs and specs, VCG

Not supported:
 types (char, float, records), pointers, concurrency, ...
 Tools: ghost variables, compiler, ...

Caveats:
 Procedures+Recursion tools not well-tested
 VCG is slow for many procedures/inlines

Chapter 9

Compiler

8 Stack Machine

9 Compiler

8 Stack Machine

9 Compiler

Stack Machine

Instructions:

datatype $instr =$	
$\quad LOADI\ int$	load value
$\mid LOAD\ vname$	load var
$\mid ADD$	add top of stack

Stack Machine

Instructions:

datatype $instr =$	
$\quad LOADI\ int$	load value
$\mid LOAD\ vname$	load var
$\mid ADD$	add top of stack
$\mid STORE\ vname$	store var

Stack Machine

Instructions:

datatype $instr =$	
$\quad LOADI\ int$	load value
$\mid LOAD\ vname$	load var
$\mid ADD$	add top of stack
$\mid STORE\ vname$	store var
$\mid JMP\ int$	jump

Stack Machine

Instructions:

datatype $instr =$	
$LOADI\ int$	load value
$\mid\ LOAD\ vname$	load var
$\mid\ ADD$	add top of stack
$\mid\ STORE\ vname$	store var
$\mid\ JMP\ int$	jump
$\mid\ JMPLESS\ int$	jump if $<$

Stack Machine

Instructions:

datatype $instr =$

$LOADI\ int$	load value
$\mid LOAD\ vname$	load var
$\mid ADD$	add top of stack
$\mid STORE\ vname$	store var
$\mid JMP\ int$	jump
$\mid JMPLESS\ int$	jump if $<$
$\mid JMPGE\ int$	jump if $\geq$

Semantics

Type synonyms:

$$\begin{array}{rcl} stack & = & int\ list \\ config & = & int \times state \times stack \end{array}$$

Semantics

Type synonyms:

$$\begin{array}{rcl} stack & = & int\ list \\ config & = & int \times state \times stack \end{array}$$

Execution of 1 instruction:

$$iexec :: instr \Rightarrow config \Rightarrow config$$

Instruction execution

$$
\begin{array}{l}
\textit{iexec instr } (i,\ s,\ stk) = \\
(\textsf{case } \textit{instr } \textsf{of } \textit{LOADI } n \Rightarrow (i + 1,\ s,\ n \ \#\ stk) \\
\ \mid \textit{LOAD } x \Rightarrow (i + 1,\ s,\ s\ x \ \#\ stk) \\
\ \mid \textit{ADD} \Rightarrow (i + 1,\ s,\ (\textit{hd2 stk} + \textit{hd stk}) \ \#\ \textit{tl2 stk}) \\
\ \mid \textit{STORE } x \Rightarrow (i + 1,\ s(x := \textit{hd stk}),\ \textit{tl stk}) \\
\ \mid \textit{JMP } n \Rightarrow (i + 1 + n,\ s,\ stk) \\
\ \mid \textit{JMPLESS } n \Rightarrow \\
\qquad (\textsf{if } \textit{hd2 stk} < \textit{hd stk } \textsf{then } i + 1 + n \textsf{ else } i + 1, \\
\qquad\ s,\ \textit{tl2 stk}) \\
\ \mid \textit{JMPGE } n \Rightarrow \\
\qquad (\textsf{if } \textit{hd stk} \le \textit{hd2 stk } \textsf{then } i + 1 + n \textsf{ else } i + 1, \\
\qquad\ s,\ \textit{tl2 stk}))
\end{array}
$$

Program execution (1 step)

Programs are instruction lists.

Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$\mathit{instr\ list} \vdash \mathit{config} \rightarrow \mathit{config}$$

Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$\mathit{instr\ list} \vdash \mathit{config} \rightarrow \mathit{config}$$

$$\begin{aligned}
&P \vdash c \rightarrow c' = \\
&(\exists\, i\ s\ stk. \\
&\quad c = (i,\ s,\ stk) \wedge \\
&\quad c' = iexec\ (P\ !!\ i)\ (i,\ s,\ stk) \wedge \\
&\quad 0 \leq i \wedge i < size\ P)
\end{aligned}$$

Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$\textit{instr list} \vdash \textit{config} \rightarrow \textit{config}$$

$$
\begin{array}{l}
P \vdash c \rightarrow c' = \\
(\exists\, i\ s\ stk. \\
\quad c = (i,\ s,\ stk) \land \\
\quad c' = iexec\ (P\ !!\ i)\ (i,\ s,\ stk) \land \\
\quad 0 \leq i \land i < size\ P)
\end{array}
$$

where	$'a\ list\ !!\ int$	= nth instruction of list
	$size :: 'a\ list \Rightarrow int$	= list size as integer

Program execution ($*$ steps)

Defined in the usual manner:

$$P \vdash (pc,\ s,\ stk) \rightarrow* (pc',\ s',\ stk')$$

`Compiler.thy`

Stack Machine

8 Stack Machine

9 Compiler

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

$$acomp\ a$$
$$\vdash (0,\ s,\ stk) \rightarrow\!* (size\ (acomp\ a),\ s,\ aval\ a\ s\ \#\ stk)$$

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

$$acomp\ a$$
$$\vdash (0,\ s,\ stk) \rightarrow\!* (size\ (acomp\ a),\ s,\ aval\ a\ s\ \#\ stk)$$

Proof by induction on a

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

$$acomp\ a$$
$$\vdash (0,\ s,\ stk) \to\!\ast\ (size\ (acomp\ a),\ s,\ aval\ a\ s\ \#\ stk)$$

Proof by induction on a (with arbitrary stk).

Compiling $aexp$

Same as before:

$$acomp\ (N\ n) = [LOADI\ n]$$
$$acomp\ (V\ x) = [LOAD\ x]$$
$$acomp\ (Plus\ a_1\ a_2) = acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD]$$

Correctness theorem:

$$acomp\ a$$
$$\vdash (0,\ s,\ stk) \to\!* (size\ (acomp\ a),\ s,\ aval\ a\ s\ \#\ stk)$$

Proof by induction on a (with arbitrary stk).

Needs lemmas!

$$P \vdash c \rightarrow\!* \ c' \Longrightarrow P \ @ \ P' \vdash c \rightarrow\!* \ c'$$

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow* (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow* (size\ P' + i',\ s',\ stk')$$

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow* (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow* (size\ P' + i',\ s',\ stk')$$

Proofs by rule induction on $\rightarrow*$,

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i, s, stk) \rightarrow* (i', s', stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i, s, stk) \rightarrow* (size\ P' + i', s', stk')$$

Proofs by rule induction on $\rightarrow*$,
using the corresponding single step lemmas:

$$P \vdash c \rightarrow\!* \; c' \Longrightarrow P \; @ \; P' \vdash c \rightarrow\!* \; c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow\!* \; (i',\ s',\ stk') \Longrightarrow$$
$$P' \; @ \; P \vdash (size\ P' + i,\ s,\ stk) \rightarrow\!* \; (size\ P' + i',\ s',\ stk')$$

Proofs by rule induction on $\rightarrow\!*$,
using the corresponding single step lemmas:

$$P \vdash c \rightarrow c' \Longrightarrow P \; @ \; P' \vdash c \rightarrow c'$$

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow* (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow* (size\ P' + i',\ s',\ stk')$$

Proofs by rule induction on $\rightarrow*$,
using the corresponding single step lemmas:

$$P \vdash c \rightarrow c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow (size\ P' + i',\ s',\ stk')$$

$$P \vdash c \rightarrow* c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow* c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow* (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow* (size\ P' + i',\ s',\ stk')$$

Proofs by rule induction on $\rightarrow*$,
using the corresponding single step lemmas:

$$P \vdash c \rightarrow c' \Longrightarrow P \mathbin{@} P' \vdash c \rightarrow c'$$

$$P \vdash (i,\ s,\ stk) \rightarrow (i',\ s',\ stk') \Longrightarrow$$
$$P' \mathbin{@} P \vdash (size\ P' + i,\ s,\ stk) \rightarrow (size\ P' + i',\ s',\ stk')$$

Proofs by cases.

Compiling $bexp$

Let ins be the compilation of b:

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins
- or it jumps to offset $+n$ beyond ins.

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins
- or it jumps to offset $+n$ beyond ins.

Parameters: when to jump (if b is $True$ or $False$)

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins
- or it jumps to offset $+n$ beyond ins.

Parameters: when to jump (if b is $True$ or $False$)
where to jump to (n)

Compiling $bexp$

Let ins be the compilation of b:

Do not put value of b on the stack
but let value of b determine where execution of ins ends.

Principle:

- Either execution leads to the end of ins
- or it jumps to offset $+n$ beyond ins.

Parameters: when to jump (if b is $True$ or $False$)
where to jump to (n)

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

Example

Let $b = And \quad (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x''$,

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a''))).$

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x'',$
$LOAD\ ''y'',$

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x'',$
$LOAD\ ''y'',$

,

Example

Let $b = And \ \ (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x'',$
$LOAD\ ''y'',$

$,$

$LOAD\ ''z'',$

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x'',$
$LOAD\ ''y'',$

$,$
$LOAD\ ''z'',$
$LOAD\ ''a'',$

Example

Let $b = And \ \ (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x'',$
$\ LOAD\ ''y'',$

$\qquad\qquad\quad ,$
$\ LOAD\ ''z'',$
$\ LOAD\ ''a'',$

$]$

Example

Let $b = And \quad (Less\ (V\ ''x'')\ (V\ ''y''))$
$\qquad\qquad\qquad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x''$,
$LOAD\ ''y''$,
$JMPGE\ 6$,
$LOAD\ ''z''$,
$LOAD\ ''a''$,

$]$

Example

Let $b = And \;\; (Less\ (V\ ''x'')\ (V\ ''y''))$
$\quad\quad\quad\quad\quad\quad (Not\ (Less\ (V\ ''z'')\ (V\ ''a'')))$.

$bcomp\ b\ False\ 3 =$

$[LOAD\ ''x''$,
$LOAD\ ''y''$,
$JMPGE\ 6$,
$LOAD\ ''z''$,
$LOAD\ ''a''$,
$JMPLESS\ 3$
$]$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$$bcomp\ (Bc\ v)\ f\ n = (\textit{if}\ v = f\ \textit{then}\ [JMP\ n]\ \textit{else}\ [])$$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$bcomp\ (Bc\ v)\ f\ n = (\textsf{\textit{if}}\ v = f\ \textsf{\textit{then}}\ [JMP\ n]\ \textsf{\textit{else}}\ [])$

$bcomp\ (Not\ b)\ f\ n = bcomp\ b\ (\neg f)\ n$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$bcomp\ (Bc\ v)\ f\ n = (\textit{if}\ v = f\ \textit{then}\ [JMP\ n]\ \textit{else}\ [])$

$bcomp\ (Not\ b)\ f\ n = bcomp\ b\ (\neg f)\ n$

$bcomp\ (Less\ a_1\ a_2)\ f\ n =$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$bcomp\ (Bc\ v)\ f\ n =$ (*if* $v = f$ *then* $[JMP\ n]$ *else* $[]$)

$bcomp\ (Not\ b)\ f\ n = bcomp\ b\ (\neg f)\ n$

$bcomp\ (Less\ a_1\ a_2)\ f\ n =$

$acomp\ a_1$ @
$acomp\ a_2$ @ (*if* f *then* $[JMPLESS\ n]$ *else* $[JMPGE\ n]$)

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$bcomp\ (Bc\ v)\ f\ n = (\textsf{if}\ v = f\ \textsf{then}\ [JMP\ n]\ \textsf{else}\ [])$

$bcomp\ (Not\ b)\ f\ n = bcomp\ b\ (\neg f)\ n$

$bcomp\ (Less\ a_1\ a_2)\ f\ n =$

$acomp\ a_1\ @$
$acomp\ a_2\ @\ (\textsf{if}\ f\ \textsf{then}\ [JMPLESS\ n]\ \textsf{else}\ [JMPGE\ n])$

$bcomp\ (And\ b_1\ b_2)\ f\ n =$

$$bcomp :: bexp \Rightarrow bool \Rightarrow int \Rightarrow instr\ list$$

$bcomp\ (Bc\ v)\ f\ n = (\textsf{if}\ v = f\ \textsf{then}\ [JMP\ n]\ \textsf{else}\ [])$

$bcomp\ (Not\ b)\ f\ n = bcomp\ b\ (\neg f)\ n$

$bcomp\ (Less\ a_1\ a_2)\ f\ n =$

$acomp\ a_1\ @$
$acomp\ a_2\ @\ (\textsf{if}\ f\ \textsf{then}\ [JMPLESS\ n]\ \textsf{else}\ [JMPGE\ n])$

$bcomp\ (And\ b_1\ b_2)\ f\ n =$

```
let $cb_2 = bcomp\ b_2\ f\ n;$
    $m = \textsf{if}\ f\ \textsf{then}\ size\ cb_2\ \textsf{else}\ size\ cb_2 + n;$
    $cb_1 = bcomp\ b_1\ False\ m$
in $cb_1\ @\ cb_2$
```

Correctness of $bcomp$

Correctness of $bcomp$

$$
\begin{array}{l}
0 \le n \Longrightarrow \\
bcomp\ b\ f\ n \\
\vdash (0,\ s,\ stk) \rightarrow\!* \\
\quad (size\ (bcomp\ b\ f\ n) + (\textit{if}\ f = bval\ b\ s\ \textit{then}\ n\ \textit{else}\ 0), \\
\quad\ s,\ stk)
\end{array}
$$

Compiling com

$ccomp :: com \Rightarrow instr\ list$

Compiling com

$ccomp :: com \Rightarrow instr\ list$

$ccomp\ SKIP = []$

Compiling com

$ccomp :: com \Rightarrow instr\ list$

$ccomp\ SKIP = []$

$ccomp\ (x ::= a) = acomp\ a\ @\ [STORE\ x]$

Compiling com

$ccomp :: com \Rightarrow instr\ list$

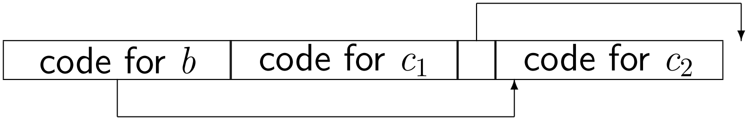
$ccomp\ SKIP = []$

$ccomp\ (x ::= a) = acomp\ a\ @\ [STORE\ x]$

$ccomp\ (c_1;;\ c_2) = ccomp\ c_1\ @\ ccomp\ c_2$

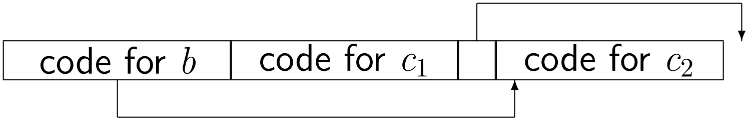
$$ccomp\ (IF\ b\ \ THEN\ c_1\ \ ELSE\ c_2) =$$

$ccomp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$



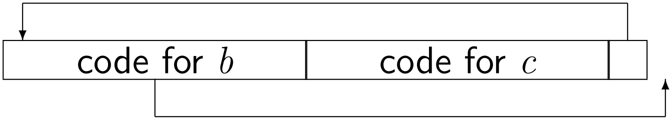
$ccomp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$

let $cc_1 = ccomp\ c_1;\ cc_2 = ccomp\ c_2;$
$\quad cb = bcomp\ b\ False\ (size\ cc_1 + 1)$
in $cb\ @\ cc_1\ @\ JMP\ (size\ cc_2)\ \#\ cc_2$



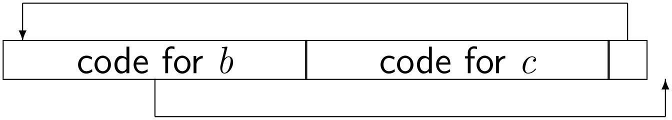
$$ccomp\ (WHILE\ b\ DO\ c) =$$

$ccomp\ (WHILE\ b\ DO\ c) =$



$ccomp\ (WHILE\ b\ DO\ c) =$

let $cc = ccomp\ c;\ cb = bcomp\ b\ False\ (size\ cc + 1)$
in $cb\ @\ cc\ @\ [JMP\ (-\ (size\ cb + size\ cc + 1))]$



Correctness of $ccomp$

If the source code produces a certain result,
so should the compiled code:

Correctness of $ccomp$

If the source code produces a certain result,
so should the compiled code:

$$(c,\ s) \Rightarrow t \Longrightarrow$$
$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow\!* \ (size\ (ccomp\ c),\ t,\ stk)$$

Correctness of *ccomp*

If the source code produces a certain result, so should the compiled code:

$$(c,\ s) \Rightarrow t \Longrightarrow$$
$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (size\ (ccomp\ c),\ t,\ stk)$$

Proof by rule induction.

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,
and if(!) $(c,\ s) \Rightarrow t'$,

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,
and if(!) $(c,\ s) \Rightarrow t'$, then "$\Longrightarrow$" implies
that $ccomp\ c$ with start state s must also produce t'

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,
and if(!) $(c,\ s) \Rightarrow t'$, then "$\Longrightarrow$" implies
that $ccomp\ c$ with start state s must also produce t'
and thus $t' = t$ (why?).

The other direction

We have only shown "$\Longrightarrow$":

compiled code simulates source code.

How about "$\Longleftarrow$":

source code simulates compiled code?

If $ccomp\ c$ with start state s produces result t,
and if(!) $(c,\ s) \Rightarrow t'$, then "$\Longrightarrow$" implies
that $ccomp\ c$ with start state s must also produce t'
and thus $t' = t$ (why?).

But we have *not* ruled out this potential error:

c does not terminate but $ccomp\ c$ does.

The other direction

Two approaches:

- In the absence of nondeterminism:
  Prove that $ccomp$ preserves nontermination.

The other direction

Two approaches:

- In the absence of nondeterminism:
  Prove that $ccomp$ preserves nontermination.
  A nice proof of this fact requires *coinduction*.

The other direction

Two approaches:

- In the absence of nondeterminism:
  Prove that $ccomp$ preserves nontermination.
  A nice proof of this fact requires *coinduction*.
  Isabelle supports coinduction, this course avoids it.

The other direction

Two approaches:

- In the absence of nondeterminism:
  Prove that $ccomp$ preserves nontermination.
  A nice proof of this fact requires *coinduction*.
  Isabelle supports coinduction, this course avoids it.
- A direct proof: theory *Compiler2*

$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow\!* (size\ (ccomp\ c),\ t,\ stk') \Longrightarrow (c,\ s) \Rightarrow t$$

Chapter 10

Types

10 A Typed Version of IMP

10 A Typed Version of IMP

10 A Typed Version of IMP

Remarks on Type Systems

Typed IMP: Semantics

Typed IMP: Type System

Type Safety of Typed IMP

Why Types?

Why Types?

To prevent mistakes, dummy!

There are 3 kinds of types

There are 3 kinds of types

The Good Static types that *guarantee* absence of certain runtime faults.

There are 3 kinds of types

The Good Static types that *guarantee* absence of certain runtime faults.
Example: no memory access errors in Java.

There are 3 kinds of types

The Good
: Static types that *guarantee* absence of certain runtime faults.
  Example: no memory access errors in Java.

The Bad
: Static types that have mostly decorative value but do not guarantee anything at runtime.

There are 3 kinds of types

The Good	Static types that *guarantee* absence of certain runtime faults. Example: no memory access errors in Java.
The Bad	Static types that have mostly decorative value but do not guarantee anything at runtime. Example: C, C++

There are 3 kinds of types

The Good	Static types that *guarantee* absence of certain runtime faults. Example: no memory access errors in Java.
The Bad	Static types that have mostly decorative value but do not guarantee anything at runtime. Example: C, C++
The Ugly	Dynamic types that detect errors when it can be too late.

There are 3 kinds of types

our focus The Good	Static types that *guarantee* absence of certain runtime faults. Example: no memory access errors in Java.
The Bad	Static types that have mostly decorative value but do not guarantee anything at runtime. Example: C, C++
The Ugly	Dynamic types that detect errors when it can be too late. Example: "TypeError: ..." in Python.

The ideal

Well-typed programs cannot go wrong.

The ideal

Well-typed programs cannot go wrong.

Robin Milner, *A Theory of Type Polymorphism in Programming*, 1978.

The ideal

Well-typed programs cannot go wrong.

Robin Milner, *A Theory of Type Polymorphism in Programming*, 1978.

The most influential slogan and one of the most influential papers in programming language theory.

What could go wrong?

1. Corruption of data

What could go wrong?

1. Corruption of data
2. Null pointer exception

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory
5. Secret leaked

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory
5. Secret leaked
6. and many more . . .

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory
5. Secret leaked
6. and many more . . .

There are type systems for *everything* (and more)

What could go wrong?

1. Corruption of data
2. Null pointer exception
3. Nontermination
4. Run out of memory
5. Secret leaked
6. and many more . . .

There are type systems for *everything* (and more) but in practice (Java, C#) only 1 is covered.

Type safety

A programming language is *type safe* if the execution of a well-typed program cannot lead to certain errors.

Type safety

A programming language is *type safe* if the execution of a well-typed program cannot lead to certain errors.

Java and the JVM have been *proved* to be type safe.

Type safety

A programming language is *type safe* if the execution of a well-typed program cannot lead to certain errors.

Java and the JVM have been *proved* to be type safe. (Note: Java exceptions are not errors!)

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

The semantics is the primary definition,
the type system must be justified w.r.t. it.

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

The semantics is the primary definition,
the type system must be justified w.r.t. it.

How about completeness?

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

The semantics is the primary definition,
the type system must be justified w.r.t. it.

How about completeness? Remember Rice:

> *Nontrivial semantic properties of programs*
> *(e.g. termination) are undecidable.*

Correctness and completeness

Type soundness means that the type system is *sound/correct* w.r.t. the semantics:

> *If the type system says yes,*
> *the semantics does not lead to an error.*

The semantics is the primary definition,
the type system must be justified w.r.t. it.

How about completeness? Remember Rice:

> *Nontrivial semantic properties of programs*
> *(e.g. termination) are undecidable.*

Hence there is no (decidable) type system that accepts *all* programs that have a certain semantic property.

Automatic analysis of semantic program properties
is necessarily incomplete.

10 A Typed Version of IMP
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP

Arithmetic

Values:

datatype $val = Iv\ int \mid Rv\ real$

Arithmetic

Values:

datatype $val = Iv\ int \mid Rv\ real$

The state:

$state = vname \Rightarrow val$

Arithmetic

Values:

datatype $val = Iv\ int \mid Rv\ real$

The state:

$state = vname \Rightarrow val$

Arithmetic expresssions:

datatype $aexp =$
$\quad Ic\ int \mid Rc\ real \mid V\ vname \mid Plus\ aexp\ aexp$

Why tagged values?

Because we want to detect if things “go wrong”.

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong?

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!
No automatic coercions.

Why tagged values?

Because we want to detect if things "go wrong".

What can go wrong? Adding integer and real!
No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!
No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!
No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

Tags are only used to detect certain errors

Why tagged values?

Because we want to detect if things “go wrong”.

What can go wrong? Adding integer and real!
No automatic coercions.

Does this mean any implementation of IMP also needs to tag values?

No! Compilers compile only well-typed programs, and well-typed programs do not need tags.

Tags are only used to detect certain errors
and to prove that the type system avoids those errors.

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

$$taval\ (Rc\ r)\ s\ (Rv\ r)$$

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

$$taval\ (Rc\ r)\ s\ (Rv\ r)$$

$$taval\ (V\ x)\ s\ (s\ x)$$

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

$$taval\ (Rc\ r)\ s\ (Rv\ r)$$

$$taval\ (V\ x)\ s\ (s\ x)$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))}$$

Evaluation of $aexp$

Not recursive function but inductive predicate:

$$taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$$

$$taval\ (Ic\ i)\ s\ (Iv\ i)$$

$$taval\ (Rc\ r)\ s\ (Rv\ r)$$

$$taval\ (V\ x)\ s\ (s\ x)$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Rv\ (r_1 + r_2))}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i)}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s\ (Iv\ 1)}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s\ (Iv\ 1)}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s\ (Iv(i+1))}$$

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s\ (Iv\ 1)}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s\ (Iv(i+1))}$$

If $s\ ''x'' = Rv\ r$:

Example: evaluation of $Plus\ (V\ ''x'')\ (Ic\ 1)$

If $s\ ''x'' = Iv\ i$:

$$\frac{taval\ (V\ ''x'')\ s\ (Iv\ i) \qquad taval\ (Ic\ 1)\ s\ (Iv\ 1)}{taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s\ (Iv(i+1))}$$

If $s\ ''x'' = Rv\ r$: then there is *no* value v such that $taval\ (Plus\ (V\ ''x'')\ (Ic\ 1))\ s\ v$.

The functional alternative

$$taval :: aexp \Rightarrow state \Rightarrow val\ option$$

The functional alternative

$$taval :: aexp \Rightarrow state \Rightarrow val\ option$$

Exercise!

Boolean expressions

Syntax as before. Semantics:

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$tbval\ (Bc\ v)\ s\ v$

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$$tbval\ (Bc\ v)\ s\ v \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$$tbval\ (Bc\ v)\ s\ v \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

$$\frac{tbval\ b_1\ s\ bv_1 \qquad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)}$$

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$$tbval\ (Bc\ v)\ s\ v \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

$$\frac{tbval\ b_1\ s\ bv_1 \qquad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)}$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (i_1 < i_2)}$$

Boolean expressions

Syntax as before. Semantics:

$$tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$$

$$tbval\ (Bc\ v)\ s\ v \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

$$\frac{tbval\ b_1\ s\ bv_1 \qquad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)}$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (i_1 < i_2)}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (r_1 < r_2)}$$

com: big or small steps?

We need to detect if things “go wrong”.

com: big or small steps?

We need to detect if things "go wrong".

- Big step semantics:
  Cannot model error by absence of final state.

com: big or small steps?

We need to detect if things "go wrong".

- Big step semantics:
  Cannot model error by absence of final state.
  Would confuse error and nontermination.

com: big or small steps?

We need to detect if things “go wrong”.

- Big step semantics:
  Cannot model error by absence of final state.
  Would confuse error and nontermination.
  Could introduce an extra error-element, e.g.
  $big\_step :: com \times state \Rightarrow state\ option \Rightarrow bool$

com: big or small steps?

We need to detect if things "go wrong".

- Big step semantics:
  Cannot model error by absence of final state.
  Would confuse error and nontermination.
  Could introduce an extra error-element, e.g.
  $big\_step :: com \times state \Rightarrow state\ option \Rightarrow bool$
  Complicates formalization.

com: big or small steps?

We need to detect if things "go wrong".

- Big step semantics:
  Cannot model error by absence of final state.
  Would confuse $\overset{\text{type}}{\text{error}}$ and nontermination.
  Could introduce an extra error-element, e.g.
  $big\_step :: com \times state \Rightarrow state\ option \Rightarrow bool$
  Complicates formalization.
- Small step semantics:
  error = semantics gets stuck

Small step semantics

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \rightarrow (SKIP,\ s(x := v))}$$

Small step semantics

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \rightarrow (SKIP,\ s(x := v))}$$

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

Small step semantics

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \to (SKIP,\ s(x := v))}$$

$$\frac{tbval\ b\ s\ True}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_1,\ s)}$$

$$\frac{tbval\ b\ s\ False}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_2,\ s)}$$

Small step semantics

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \rightarrow (SKIP,\ s(x := v))}$$

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

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

The other rules remain unchanged.

typical exam question: give a derivation with these rules

Example

Let $c \;=\; (''x'' ::= Plus\ (V\ ''x'')\ (Ic\ 1))$.

Example

Let $c \;=\; (''x'' ::= \; Plus \; (\, V \; ''x'') \; (Ic \; 1))$.

- If $s \; ''x'' = \; Iv \; i$:

Example

Let $c \ = \ (''x'' ::= Plus\ (V\ ''x'')\ (Ic\ 1))$.

- If $s\ ''x'' = Iv\ i$:
  $(c,\ s) \rightarrow (SKIP,\ s(''x'' := Iv\ (i+1)))$

Example

Let $c \; = \; (''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (\mathit{Ic}\ 1))$.

- If $s\ ''x'' = \mathit{Iv}\ i$:
  $(c,\ s) \rightarrow (\mathit{SKIP},\ s(''x'' := \mathit{Iv}\ (i + 1)))$
- If $s\ ''x'' = \mathit{Rv}\ r$:

Example

Let $c \;=\; (''x'' ::= Plus\ (V\ ''x'')\ (Ic\ 1))$.

- If $s\ ''x'' = Iv\ i$:
  $(c,\ s) \to (SKIP,\ s(''x'' := Iv\ (i + 1)))$
- If $s\ ''x'' = Rv\ r$:
  $(c,\ s) \not\to$

10 A Typed Version of IMP
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP

Type system

There are two types:

datatype $ty = Ity \mid Rty$

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ ''x'')\ (V\ ''y'')$?

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ ''x'')\ (V\ ''y'')$?

Depends on the type of $''x''$ and $''y''$!

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ ''x'')\ (V\ ''y'')$?

Depends on the type of $''x''$ and $''y''$!

A *type environment* maps variable names to their types:

$tyenv = vname \Rightarrow ty$

Type system

There are two types:

datatype $ty = Ity \mid Rty$

What is the type of $Plus\ (V\ ''x'')\ (V\ ''y'')$?

Depends on the type of $''x''$ and $''y''$!

A *type environment* maps variable names to their types:

$tyenv = vname \Rightarrow ty$

The type of an expression is always relative to a type enviroment Γ. Standard notation:

$$\Gamma \vdash e : \tau$$

Read: *In the context of* Γ, e *has type* τ

The type of an $aexp$

$\Gamma \vdash a : \tau$

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The rules:

$$\Gamma \vdash Ic\ i : Ity$$

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The rules:

$$\Gamma \vdash Ic\ i : Ity$$

$$\Gamma \vdash Rc\ r : Rty$$

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The rules:

$$\Gamma \vdash Ic\ i : Ity$$

$$\Gamma \vdash Rc\ r : Rty$$

$$\Gamma \vdash V\,x : \Gamma\ x$$

The type of an $aexp$

$$\Gamma \vdash a : \tau$$
$$tyenv \vdash aexp : ty$$

The rules:

$$\Gamma \vdash Ic\ i : Ity$$

$$\Gamma \vdash Rc\ r : Rty$$

$$\Gamma \vdash V\ x : \Gamma\ x$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Plus\ a_1\ a_2 : \tau}$$

Example

$$\frac{\vdots}{\Gamma \vdash \ Plus\ (V\ ''x'')\ (Plus\ (V\ ''x'')\ (Ic\ 0))\ :\ ?}$$

where $\Gamma\ ''x'' = Ity$.

Well-typed $bexp$

Notation:

$$\Gamma \vdash b$$

Well-typed $bexp$

Notation:

$$\Gamma \vdash b$$
$$tyenv \vdash bexp$$

Well-typed $bexp$

Notation:

$$\Gamma \vdash b$$
$$tyenv \vdash bexp$$

Read: *In context Γ, b is well-typed*.

The rules:

$$\Gamma \vdash Bc\ v$$

The rules:

$$\Gamma \vdash \mathit{Bc}\ v$$

$$\frac{\Gamma \vdash b}{\Gamma \vdash \mathit{Not}\ b}$$

The rules:

$$\Gamma \vdash \; Bc \; v$$

$$\frac{\Gamma \vdash \; b}{\Gamma \vdash \; Not \; b}$$

$$\frac{\Gamma \vdash \; b_1 \qquad \Gamma \vdash \; b_2}{\Gamma \vdash \; And \; b_1 \; b_2}$$

The rules:

$$\Gamma \vdash Bc\ v$$

$$\frac{\Gamma \vdash b}{\Gamma \vdash Not\ b}$$

$$\frac{\Gamma \vdash b_1 \qquad \Gamma \vdash b_2}{\Gamma \vdash And\ b_1\ b_2}$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Less\ a_1\ a_2}$$

The rules:

$$\Gamma \vdash Bc\ v$$

$$\frac{\Gamma \vdash b}{\Gamma \vdash Not\ b}$$

$$\frac{\Gamma \vdash b_1 \qquad \Gamma \vdash b_2}{\Gamma \vdash And\ b_1\ b_2}$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Less\ a_1\ a_2}$$

Example: $\Gamma \vdash Less\ (Ic\ i)\ (Rc\ r)$ does not hold.

Well-typed commands

Notation:

$$\Gamma \vdash c$$

Well-typed commands

Notation:

$$\Gamma \vdash c$$
$$tyenv \vdash com$$

Well-typed commands

Notation:

$$\Gamma \vdash c$$
$$tyenv \vdash com$$

Read: *In context Γ, c is well-typed*.

The rules:

$$\Gamma \vdash SKIP$$

The rules:

$$\Gamma \vdash SKIP \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

The rules:

$$\Gamma \vdash SKIP \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1;;\ c_2}$$

The rules:

$$\Gamma \vdash \mathit{SKIP} \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1;;\ c_2}$$

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2}$$

The rules:

$$\Gamma \vdash \mathit{SKIP} \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1;;\ c_2}$$

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2}$$

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash c}{\Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

- There is exactly one rule for each syntactic construct ($SKIP$, $::=$, $\ldots$).

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

- There is exactly one rule for each syntactic construct ($SKIP$, $::=$, ...).
- Well-typedness of a term $C\ t_1 \ldots t_n$ depends only on the well-typedness of its subterms t_1, ..., t_n.

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

- There is exactly one rule for each syntactic construct ($SKIP$, $::=$, ...).
- Well-typedness of a term $C\ t_1 \dots t_n$ depends only on the well-typedness of its subterms t_1, ..., t_n.

A syntax-directed set of rules

- is executable by backchaining without backtracking

Syntax-directedness

All three sets of typing rules are *syntax-directed*:

- There is exactly one rule for each syntactic construct ($SKIP$, $::=$, $\ldots$).
- Well-typedness of a term $C\ t_1 \ldots t_n$ depends only on the well-typedness of its subterms t_1, $\ldots$, t_n.

A syntax-directed set of rules

- is executable by backchaining without backtracking and
- backchaining terminates and requires at most as many steps as the size of the term.

Syntax-directedness

The big-step semantics is not syntax-directed:

Syntax-directedness

The big-step semantics is not syntax-directed:

- more than one rule per construct and

Syntax-directedness

The big-step semantics is not syntax-directed:

- more than one rule per construct and
- the execution of $WHILE$ depends on the execution of $WHILE$.

⓾ A Typed Version of IMP
Remarks on Type Systems
Typed IMP: Semantics
Typed IMP: Type System
Type Safety of Typed IMP

Well-typed states

Even well-typed programs can get stuck . . .

Well-typed states

Even well-typed programs can get stuck . . .
. . . if they start in an unsuitable state.

Well-typed states

Even well-typed programs can get stuck ...
... if they start in an unsuitable state.

Remember:
If $s\ ''x'' = Rv\ r$
then $(''x'' ::= Plus\ (V\ ''x'')\ (Ic\ 1),\ s) \not\to$

Well-typed states

Even well-typed programs can get stuck . . .
. . . if they start in an unsuitable state.

Remember:
If $s\ ''x'' = Rv\ r$
then $(''x'' ::= Plus\ (V\ ''x'')\ (Ic\ 1),\ s) \not\to$

The state must be well-typed w.r.t. Γ.

The type of a value:

$$type\ (Iv\ i) = Ity$$
$$type\ (Rv\ r) = Rty$$

The type of a value:

$$\begin{aligned}&type\ (Iv\ i) = Ity\\&type\ (Rv\ r) = Rty\end{aligned}$$

Well-typed state:

$$\Gamma \vdash s \longleftrightarrow (\forall x.\ type\ (s\ x) = \Gamma\ x)$$

Type soundness

Reduction cannot get stuck:

Type soundness

Reduction cannot get stuck:

If everything is ok ($\Gamma \vdash s$, $\Gamma \vdash c$ *),*

Type soundness

Reduction cannot get stuck:

> *If everything is ok (* $\Gamma \vdash s$*,* $\Gamma \vdash c$ *),*
> *and you take a finite number of steps,*

Type soundness

Reduction cannot get stuck:

If everything is ok ($\Gamma \vdash s$, $\Gamma \vdash c$ *),*
and you take a finite number of steps,
and you have not reached SKIP,

Type soundness

Reduction cannot get stuck:

> *If everything is ok ($\Gamma \vdash s$, $\Gamma \vdash c$),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Type soundness

Reduction cannot get stuck:

> *If everything is ok (* $\Gamma \vdash s$, $\Gamma \vdash c$ *),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Follows from *progress*:

Type soundness

Reduction cannot get stuck:

> *If everything is ok ($\Gamma \vdash s$, $\Gamma \vdash c$),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Follows from *progress*:

> *If everything is ok and you have not reached SKIP,*
> *then you can take one more step.*

Type soundness

Reduction cannot get stuck:

> *If everything is ok ($\Gamma \vdash s$, $\Gamma \vdash c$),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Follows from *progress*:

> *If everything is ok and you have not reached SKIP,*
> *then you can take one more step.*

and *preservation*:

Type soundness

Reduction cannot get stuck:

> *If everything is ok (* $\Gamma \vdash s$, $\Gamma \vdash c$ *),*
> *and you take a finite number of steps,*
> *and you have not reached SKIP,*
> *then you can take one more step.*

Follows from *progress*:

> *If everything is ok and you have not reached SKIP,*
> *then you can take one more step.*

and *preservation*:

> *If everything is ok and you take a step,*
> *then everything is ok again.*

The slogan

$$\text{Progress} \wedge \text{Preservation} \implies \text{Type safety}$$

The slogan

$$\text{Progress} \land \text{Preservation} \implies \text{Type safety}$$

Progress Well-typed programs do not get stuck.

The slogan

$$\text{Progress} \wedge \text{Preservation} \implies \text{Type safety}$$

Progress	Well-typed programs do not get stuck.
Preservation	Well-typedness is preserved by reduction.

The slogan

$$\text{Progress} \land \text{Preservation} \implies \text{Type safety}$$

Progress	Well-typed programs do not get stuck.
Preservation	Well-typedness is preserved by reduction.

Preservation: Well-typedness is an *invariant*.

com

Progress:

$$\llbracket \Gamma \vdash c;\ \Gamma \vdash s;\ c \neq SKIP \rrbracket \Longrightarrow \exists\, cs'.\ (c,\ s) \rightarrow cs'$$

com

Progress:

$$[\![\Gamma \vdash c;\ \Gamma \vdash s;\ c \neq SKIP]\!] \Longrightarrow \exists\, cs'.\ (c,\ s) \rightarrow cs'$$

Preservation:

$$[\![(c,\ s) \rightarrow (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s]\!] \Longrightarrow \Gamma \vdash s'$$

$$[\![(c,\ s) \rightarrow (c',\ s');\ \Gamma \vdash c]\!] \Longrightarrow \Gamma \vdash c'$$

com

Progress:

$$[\![\Gamma \vdash c;\ \Gamma \vdash s;\ c \neq \mathit{SKIP}]\!] \Longrightarrow \exists\, cs'.\ (c,\ s) \rightarrow cs'$$

Preservation:

$$[\![(c,\ s) \rightarrow (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s]\!] \Longrightarrow \Gamma \vdash s'$$

$$[\![(c,\ s) \rightarrow (c',\ s');\ \Gamma \vdash c]\!] \Longrightarrow \Gamma \vdash c'$$

Type soundness:

$$[\![(c,\ s) \rightarrow\!* \ (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s;\ c' \neq \mathit{SKIP}]\!]$$
$$\Longrightarrow \exists\, cs''.\ (c',\ s') \rightarrow cs''$$

$bexp$

Progress:

$$[\![\Gamma \vdash\ b;\ \Gamma \vdash\ s]\!] \Longrightarrow \exists\, v.\ tbval\ b\ s\ v$$

$aexp$

Progress:

$$\llbracket \Gamma \vdash a : \tau;\ \Gamma \vdash s \rrbracket \Longrightarrow \exists\, v.\ \mathit{taval}\ a\ s\ v$$

Preservation:

$$\llbracket \Gamma \vdash a : \tau;\ \mathit{taval}\ a\ s\ v;\ \Gamma \vdash s \rrbracket \Longrightarrow \mathit{type}\ v = \tau$$

All proofs by rule induction.

Types.thy

The mantra

Type systems have a purpose:

*The static analysis of programs
in order to predict their runtime behaviour.*

The mantra

Type systems have a purpose:

> *The static analysis of programs*
> *in order to predict their runtime behaviour.*

The correctness of the prediction must be provable.

Chapter 11

Data-Flow Analyses and Optimization

11 Definite Initialization Analysis

12 Live Variable Analysis

11 Definite Initialization Analysis

12 Live Variable Analysis

Each local variable must have a definitely assigned value when any access of its value occurs. A compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable x*,* x *is definitely assigned before the access; otherwise a compile-time error must occur.*

Each local variable must have a definitely assigned value when any access of its value occurs. A compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable x, x *is definitely assigned before the access; otherwise a compile-time error must occur.*

Java Language Specification

Each local variable must have a definitely assigned value when any access of its value occurs. A compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable x, x is definitely assigned before the access; otherwise a compile-time error must occur.

Java Language Specification

Java was the first language to force programmers to initialize their variables.

Examples: ok or not?

Examples: ok or not?

Assume `x` is initialized:

```
IF x < 1 THEN y := x ELSE y := x + 1;
y := y + 1
```

Examples: ok or not?

Assume `x` is initialized:

```
IF x < 1 THEN y := x ELSE y := x + 1;
y := y + 1
```

```
IF x < x THEN y := y + 1 ELSE y := x
```

Examples: ok or not?

Assume `x` is initialized:

```
IF x < 1 THEN y := x ELSE y := x + 1;
y := y + 1
```

```
IF x < x THEN y := y + 1 ELSE y := x
```

Assume `x` and `y` are initialized:

```
WHILE x < y DO z := x; z := z + 1
```

Simplifying principle

We do not analyze boolean expressions to determine program execution.

11 Definite Initialization Analysis

Prelude: Variables in Expressions

Definite Initialization Analysis

Initialization Sensitive Semantics

Theory $Vars$ provides an overloaded function $vars$:

$vars :: aexp \Rightarrow vname\ set$

Theory $Vars$ provides an overloaded function $vars$:

$vars :: aexp \Rightarrow vname\ set$

$vars\ (N\ n) = \{\}$

$vars\ (V\ x) = \{x\}$

$vars\ (Plus\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

Theory $Vars$ provides an overloaded function $vars$:

$vars :: aexp \Rightarrow vname\ set$
$vars\ (N\ n) = \{\}$
$vars\ (V\ x) = \{x\}$
$vars\ (Plus\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

$vars :: bexp \Rightarrow vname\ set$

Theory $Vars$ provides an overloaded function $vars$:

$vars :: aexp \Rightarrow vname\ set$
$vars\ (N\ n) = \{\}$
$vars\ (V\ x) = \{x\}$
$vars\ (Plus\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

$vars :: bexp \Rightarrow vname\ set$
$vars\ (Bc\ v) = \{\}$
$vars\ (Not\ b) = vars\ b$
$vars\ (And\ b_1\ b_2) = vars\ b_1 \cup vars\ b_2$
$vars\ (Less\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

Vars.thy

11 Definite Initialization Analysis
Prelude: Variables in Expressions
Definite Initialization Analysis
Initialization Sensitive Semantics

Modified example from the JLS:

> *Variable x is definitely initialized after $SKIP$ iff x is definitely initialized before $SKIP$.*

Modified example from the JLS:

> *Variable x is definitely initialized after $SKIP$*
> *iff x is definitely initialized before $SKIP$.*

Similar statements for each language construct.

$$D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$$

$D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$

$D\ A\ c\ A'$ should imply:

$D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$

$D\ A\ c\ A'$ should imply:

If all variables in A are initialized before c is executed,

$D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$

$D\ A\ c\ A'$ should imply:

If all variables in A are initialized before c is executed, then no uninitialized variable is accessed during execution,

$D :: \mathit{vname\ set} \Rightarrow \mathit{com} \Rightarrow \mathit{vname\ set} \Rightarrow \mathit{bool}$

$D\ A\ c\ A'$ should imply:

If all variables in A are initialized before c is executed, then no uninitialized variable is accessed during execution, and all variables in A' are initialized afterwards.

D A SKIP A

$$\frac{D\ A\ SKIP\ A \qquad vars\ a \subseteq A}{D\ A\ (x ::= a)\ (insert\ x\ A)}$$

$$D\ A\ SKIP\ A$$

$$\frac{vars\ a \subseteq A}{D\ A\ (x ::= a)\ (insert\ x\ A)}$$

$$\frac{D\ A_1\ c_1\ A_2 \qquad D\ A_2\ c_2\ A_3}{D\ A_1\ (c_1;;\ c_2)\ A_3}$$

$$D\ A\ \mathit{SKIP}\ A$$

$$\frac{\mathit{vars}\ a \subseteq A}{D\ A\ (x ::= a)\ (\mathit{insert}\ x\ A)}$$

$$\frac{D\ A_1\ c_1\ A_2 \qquad D\ A_2\ c_2\ A_3}{D\ A_1\ (c_1;;\ c_2)\ A_3}$$

$$\frac{\mathit{vars}\ b \subseteq A \qquad D\ A\ c_1\ A_1 \qquad D\ A\ c_2\ A_2}{D\ A\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ (A_1 \cap A_2)}$$

$$D\ A\ \mathit{SKIP}\ A$$

$$\frac{\mathit{vars}\ a \subseteq A}{D\ A\ (x ::= a)\ (\mathit{insert}\ x\ A)}$$

$$\frac{D\ A_1\ c_1\ A_2 \qquad D\ A_2\ c_2\ A_3}{D\ A_1\ (c_1;;\ c_2)\ A_3}$$

$$\frac{\mathit{vars}\ b \subseteq A \qquad D\ A\ c_1\ A_1 \qquad D\ A\ c_2\ A_2}{D\ A\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ (A_1 \cap A_2)}$$

$$\frac{\mathit{vars}\ b \subseteq A \qquad D\ A\ c\ A'}{D\ A\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ A}$$

Correctness of D

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

  $\Longrightarrow$ We need a new, finer-grained semantics.

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

  $\Longrightarrow$ We need a new, finer-grained semantics.

- Big step semantics:
  semantics longer, correctness proof shorter

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

  $\Longrightarrow$ We need a new, finer-grained semantics.

- Big step semantics:
  semantics longer, correctness proof shorter

- Small step semantics:
  semantics shorter, correctness proof longer

Correctness of D

- Things can go wrong:
  execution may access uninitialized variable.

  $\Longrightarrow$ We need a new, finer-grained semantics.

- Big step semantics:
  semantics longer, correctness proof shorter

- Small step semantics:
  semantics shorter, correctness proof longer

For variety's sake, we choose a big step semantics.

11 Definite Initialization Analysis
Prelude: Variables in Expressions
Definite Initialization Analysis
Initialization Sensitive Semantics

$$state = vname \Rightarrow val\ option$$

$$state = vname \Rightarrow val\ option$$

where

datatype $'a\ option = None \mid Some\ 'a$

$$state = vname \Rightarrow val\ option$$

where

datatype $'a\ option = None \mid Some\ 'a$

Notation: $s(x \mapsto y)$ means $s(x := Some\ y)$

$$state = vname \Rightarrow val\ option$$

where

datatype $'a\ option = None \mid Some\ 'a$

Notation: $s(x \mapsto y)$ means $s(x := Some\ y)$

Definition: $dom\ s = \{a.\ s\ a \neq None\}$

Expression evaluation

$aval :: aexp \Rightarrow state \Rightarrow val\ option$

Expression evaluation

$aval :: aexp \Rightarrow state \Rightarrow val\ option$

$aval\ (N\ i)\ s = Some\ i$

Expression evaluation

$aval :: aexp \Rightarrow state \Rightarrow val\ option$

$aval\ (N\ i)\ s = Some\ i$

$aval\ (V\ x)\ s = s\ x$

Expression evaluation

$aval :: aexp \Rightarrow state \Rightarrow val\ option$

$aval\ (N\ i)\ s = Some\ i$

$aval\ (V\ x)\ s = s\ x$

$aval\ (Plus\ a_1\ a_2)\ s =$
$(\textsf{case}\ (aval\ a_1\ s,\ aval\ a_2\ s)\ \textsf{of}$
$\quad (Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1{+}i_2)$
$\ |\ \_ \Rightarrow None)$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval\ (Bc\ v)\ s = Some\ v$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval\ (Bc\ v)\ s = Some\ v$

$bval\ (Not\ b)\ s =$
$(\textsf{case}\ bval\ b\ s\ \textsf{of}\ None \Rightarrow None$
$\mid Some\ bv \Rightarrow Some\ (\neg\ bv))$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval\ (Bc\ v)\ s = Some\ v$

$bval\ (Not\ b)\ s =$
$(\textsf{case}\ bval\ b\ s\ \textsf{of}\ None \Rightarrow None$
$\ \mid Some\ bv \Rightarrow Some\ (\neg\ bv))$

$bval\ (And\ b_1\ b_2)\ s =$
$(\textsf{case}\ (bval\ b_1\ s,\ bval\ b_2\ s)\ \textsf{of}$
$\quad (Some\ bv_1,\ Some\ bv_2) \Rightarrow Some(bv_1 \wedge bv_2)$
$\ \mid \_ \Rightarrow None)$

$bval :: bexp \Rightarrow state \Rightarrow bool\ option$

$bval\ (Bc\ v)\ s = Some\ v$

$bval\ (Not\ b)\ s =$
$(\textsf{case}\ bval\ b\ s\ \textsf{of}\ None \Rightarrow None$
$\ \mid Some\ bv \Rightarrow Some\ (\neg\ bv))$

$bval\ (And\ b_1\ b_2)\ s =$
$(\textsf{case}\ (bval\ b_1\ s,\ bval\ b_2\ s)\ \textsf{of}$
$\quad (Some\ bv_1,\ Some\ bv_2) \Rightarrow Some(bv_1 \wedge bv_2)$
$\ \mid \_ \Rightarrow None)$

$bval\ (Less\ a_1\ a_2)\ s =$
$(\textsf{case}\ (aval\ a_1\ s,\ aval\ a_2\ s)\ \textsf{of}$
$\quad (Some\ i_1,\ Some\ i_2) \Rightarrow Some(i_1 < i_2)$
$\ \mid \_ \Rightarrow None)$

Big step semantics

$$(com,\ state) \Rightarrow state\ option$$

Big step semantics

$$(com,\ state) \Rightarrow state\ option$$

A small complication:

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

Big step semantics

$$(com,\ state) \Rightarrow state\ option$$

A small complication:

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

$$\frac{(c_1,\ s_1) \Rightarrow None}{(c_1;;\ c_2,\ s_1) \Rightarrow None}$$

Big step semantics

$$(com,\ state) \Rightarrow state\ option$$

A small complication:

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

$$\frac{(c_1,\ s_1) \Rightarrow None}{(c_1;;\ c_2,\ s_1) \Rightarrow None}$$

More convenient, because compositional:

$$(com,\ state\ option) \Rightarrow state\ option$$

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Execution starting in (mostly) normal states ($Some\ s$):

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

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Execution starting in (mostly) normal states ($Some\ s$):

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

$$\frac{aval\ a\ s = Some\ i}{(x ::= a,\ Some\ s) \Rightarrow Some\ (s(x \mapsto i))}$$

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Execution starting in (mostly) normal states ($Some\ s$):

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

$$\frac{aval\ a\ s = Some\ i}{(x ::= a,\ Some\ s) \Rightarrow Some\ (s(x \mapsto i))}$$

$$\frac{aval\ a\ s = None}{(x ::= a,\ Some\ s) \Rightarrow None}$$

Error ($None$) propagates:

$$(c,\ None) \Rightarrow None$$

Execution starting in (mostly) normal states ($Some\ s$):

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

$$\frac{aval\ a\ s = Some\ i}{(x ::= a,\ Some\ s) \Rightarrow Some\ (s(x \mapsto i))}$$

$$\frac{aval\ a\ s = None}{(x ::= a,\ Some\ s) \Rightarrow None}$$

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

$$\frac{bval\ b\ s = Some\ True \qquad (c_1,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c_1,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \qquad (c_2,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c_1,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \qquad (c_2,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = None}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow None}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{\begin{array}{c} bval\ b\ s = Some\ True \\ (c,\ Some\ s) \Rightarrow s' \qquad (WHILE\ b\ DO\ c,\ s') \Rightarrow s'' \end{array}}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{\begin{array}{c} bval\ b\ s = Some\ True \\ (c,\ Some\ s) \Rightarrow s' \qquad (WHILE\ b\ DO\ c,\ s') \Rightarrow s'' \end{array}}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = None}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow None}$$

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

$\textit{If } D\ (\textit{dom } s)\ c\ A' \textit{ and } (c, \textit{Some } s) \Rightarrow s'$
$\textit{then } s' \neq \textit{None}.$

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

$\textit{If } D\ (dom\ s)\ c\ A' \textit{ and } (c,\ Some\ s) \Rightarrow s'$
$\textit{then } s' \neq None.$

We need to prove a generalized statement:

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

$\textit{If } D\ (\textit{dom } s)\ c\ A' \textit{ and } (c, \textit{Some } s) \Rightarrow s'$
$\textit{then } s' \neq \textit{None}.$

We need to prove a generalized statement:

$\textit{If } (c, \textit{Some } s) \Rightarrow s' \textit{ and } D\ A\ c\ A' \textit{ and } A \subseteq \textit{dom } s$
$\textit{then } \exists\, t.\ s' = \textit{Some } t \wedge A' \subseteq \textit{dom } t.$

Correctness of D w.r.t. $\Rightarrow$

We want in the end:

Well-initialized programs cannot go wrong.

$\mathit{If}\ \ D\ (\mathit{dom}\ s)\ c\ A'\ \ \mathit{and}\ \ (c,\ \mathit{Some}\ s) \Rightarrow s'$
$\mathit{then}\ \ s' \neq \mathit{None}.$

We need to prove a generalized statement:

$\mathit{If}\ \ (c,\ \mathit{Some}\ s) \Rightarrow s'\ \ \mathit{and}\ \ D\ A\ c\ A'\ \ \mathit{and}\ \ A \subseteq \mathit{dom}\ s$
$\mathit{then}\ \ \exists\, t.\ s' = \mathit{Some}\ t \wedge A' \subseteq \mathit{dom}\ t.$

By rule induction on $(c,\ \mathit{Some}\ s) \Rightarrow s'$.

Proof needs some easy lemmas:

$$vars\ a \subseteq\ dom\ s \Longrightarrow \exists\, i.\ aval\ a\ s =\ Some\ i$$

$$vars\ b \subseteq\ dom\ s \Longrightarrow \exists\, bv.\ bval\ b\ s =\ Some\ bv$$

$$D\ A\ c\ A' \Longrightarrow\ A \subseteq\ A'$$

11 Definite Initialization Analysis

12 Live Variable Analysis

Motivation

Consider the following program:

```
    x := y + 1;
    y := y + 2;
    x := y + 3
```

Motivation

Consider the following program:

```
x := y + 1;
y := y + 2;
x := y + 3
```

Motivation

Consider the following program:

```
x := y + 1;
y := y + 2;
x := y + 3
```

The first assignment is redundant and can be removed

Motivation

Consider the following program:

```
x := y + 1;
y := y + 2;
x := y + 3
```

The first assignment is redundant and can be removed because x is dead at that point.

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

A weaker but easier to check condition:

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
x := 0
```

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
 x := 0
```

☹

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
x := 0
y := x; y := 0; x := 0
```

☹

Semantically, a variable x is live before command c
if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
x := 0                         ☹
y := x; y := 0; x := 0         ☺
```

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
x := 0                          ☹
y := x; y := 0; x := 0          ☺
WHILE b DO y := x; x := 1
```

Semantically, a variable x is live before command c if the initial value of x can influence the final state.

A weaker but easier to check condition:

We call x *live* before c
if there is some potential execution of c
where x is read before it can be overwritten.
Implicitly, every variable is read at the end of c.

Examples: Is `x` initially dead or live?

```
x := 0                               ☹
y := x; y := 0; x := 0               ☺
WHILE b DO y := x; x := 1            ☺
```

At the end of a command, we may be interested in the value of *only some of the variables*, At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

At the end of a command, we may be interested in the value of *only some of the variables*, e.g. *only the global variables* at the end of a procedure.

Then we say that x is live before c *relative to* the set of variables X.

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X\ =$ live before c relative to X

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X \ = \ \text{live before } c \text{ relative to } X$$

$L\ SKIP\ X \qquad =$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X\ =$ live before c relative to X

$$L\ SKIP\ X \quad = \quad X$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

$$\begin{aligned} L\ SKIP\ X &= X \\ L\ (x ::= a)\ X &= \end{aligned}$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X\ =$ live before c relative to X

$$
\begin{array}{lcl}
L\ SKIP\ X & = & X \\
L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\})
\end{array}
$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X \ = \ \text{live before } c \text{ relative to } X$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & \end{array}$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X \ = \ \text{live before } c \text{ relative to } X$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \end{array}$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X$ = live before c relative to X

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \\ L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ = & & \end{array}$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

$$\begin{aligned}
L\ SKIP\ X &= X \\
L\ (x ::= a)\ X &= vars\ a \cup (X - \{x\}) \\
L\ (c_1;;\ c_2)\ X &= L\ c_1\ (L\ c_2\ X) \\
L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X &= \\
&\quad vars\ b \cup L\ c_1\ X \cup L\ c_2\ X
\end{aligned}$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \\ \multicolumn{3}{l}{L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =} \\ \multicolumn{3}{l}{\quad vars\ b \cup L\ c_1\ X \cup L\ c_2\ X} \end{array}$$

Example:

$$L\ (''y'' ::= V\ ''z'';;\ ''x'' ::= Plus\ (V\ ''y'')\ (V\ ''z''))$$
$$\{''x''\}\ =$$

Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

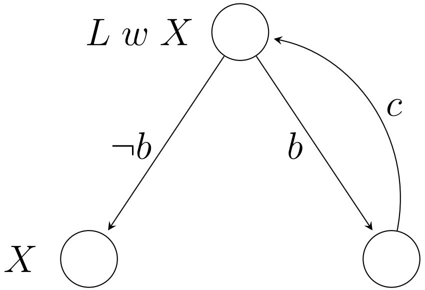
$$L\ c\ X\ =\ \text{live before } c \text{ relative to } X$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \\ \multicolumn{3}{l}{L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =} \\ \multicolumn{3}{l}{\quad vars\ b \cup L\ c_1\ X \cup L\ c_2\ X} \end{array}$$

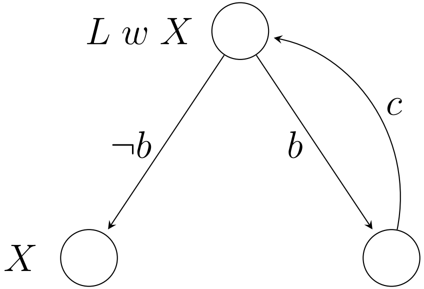
Example:

$$L\ (''y'' ::= V\ ''z'';;\ ''x'' ::= Plus\ (V\ ''y'')\ (V\ ''z''))$$
$$\{''x''\}\ =\ \{''z''\}$$

WHILE b DO c

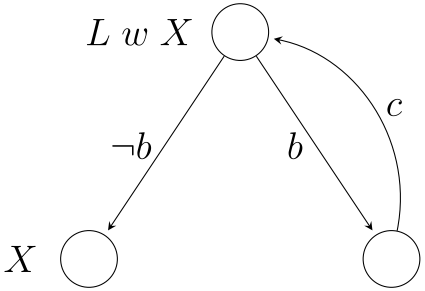


$WHILE\ b\ DO\ c$



$L\ w\ X$ must satisfy

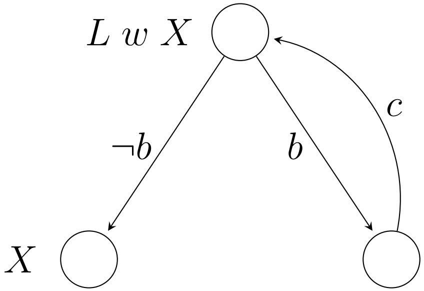
$WHILE\ b\ DO\ c$



$L\ w\ X$ must satisfy

$vars\ b \subseteq L\ w\ X$ (evaluation of b)

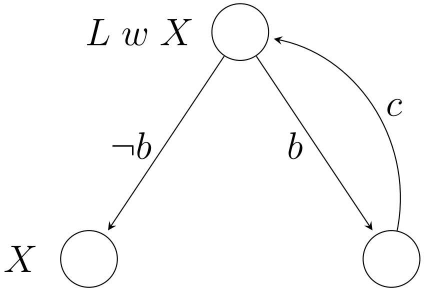
$WHILE\ b\ DO\ c$



$L\ w\ X$ must satisfy

$vars\ b$	$\subseteq$	$L\ w\ X$	(evaluation of b)
X	$\subseteq$	$L\ w\ X$	(exit)

$WHILE\ b\ DO\ c$



$L\ w\ X$ must satisfy

$vars\ b$	$\subseteq$	$L\ w\ X$	(evaluation of b)
X	$\subseteq$	$L\ w\ X$	(exit)
$L\ c\ (L\ w\ X)$	$\subseteq$	$L\ w\ X$	(execution of c)

We define

$$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$$

We define

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

$\Longrightarrow$

$vars\ b \subseteq L\ w\ X$ ✓

$X \subseteq L\ w\ X$ ✓

We define

$$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$$

$\Longrightarrow$

$vars\ b \subseteq L\ w\ X$	✓
$X \subseteq L\ w\ X$	✓
$L\ c\ (L\ w\ X) \subseteq L\ w\ X$	?

$$\begin{array}{lcl} L\ \mathit{SKIP}\ X & = & X \\ L\ (x ::= a)\ X & = & \mathit{vars}\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \end{array}$$

$$L\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ X\ =\ \mathit{vars}\ b \cup L\ c_1\ X \cup L\ c_2\ X$$

$$L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X\ =\ \mathit{vars}\ b \cup X \cup L\ c\ X$$

Example:

$$L\ (\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (V\ ''x'')\ \mathit{DO}\ ''y'' ::= V\ ''z'') \\ \{''x''\}\ =$$

$$\begin{array}{lcl} L\ SKIP\ X & = & X \\ L\ (x ::= a)\ X & = & vars\ a \cup (X - \{x\}) \\ L\ (c_1;;\ c_2)\ X & = & L\ c_1\ (L\ c_2\ X) \end{array}$$

$$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =\ vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$$

$$L\ (WHILE\ b\ DO\ c)\ X\ =\ vars\ b \cup X \cup L\ c\ X$$

Example:

$$L\ (WHILE\ Less\ (V\ ''x'')\ (V\ ''x'')\ DO\ ''y'' ::= V\ ''z'')$$
$$\{''x''\}\ =\ \{''x'', ''z''\}$$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = X - kill\ c \cup gen\ c$$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = X - kill\ c \cup gen\ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \Longrightarrow A\ c\ X_1 \subseteq A\ c\ X_2$$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = X - kill\ c \cup gen\ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \Longrightarrow A\ c\ X_1 \subseteq A\ c\ X_2$$
$$A\ c\ (X_1 \cap X_2) = A\ c\ X_1 \cap A\ c\ X_2$$

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = X - kill\ c \cup gen\ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \Longrightarrow A\ c\ X_1 \subseteq A\ c\ X_2$$
$$A\ c\ (X_1 \cap X_2) = A\ c\ X_1 \cap A\ c\ X_2$$

Many standard data-flow analyses are gen/kill.

Gen/kill analyses

A data-flow analysis $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$
is called gen/kill analysis
if there are functions gen and $kill$ such that

$$A\ c\ X = (X - kill\ c) \cup gen\ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \Longrightarrow A\ c\ X_1 \subseteq A\ c\ X_2$$
$$A\ c\ (X_1 \cap X_2) = A\ c\ X_1 \cap A\ c\ X_2$$

Many standard data-flow analyses are gen/kill.
In particular liveness analysis.

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$kill\ SKIP \qquad\qquad\qquad =$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$kill\ SKIP \quad = \quad \{\}$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} kill\ SKIP & = & \{\} \\ kill\ (x ::= a) & = & \end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} kill\ SKIP & = & \{\} \\ kill\ (x ::= a) & = & \{x\} \end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} kill\ SKIP & = & \{\} \\ kill\ (x ::= a) & = & \{x\} \\ kill\ (c_1;;\ c_2) & = & \end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
kill\ SKIP & = & \{\} \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;;\ c_2) & = & kill\ c_1 \cup kill\ c_2
\end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
kill\ SKIP & = & \{\} \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;;\ c_2) & = & kill\ c_1 \cup kill\ c_2 \\
kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = &
\end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
kill\ SKIP & = & \{\} \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;;\ c_2) & = & kill\ c_1 \cup kill\ c_2 \\
kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & kill\ c_1 \cap kill\ c_2
\end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
kill\ SKIP & = & \{\} \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;;\ c_2) & = & kill\ c_1 \cup kill\ c_2 \\
kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & kill\ c_1 \cap kill\ c_2 \\
kill\ (WHILE\ b\ DO\ c) & = &
\end{array}$$

Liveness via gen/kill

$kill :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
kill\ SKIP & = & \{\} \\
kill\ (x ::= a) & = & \{x\} \\
kill\ (c_1;;\ c_2) & = & kill\ c_1 \cup kill\ c_2 \\
kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & kill\ c_1 \cap kill\ c_2 \\
kill\ (WHILE\ b\ DO\ c) & = & \{\}
\end{array}$$

$gen :: com \Rightarrow vname\ set$

$gen :: com \Rightarrow vname\ set$

$gen\ SKIP \qquad =$

$$gen :: com \Rightarrow vname\ set$$

$$gen\ SKIP \quad = \quad \{\}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} gen\ SKIP & = & \{\} \\ gen\ (x ::= a) & = & \end{array}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} gen\ SKIP & = & \{\} \\ gen\ (x ::= a) & = & vars\ a \end{array}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl} gen\ SKIP & = & \{\} \\ gen\ (x ::= a) & = & vars\ a \\ gen\ (c_1;;\ c_2) & = & \end{array}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1)
\end{array}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1) \\
\multicolumn{3}{l}{gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ =}
\end{array}$$

$gen :: com \Rightarrow vname\ set$

$$\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1)
\end{array}$$

$$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$
$$\quad vars\ b \cup gen\ c_1 \cup gen\ c_2$$

$gen :: com \Rightarrow vname\ set$

$$
\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1)
\end{array}
$$

$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ =$

$\qquad vars\ b \cup gen\ c_1 \cup gen\ c_2$

$gen\ (WHILE\ b\ DO\ c)\ =$

$$gen :: com \Rightarrow vname\ set$$

$$
\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & vars\ a \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup (gen\ c_2 - kill\ c_1)
\end{array}
$$

$$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$$
$$\quad vars\ b \cup gen\ c_1 \cup gen\ c_2$$
$$gen\ (WHILE\ b\ DO\ c) = vars\ b \cup gen\ c$$

$$L\ c\ X = gen\ c \cup (X - kill\ c)$$

$$L\ c\ X = gen\ c \cup (X - kill\ c)$$

Proof by induction on c.

$$L\ c\ X = gen\ c \cup (X - kill\ c)$$

Proof by induction on c.

$\Longrightarrow$

$$L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$\mathit{gen}\ \mathit{SKIP} \qquad =$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\mathit{gen}\ \mathit{SKIP} = \{\}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{aligned}
\mathit{gen}\ \mathit{SKIP} &= \{\} \\
\mathit{gen}\ (x ::= a) &=
\end{aligned}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{array}{lcl} \mathit{gen}\ \mathit{SKIP} & = & \{\} \\ \mathit{gen}\ (x ::= a) & = & \{x\} \end{array}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{aligned}
\mathit{gen}\ \mathit{SKIP} &= \{\} \\
\mathit{gen}\ (x ::= a) &= \{x\} \\
\mathit{gen}\ (c_1;;\ c_2) &=
\end{aligned}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{array}{lcl} \mathit{gen}\ \mathit{SKIP} & = & \{\} \\ \mathit{gen}\ (x ::= a) & = & \{x\} \\ \mathit{gen}\ (c_1;;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \end{array}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$
\begin{array}{lcl}
\mathit{gen}\ \mathit{SKIP} & = & \{\} \\
\mathit{gen}\ (x ::= a) & = & \{x\} \\
\mathit{gen}\ (c_1;;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2) & = &
\end{array}
$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - kill\ c \cup gen\ c$:

$$\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & \{x\} \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup gen\ c_2 \\
gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & gen\ c_1 \cap gen\ c_2
\end{array}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{array}{lcl}
\mathit{gen}\ \mathit{SKIP} & = & \{\} \\
\mathit{gen}\ (x ::= a) & = & \{x\} \\
\mathit{gen}\ (c_1;;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2) & = & \mathit{gen}\ c_1 \cap \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c) & = &
\end{array}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$\begin{array}{lcl}
\mathit{gen}\ \mathit{SKIP} & = & \{\} \\
\mathit{gen}\ (x ::= a) & = & \{x\} \\
\mathit{gen}\ (c_1;;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2) & = & \mathit{gen}\ c_1 \cap \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c) & = & \{\}
\end{array}$$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c
if X was initialized before c

How to obtain $A\ c\ X = X - \mathit{kill}\ c \cup \mathit{gen}\ c$:

$$
\begin{array}{lcl}
\mathit{gen}\ \mathit{SKIP} & = & \{\} \\
\mathit{gen}\ (x ::= a) & = & \{x\} \\
\mathit{gen}\ (c_1;;\ c_2) & = & \mathit{gen}\ c_1 \cup \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2) & = & \mathit{gen}\ c_1 \cap \mathit{gen}\ c_2 \\
\mathit{gen}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c) & = & \{\}
\end{array}
$$

$\mathit{kill}\ c\ =$

Digression: definite initialization via gen/kill

$A\ c\ X$: the set of variables initialized after c if X was initialized before c

How to obtain $A\ c\ X = X - kill\ c \cup gen\ c$:

$$\begin{array}{lcl}
gen\ SKIP & = & \{\} \\
gen\ (x ::= a) & = & \{x\} \\
gen\ (c_1;;\ c_2) & = & gen\ c_1 \cup gen\ c_2 \\
gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) & = & gen\ c_1 \cap gen\ c_2 \\
gen\ (WHILE\ b\ DO\ c) & = & \{\}
\end{array}$$

$$kill\ c = \{\}$$

12 Live Variable Analysis
Correctness of L
Dead Variable Elimination
True Liveness
Comparisons

$(.,.) \Rightarrow .$ and L should roughly be related like this:

The value of the final state on X
only depends on
the value of the initial state on $L\ c\ X$.

$(.,.) \Rightarrow .$ and L should roughly be related like this:

The value of the final state on X
only depends on
the value of the initial state on $L \; c \; X$*.*

Put differently:

If two initial states agree on $L \; c \; X$
then the corresponding final states agree on X*.*

Equality on

An abbreviation:

$$f = g \ on \ X \quad \equiv \quad \forall x \in X.\ f\,x = g\ x$$

Equality on

An abbreviation:

$$f = g\ on\ X \quad \equiv \quad \forall\, x \in X.\ f\, x = g\ x$$

Two easy theorems (in theory $Vars$):

$$s_1 = s_2\ on\ vars\ a \Longrightarrow aval\ a\ s_1 = aval\ a\ s_2$$

Equality on

An abbreviation:

$$f = g \ on \ X \quad \equiv \quad \forall\, x \in X.\ f\,x = g\ x$$

Two easy theorems (in theory $Vars$):

$$s_1 = s_2 \ on \ vars \ a \Longrightarrow aval \ a \ s_1 = aval \ a \ s_2$$
$$s_1 = s_2 \ on \ vars \ b \Longrightarrow bval \ b \ s_1 = bval \ b \ s_2$$

Correctness of L

If $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
then $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Correctness of L

If $(c, s) \Rightarrow s'$ *and* $s = t \ on \ L \ c \ X$
then $\exists\, t'.\ (c, t) \Rightarrow t' \wedge s' = t' \ on \ X$.

Proof by rule induction.

Correctness of L

If $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
then $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof by rule induction.
For the two $WHILE$ cases we do not need the definition of $L\ w$ but only the characteristic property

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

Optimality of $L\ w$

The result of L should be as small as possible: the more dead variables, the better

Optimality of $L\ w$

The result of L should be as small as possible: the more dead variables, the better (for program optimization).

Optimality of $L\ w$

The result of L should be as small as possible: the more dead variables, the better (for program optimization).

> *$L\ w\ X$ should be the least set such that*
> *$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$.*

Optimality of $L\ w$

The result of L should be as small as possible: the more dead variables, the better (for program optimization).

> *$L\ w\ X$ should be the least set such that*
> *$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$.*

Follows easily from $L\ c\ X = gen\ c \cup (X - kill\ c)$:

$$vars\ b \cup X \cup L\ c\ P \subseteq P \Longrightarrow$$
$$L\ (WHILE\ b\ DO\ c)\ X \subseteq P$$

12 Live Variable Analysis

Correctness of L
Dead Variable Elimination
True Liveness
Comparisons

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$bury\ SKIP\ X \qquad =$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$bury\ SKIP\ X \quad = \quad SKIP$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl} bury\ SKIP\ X & = & SKIP \\ bury\ (x ::= a)\ X & = & \end{array}$$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl} bury\ SKIP\ X & = & SKIP \\ bury\ (x ::= a)\ X & = & \textit{if } x \in X \textit{ then } x ::= a \textit{ else } SKIP \end{array}$$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl} bury\ SKIP\ X & = & SKIP \\ bury\ (x ::= a)\ X & = & \textit{if}\ x \in X\ \textit{then}\ x ::= a\ \textit{else}\ SKIP \\ bury\ (c_1;;\ c_2)\ X & = & \end{array}$$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl} bury\ SKIP\ X & = & SKIP \\ bury\ (x ::= a)\ X & = & \textit{if}\ x \in X\ \textit{then}\ x ::= a\ \textit{else}\ SKIP \\ bury\ (c_1;;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X \end{array}$$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl}
bury\ SKIP\ X & = & SKIP \\
bury\ (x ::= a)\ X & = & \textsf{\textit{if}}\ x \in X\ \textsf{\textit{then}}\ x ::= a\ \textsf{\textit{else}}\ SKIP \\
bury\ (c_1;;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X
\end{array}$$

$bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl}
bury\ SKIP\ X & = & SKIP \\
bury\ (x ::= a)\ X & = & \textit{if}\ x \in X\ \textit{then}\ x ::= a\ \textit{else}\ SKIP \\
bury\ (c_1;;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X
\end{array}$$

$$bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =$$
$$\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X$$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$
\begin{array}{lcl}
bury\ SKIP\ X & = & SKIP \\
bury\ (x ::= a)\ X & = & \textsf{\textit{if}}\ x \in X\ \textsf{\textit{then}}\ x ::= a\ \textsf{\textit{else}}\ SKIP \\
bury\ (c_1;;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X
\end{array}
$$

$bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X\ =$
$\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X$
$bury\ (WHILE\ b\ DO\ c)\ X\ =$

Bury all assignments to dead variables:

$bury :: com \Rightarrow vname\ set \Rightarrow com$

$$\begin{array}{lcl}
bury\ SKIP\ X & = & SKIP \\
bury\ (x ::= a)\ X & = & \textit{if } x \in X \textit{ then } x ::= a \textit{ else } SKIP \\
bury\ (c_1;;\ c_2)\ X & = & bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X
\end{array}$$

$bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$
$\quad IF\ b\ THEN\ bury\ c_1\ X\ ELSE\ bury\ c_2\ X$

$bury\ (WHILE\ b\ DO\ c)\ X =$
$\quad WHILE\ b\ DO\ bury\ c\ (L\ (WHILE\ b\ DO\ c)\ X)$

Correctness of $bury$

Correctness of $bury$

$$bury\ c\ UNIV \sim c$$

where $UNIV$ is the set of all variables.

Correctness of $bury$

$$bury\ c\ UNIV \sim c$$

where $UNIV$ is the set of all variables.

The two directions need to be proved separately.

$$(c,\ s) \Rightarrow s' \Longrightarrow (bury\ c\ UNIV,\ s) \Rightarrow s'$$

$$(c,\ s) \Rightarrow s' \Longrightarrow (bury\ c\ UNIV,\ s) \Rightarrow s'$$

Follows from generalized statement:

> *If* $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
> *then* $\exists\, t'.\ (bury\ c\ X,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

$$(c,\ s) \Rightarrow s' \Longrightarrow (bury\ c\ UNIV,\ s) \Rightarrow s'$$

Follows from generalized statement:

> *If* $(c,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
> *then* $\exists\, t'.\ (bury\ c\ X,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof by rule induction, like for correctness of L.

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \Longrightarrow (c,\ s) \Rightarrow s'$$

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \Longrightarrow (c,\ s) \Rightarrow s'$$

Follows from generalized statement:

> *If* $(bury\ c\ X,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
> *then* $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \Longrightarrow (c,\ s) \Rightarrow s'$$

Follows from generalized statement:

> *If* $(bury\ c\ X,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
> *then* $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof very similar to other direction, but needs inversion lemmas for $bury$ for every kind of command,

$$(bury\ c\ UNIV,\ s) \Rightarrow s' \Longrightarrow (c,\ s) \Rightarrow s'$$

Follows from generalized statement:

> *If* $(bury\ c\ X,\ s) \Rightarrow s'$ *and* $s = t\ on\ L\ c\ X$
> *then* $\exists\, t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$.

Proof very similar to other direction, but needs inversion lemmas for $bury$ for every kind of command, e.g.

$$\begin{aligned}
&(bc_1;;\ bc_2 = bury\ c\ X) = \\
&(\exists\, c_1\ c_2. \\
&\quad c = c_1;;\ c_2 \wedge \\
&\quad bc_2 = bury\ c_2\ X \wedge bc_1 = bury\ c_1\ (L\ c_2\ X))
\end{aligned}$$

12 Live Variable Analysis

- Correctness of L
- Dead Variable Elimination
- True Liveness
- Comparisons

Terminology

Let $f :: \tau \Rightarrow \tau$ and $x :: \tau$.

Terminology

Let $f :: \tau \Rightarrow \tau$ and $x :: \tau$.

If $f\ x = x$ then x is a *fixpoint* of f.

Terminology

Let $f :: \tau \Rightarrow \tau$ and $x :: \tau$.

If $f\ x = x$ then x is a *fixpoint* of f.

Let $\leq$ be a partial order on τ, eg $\subseteq$ on sets.

Terminology

Let $f :: \tau \Rightarrow \tau$ and $x :: \tau$.

If $f\ x = x$ then x is a *fixpoint* of f.

Let $\leq$ be a partial order on τ, eg $\subseteq$ on sets.

If $f\ x \leq x$ then x is a *pre-fixpoint* (*pfp*) of f.

Terminology

Let $f :: \tau \Rightarrow \tau$ and $x :: \tau$.

If $f\ x = x$ then x is a *fixpoint* of f.

Let $\leq$ be a partial order on τ, eg $\subseteq$ on sets.

If $f\ x \leq x$ then x is a *pre-fixpoint* (*pfp*) of f.

If $x \leq y \Longrightarrow f\ x \leq f\ y$ for all x,y, then f is *monotone*.

Application to $L\ w$

Remember the specification of $L\ w$:

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

Application to $L\ w$

Remember the specification of $L\ w$:

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

This is the same as saying that $L\ w\ X$ should be a pfp of

$$\lambda P.\ vars\ b \cup X \cup L\ c\ P$$

Application to $L\ w$

Remember the specification of $L\ w$:

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X$$

This is the same as saying that $L\ w\ X$ should be a pfp of

$$\lambda P.\ vars\ b \cup X \cup L\ c\ P$$

and in particular of $L\ c$.

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

But $''y''$ is not truly live: it is assigned to a dead variable.

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

But $''y''$ is not truly live: it is assigned to a dead variable.

Problem: $L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

But $''y''$ is not truly live: it is assigned to a dead variable.

Problem: $L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

Better:

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

True liveness

$L\ (''x'' ::= V\ ''y'')\ \{\} = \{''y''\}$

But $''y''$ is not truly live: it is assigned to a dead variable.

Problem: $L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

Better:

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

But then

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

is not correct anymore.

$$L\ (x ::= a)\ X =$$
$$(\textsf{\textit{if}}\ x \in X\ \textsf{\textit{then}}\ vars\ a \cup (X - \{x\})\ \textsf{\textit{else}}\ X)$$

$$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$$

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ =\ Less\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ =\ Less\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$L\ (x ::= a)\ X =$
$(\textit{if } x \in X \textit{ then } vars\ a \cup (X - \{x\}) \textit{ else } X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ = \ Less\ (N\ 0)\ (V\ y)$
and $c\ = \ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$$y ::= V\ x \qquad\qquad x ::= V\ z\ \ \{y\}$$

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ = \ Less\ (N\ 0)\ (V\ y)$
and $c\ = \ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$$y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$$

$L\ (x ::= a)\ X =$
$(\textit{if } x \in X \textit{ then } vars\ a \cup (X - \{x\}) \textit{ else } X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ =\ Less\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$

Let $w = WHILE\ b\ DO\ c$
where $b\ = \ Less\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$
and $distinct\ [x,\ y,\ z]$

Then $L\ w\ \{y\} = \{x,\ y\}$, but z is live before w !

$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$

$\Longrightarrow\ L\ w\ \{y\} = \{y\} \cup \{y\} \cup \{x\}$

$$\begin{aligned} b &= Less\ (N\ 0)\ (V\ y) \\ c &= y ::= V\ x;;\ x ::= V\ z \end{aligned}$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$$b \;=\; Less\ (N\ 0)\ (V\ y)$$
$$c \;=\; y ::= V\ x;;\ x ::= V\ z$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$$y ::= V\ x \qquad\qquad x ::= V\ z\ \ \{x,\ y\}$$

$$\begin{aligned} b &= Less\ (N\ 0)\ (V\ y) \\ c &= y ::= V\ x;;\ x ::= V\ z \end{aligned}$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$$y ::= V\ x\ \ \{y,\ z\}\ x ::= V\ z\ \ \{x,\ y\}$$

$$\begin{aligned} b &= Less\ (N\ 0)\ (V\ y) \\ c &= y ::= V\ x;;\ x ::= V\ z \end{aligned}$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ x ::= V\ z\ \ \{x,\ y\}$

$$\begin{aligned} b &= Less\ (N\ 0)\ (V\ y) \\ c &= y ::= V\ x;;\ x ::= V\ z \end{aligned}$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ x ::= V\ z\ \ \{x,\ y\}$

$L\ c\ \{x,\ y\} = \{x,\ z\}$

$$b \;=\; Less\ (N\ 0)\ (V\ y)$$
$$c \;=\; y ::= V\ x;;\ x ::= V\ z$$

$L\ w\ \{y\} = \{x,\ y\}$ is not a pfp of $L\ c$:

$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ x ::= V\ z\ \ \{x,\ y\}$

$L\ c\ \{x,\ y\} = \{x,\ z\} \not\subseteq \{x,\ y\}$

$L\ w$ for true liveness

Define $L\ w\ X$ *as the least pfp of*
$\lambda P.\ vars\ b \cup X \cup L\ c\ P$

Existence of least fixpoints

Theorem (Knaster-Tarski) Let $f :: \tau\ set \Rightarrow \tau\ set$.

Existence of least fixpoints

Theorem (Knaster-Tarski) Let $f :: \tau\ set \Rightarrow \tau\ set$.
If f is monotone ($X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$)

Existence of least fixpoints

Theorem (Knaster-Tarski) Let $f :: \tau\ set \Rightarrow \tau\ set$.
If f is monotone ($X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$)
then

$$lfp(f) := \bigcap\{P \mid f(P) \subseteq P\}$$

is the least pre-fixpoint and least fixpoint of f.

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof • $f\ (\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof
- $f\ (\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$
- $\mathit{lfp}\ f$ is the least pre-fixpoint of f

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof
- $f\ (\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$
- $\mathit{lfp}\ f$ is the least pre-fixpoint of f

Lemma Let f be a monotone function on a partial order $\leq$. Then a least pre-fixpoint of f is also a least fixpoint.

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $lfp(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof
- $f\ (lfp\ f) \subseteq lfp\ f$
- $lfp\ f$ is the least pre-fixpoint of f

Lemma Let f be a monotone function on a partial order $\leq$. Then a least pre-fixpoint of f is also a least fixpoint.

Proof
- $f\ p \leq p \Longrightarrow f\ p = p$

Proof of Knaster-Tarski

Theorem If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\mathit{lfp}(f) := \bigcap\{P \mid f(P) \subseteq P\}$ is the least pre-fixpoint.

Proof
- $f\ (\mathit{lfp}\ f) \subseteq \mathit{lfp}\ f$
- $\mathit{lfp}\ f$ is the least pre-fixpoint of f

Lemma Let f be a monotone function on a partial order $\leq$. Then a least pre-fixpoint of f is also a least fixpoint.

Proof
- $f\ p \leq p \Longrightarrow f\ p = p$
- p is the least fixpoint

Definition of L

$L\ (x ::= a)\ X =$
$(\textit{if } x \in X \textit{ then } vars\ a \cup (X - \{x\}) \textit{ else } X)$

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
where $f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

Definition of L

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
where $f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

Lemma $L\ c$ is monotone.

Definition of L

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
where $f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

Lemma $L\ c$ is monotone.

Proof by induction on c using that lfp is monotone:
$lfp\ f \subseteq lfp\ g$ if for all X, $f\ X \subseteq g\ X$

Definition of L

$L\ (x ::= a)\ X =$
$(\textit{if}\ x \in X\ \textit{then}\ vars\ a \cup (X - \{x\})\ \textit{else}\ X)$

$L\ (WHILE\ b\ DO\ c)\ X = lfp\ f_w$
where $f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

Lemma $L\ c$ is monotone.

Proof by induction on c using that lfp is monotone:
$lfp\ f \subseteq lfp\ g$ if for all X, $f\ X \subseteq g\ X$

Corollary f_w is monotone.

Computation of lfp

Theorem Let $f :: \tau\ set \Rightarrow \tau\ set$.

Computation of lfp

Theorem Let $f :: \tau\ set \Rightarrow \tau\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$

Computation of lfp

Theorem Let $f :: \tau\ set \Rightarrow \tau\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and the chain $\{\} \subseteq f(\{\}) \subseteq f(f(\{\})) \subseteq \dots$
  stabilizes after a finite number of steps,
  i.e. $f^{k+1}(\{\}) = f^k(\{\})$ for some k,

Computation of lfp

Theorem Let $f :: \tau\ set \Rightarrow \tau\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and the chain $\{\} \subseteq f(\{\}) \subseteq f(f(\{\})) \subseteq \dots$ stabilizes after a finite number of steps, i.e. $f^{k+1}(\{\}) = f^k(\{\})$ for some k,

then $\mathit{lfp}(f) = f^k(\{\})$.

Computation of lfp

Theorem Let $f :: \tau\ set \Rightarrow \tau\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and the chain $\{\} \subseteq f(\{\}) \subseteq f(f(\{\})) \subseteq \ldots$ stabilizes after a finite number of steps, i.e. $f^{k+1}(\{\}) = f^k(\{\})$ for some k,

then $\mathit{lfp}(f) = f^k(\{\})$.

Proof Show $f^i(\{\}) \subseteq p$ for any pfp p of f (by induction on i).

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

Computation of $lfp\ f_w$

$f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq$

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Computation of $lfp\ f_w$

$f_w = (\lambda P.\ vars\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $vars\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq vars\ c \cup X$

Proof by induction on c

Let $V_w = vars\ b \cup vars\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Hence $f_w^k\ \{\}$ stabilizes for some $k \leq$

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Hence $f_w^k\ \{\}$ stabilizes for some $k \leq |V_w|$.

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Hence $f_w^k\ \{\}$ stabilizes for some $k \leq |V_w|$.
More precisely: $k \leq |\mathit{vars}\ c| + 1$

Computation of $\mathit{lfp}\ f_w$

$f_w = (\lambda P.\ \mathit{vars}\ b \cup X \cup L\ c\ P)$

The chain $\{\} \subseteq f_w\ \{\} \subseteq f_w^2\ \{\} \subseteq \ldots$ must stabilize:

Let $\mathit{vars}\ c$ be the variables in c.

Lemma $L\ c\ X \subseteq \mathit{vars}\ c \cup X$

Proof by induction on c

Let $V_w = \mathit{vars}\ b \cup \mathit{vars}\ c \cup X$

Corollary $P \subseteq V_w \Longrightarrow f_w\ P \subseteq V_w$

Hence $f_w^k\ \{\}$ stabilizes for some $k \leq |V_w|$.
More precisely: $k \leq |\mathit{vars}\ c| + 1$
because $f_w\ \{\} \supseteq \mathit{vars}\ b \cup X$.

Example

Let $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\ 0)\ (V\ y)$
and $c \ = \ y ::= V\ x;;\ x ::= V\ z$

Example

Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
where $b\ =\ \mathit{Less}\ (N\, 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

Example

Let $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\,0)\ (V\ y)$
and $c \ = \ y ::= \ V\ x;;\ x ::= \ V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

Example

Let $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\ 0)\ (V\ y)$
and $c \ = \ y ::= V\ x;;\ x ::= V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$$\{\}\ \ y ::= V\ x\ \ \{\}\ \ x ::= V\ z\ \ \{\}$$

Example

Let $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\ 0)\ (V\ y)$
and $c \ = \ y ::= \ V\ x;;\ x ::= \ V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$$\{\} \quad y ::= \ V\ x \quad \{\} \quad x ::= \ V\ z \quad \{\}$$

$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x,\ y\}$:

Example

Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
where $b\ =\ \mathit{Less}\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$$\{\}\ \ y ::= V\ x\ \ \{\}\ \ x ::= V\ z\ \ \{\}$$

$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x,\ y\}$:

$$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$$

Example

Let $w = \ WHILE\ b\ DO\ c$
where $b \ = \ Less\ (N\ 0)\ (V\ y)$
and $c \ = \ y ::= \ V\ x;;\ x ::= \ V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$\{\} \quad y ::= \ V\ x \quad \{\} \quad x ::= \ V\ z \quad \{\}$

$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x,\ y\}$:

$\{x\} \quad y ::= \ V\ x \quad \{y\} \quad x ::= \ V\ z \quad \{y\}$

$f_w\ \{x,\ y\} = \{y\} \cup L\ c\ \{x,y\} = \{x,\ y,\ z\}$:

Example

Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$
where $b\ =\ \mathit{Less}\ (N\ 0)\ (V\ y)$
and $c\ =\ y ::= V\ x;;\ x ::= V\ z$

To compute $L\ w\ \{y\}$ we iterate $f_w\ P = \{y\} \cup L\ c\ P$:

$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}$:

$\{\}\ \ y ::= V\ x\ \ \{\}\ \ x ::= V\ z\ \ \{\}$

$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x,\ y\}$:

$\{x\}\ \ y ::= V\ x\ \ \{y\}\ \ x ::= V\ z\ \ \{y\}$

$f_w\ \{x,\ y\} = \{y\} \cup L\ c\ \{x{,}y\} = \{x,\ y,\ z\}$:

$\{x,\ z\}\ \ y ::= V\ x\ \ \{y,\ z\}\ \ x ::= V\ z\ \ \{x,\ y\}$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (\textsf{if}\ b\ s\ \textsf{then}\ while\ b\ f\ (f\ s)\ \textsf{else}\ s)$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (\textit{if}\ b\ s\ \textit{then}\ while\ b\ f\ (f\ s)\ \textit{else}\ s)$

Lemma Let $f :: \tau\ set \Rightarrow \tau\ set$.

Computation of *lfp* in Isabelle

From the library theory `While_Combinator`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = ($*if* $b\ s$ ***then*** $while\ b\ f\ (f\ s)$ ***else*** $s)$

Lemma Let $f :: \tau\ set \Rightarrow \tau\ set$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$\mathit{while} :: ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$\mathit{while}\ b\ f\ s = (\textsf{\textit{if}}\ b\ s\ \textsf{\textit{then}}\ \mathit{while}\ b\ f\ (f\ s)\ \textsf{\textit{else}}\ s)$

Lemma Let $f :: \tau\ \mathit{set} \Rightarrow \tau\ \mathit{set}$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and bounded by some finite set C:
  $X \subseteq C \Longrightarrow f\ X \subseteq C$

Computation of lfp in Isabelle

From the library theory `While_Combinator`:

$\mathit{while} :: ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$\mathit{while}\ b\ f\ s = (\textsf{\textit{if}}\ b\ s\ \textsf{\textit{then}}\ \mathit{while}\ b\ f\ (f\ s)\ \textsf{\textit{else}}\ s)$

Lemma Let $f :: \tau\ \mathit{set} \Rightarrow \tau\ \mathit{set}$. If

- f is monotone: $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and bounded by some finite set C:
  $X \subseteq C \Longrightarrow f\ X \subseteq C$

then $\mathit{lfp}\ f = \mathit{while}\ (\lambda X.\ f\ X \neq X)\ f\ \{\}$

Limiting the number of iterations

Limiting the number of iterations

Fix some small k (eg 2)

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$:

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH)

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P \Longrightarrow f_w(g_w^i\ \{\}) = g_w\ (g_w^i\ \{\}) = g_w^i\ \{\}$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P \Longrightarrow f_w(g_w^i\ \{\}) = g_w\ (g_w^i\ \{\}) = g_w^i\ \{\}$
$\Longrightarrow L\ w\ X = lfp\ f_w \subseteq g_w^i\ \{\} = Lb\ w\ X$

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P \Longrightarrow f_w(g_w^i\ \{\}) = g_w\ (g_w^i\ \{\}) = g_w^i\ \{\}$
$\Longrightarrow L\ w\ X = lfp\ f_w \subseteq g_w^i\ \{\} = Lb\ w\ X$

If $Lb\ w\ X = V_w$:

Limiting the number of iterations

Fix some small k (eg 2) and define Lb like L except

$$Lb\ w\ X = \begin{cases} g_w^i\ \{\} & \text{if } g_w^{i+1}\ \{\} = g_w^i\ \{\} \text{ for some } i < k \\ V_w & \text{otherwise} \end{cases}$$

where $g_w\ P = vars\ b \cup X \cup Lb\ c\ P$

Theorem $L\ c\ X \subseteq Lb\ c\ X$

Proof by induction on c. In the $WHILE$ case:

If $Lb\ w\ X = g_w^i\ \{\}$: $\forall P.\ L\ c\ P \subseteq Lb\ c\ P$ (IH) $\Longrightarrow$
$\forall P.\ f_w\ P \subseteq g_w\ P \Longrightarrow f_w(g_w^i\ \{\}) = g_w\ (g_w^i\ \{\}) = g_w^i\ \{\}$
$\Longrightarrow L\ w\ X = lfp\ f_w \subseteq g_w^i\ \{\} = Lb\ w\ X$

If $Lb\ w\ X = V_w$: $L\ w\ X \subseteq V_w$ (by Lemma)

12 Live Variable Analysis
Correctness of L
Dead Variable Elimination
True Liveness
Comparisons

Comparison of analyses

Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:

Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:
  - it analyses the executions starting from some point,

Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:
    - it analyses the executions starting from some point,
    - variables *must* be assigned (on every program path) before they are used.

Comparison of analyses

- Definite initialization analysis is a
  forward must analysis:
  - it analyses the executions starting from some point,
  - variables *must* be assigned (on every program path) before they are used.

- Live variable analysis is a
  backward may analysis:

Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:
  - it analyses the executions starting from some point,
  - variables *must* be assigned (on every program path) before they are used.

- Live variable analysis is a *backward may analysis*:
  - it analyses the executions ending in some point,

Comparison of analyses

- Definite initialization analysis is a *forward must analysis*:
    - it analyses the executions starting from some point,
    - variables *must* be assigned (on every program path) before they are used.

- Live variable analysis is a *backward may analysis*:
    - it analyses the executions ending in some point,
    - live variables *may* be used (on some program path) before they are assigned.

Comparison of DFA frameworks

Comparison of DFA frameworks

Program representation:

Comparison of DFA frameworks

Program representation:

- Traditionally (e.g. Aho/Sethi/Ullman), DFA is performed on *control flow graphs* (CFGs).

Comparison of DFA frameworks

Program representation:

- Traditionally (e.g. Aho/Sethi/Ullman), DFA is performed on *control flow graphs* (CFGs). Application: optimization of intermediate or low-level code.

Comparison of DFA frameworks

Program representation:

- Traditionally (e.g. Aho/Sethi/Ullman), DFA is performed on *control flow graphs* (CFGs). Application: optimization of intermediate or low-level code.
- We analyse structured programs.

Comparison of DFA frameworks

Program representation:

- Traditionally (e.g. Aho/Sethi/Ullman), DFA is performed on *control flow graphs* (CFGs). Application: optimization of intermediate or low-level code.
- We analyse structured programs. Application: source-level program optimization.