

# 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:

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

# Commands

Abstract syntax:

**datatype** *com* = *SKIP*  
| *Assign string aexp*  
| *Seq com com*  
| *If bexp com com*  
| *While bexp com*

Com.thy

① IMP Commands

② Big-Step Semantics

③ Small-Step Semantics

# Big-step semantics

Concrete syntax:

$$(com, initial-state) \Rightarrow final-state$$

# Big-step semantics

Concrete syntax:

$$(com, initial-state) \Rightarrow final-state$$

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



# Big-step semantics

Concrete syntax:

$$(com, initial-state) \Rightarrow final-state$$

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

Command  $c$  started in state  $s$  terminates in state  $t$

# Big-step semantics

Concrete syntax:

$$(com, initial-state) \Rightarrow final-state$$

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

$$(\textit{SKIP}, s) \Rightarrow s$$

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

# Big-step rules

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

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

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

## Big-step rules

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

## Big-step rules

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

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

## Big-step rules

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



# Big-step rules

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

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

## 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

- $w = \textit{WHILE } b \textit{ DO } c$
- $b = \textit{NotEq } (V\ "x'')\ (N\ 2)$
- $c = "x'' ::= \textit{Plus } (V\ "x'')\ (N\ 1)$
- $s_i = s("x'' := i)$

## Examples: derivation trees

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

where  $w = \text{WHILE } b \text{ DO } c$   
 $b = \text{NotEq } (V\ "x'')\ (N\ 2)$   
 $c = "x'' ::= \text{Plus } (V\ "x'')\ (N\ 1)$   
 $s_i = s("x'' := i)$

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

Logically speaking

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

is just infix syntax for

$$\textit{big\_step} (c,s) t$$

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

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

# Rule inversion

What can we deduce from

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

# Rule inversion

What can we deduce from

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

# Rule inversion

What can we deduce from

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

# Rule inversion

What can we deduce from

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

# Rule inversion

What can we deduce from

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

# Rule inversion

What can we deduce from

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



# Rule inversion

What can we deduce from

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

# Automating rule inversion

Isabelle command **inductive\_cases** produces theorems that perform rule inversions automatically.

We reformulate the inverted rules. Example:

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

We reformulate the inverted rules. Example:

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

is logically equivalent to

$$\frac{\bigwedge s_2. \llbracket (c_1, s_1) \Rightarrow s_2; (c_2, s_2) \Rightarrow s_3 \rrbracket \Longrightarrow P}{P}$$

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

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

Replaces assem  $(c_1;; c_2, s_1) \Rightarrow s_3$  by two assems  
 $(c_1, s_1) \Rightarrow s_2$  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

$$\frac{\bigwedge s_2. \llbracket (c_1, s_1) \Rightarrow s_2; (c_2, s_2) \Rightarrow s_3 \rrbracket \Longrightarrow P}{P}$$

Replaces assem  $(c_1;; c_2, s_1) \Rightarrow s_3$  by two assems  $(c_1, s_1) \Rightarrow s_2$  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

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

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

No  $\exists$  and  $\wedge$ !

The general format: *elimination rules*

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



The general format: *elimination rules*

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

(possibly with  $\bigwedge \bar{x}$  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  $\bigwedge \bar{x}$  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  $\bigwedge \bar{x}$  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  $w = \text{WHILE } b \text{ DO } c$

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

# 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

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

# Equivalence proof

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

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:

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

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

# Small-step semantics

Concrete syntax:

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

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

*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:

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

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

*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:

$$(c_1, s_1) \rightarrow (c_2, s_2) \rightarrow (c_3, s_3) \rightarrow \dots$$

# 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:

$(SKIP, s)$  is final

The intention:

$(SKIP, s)$  is final

Why?

*SKIP* is the empty program.

The intention:

$(SKIP, s)$  is final

Why?

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

# Small-step rules

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

# Small-step rules

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

# Small-step rules

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

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



# Small-step rules

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

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

# Small-step rules

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

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

$$\frac{(c_1, s) \rightarrow (c'_1, s')}{(c_1;; c_2, s) \rightarrow}$$

# Small-step rules

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

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

$$\frac{(c_1, s) \rightarrow (c'_1, s')}{(c_1;; c_2, s) \rightarrow (c'_1;; c_2, s')}$$

# Small-step rules

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

# Small-step rules

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

# Small-step rules

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

## Small-step rules

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

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

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

# Small-step rules

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

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

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



## Small-step rules

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

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

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

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

## Small-step examples

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

...

where  $s = \langle "x'' := 3, "y'' := 7, "z'' := 5 \rangle$ .

## Small-step examples

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

where  $s = \langle "x'' := 3, "y'' := 7, "z'' := 5 \rangle$ .

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

where

$$\begin{aligned} w &= \text{WHILE } b \text{ DO } c \\ b &= \text{Less } (V "x'') (N 1) \\ c &= "x'' ::= \text{Plus } (V "x'') (N 1) \\ s_n &= \langle "x'' := n \rangle \end{aligned}$$

# Small\_Step.thy

Semantics

Are big and small-step semantics equivalent?

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

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

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

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

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

Proof by rule induction



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

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

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

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

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

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

In two cases a lemma is needed:

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

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

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

In two cases a lemma is needed:

**Lemma**

$$(c_1, s) \rightarrow^* (c_1', s') \implies (c_1;; c_2, s) \rightarrow^* (c_1';; c_2, s')$$

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

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

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

In two cases a lemma is needed:

**Lemma**

$(c_1, s) \rightarrow^* (c_1', s') \implies (c_1;; c_2, s) \rightarrow^* (c_1';; c_2, s')$

Proof by rule induction.

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

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

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

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

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

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

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

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

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

In the induction step a lemma is needed:



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

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

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

In the induction step a lemma is needed:

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

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

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

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

In the induction step a lemma is needed:

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

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

# Equivalence

**Corollary**  $cs \Rightarrow t \iff cs \rightarrow^* (SKIP, t)$

# 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**  $final(c, s) \implies c = SKIP$

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .



## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

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**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

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**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

- Case  $c_1;; c_2$ : by case distinction:
  - $c_1 = SKIP \implies \neg final(c_1;; c_2, s)$

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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

## Can execution stop prematurely?

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

**Lemma**  $final(c, s) \implies c = SKIP$

We prove the contrapositive

$$c \neq SKIP \implies \neg final(c, s)$$

by induction on  $c$ .

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



By rule inversion:  $(SKIP, s) \rightarrow ct \implies False$

By rule inversion:  $(SKIP, s) \rightarrow ct \implies False$

Together:

**Corollary**  $final(c, s) = (c = SKIP)$

# Infinite executions

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

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

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

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (SKIP, t))$

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (SKIP, t))$   
(by big = small)

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge final\ cs')$

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (SKIP, t))$   
    (by big = small)  
=  $(\exists cs'. cs \rightarrow^* cs' \wedge final\ cs')$



# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (\text{SKIP}, t))$   
    (by big = small)  
=  $(\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$   
    (by final = SKIP)

# Infinite executions

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

**Lemma**  $(\exists t. cs \Rightarrow t) = (\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$

Proof:  $(\exists t. cs \Rightarrow t)$   
=  $(\exists t. cs \rightarrow^* (\text{SKIP}, t))$   
    (by big = small)  
=  $(\exists cs'. cs \rightarrow^* cs' \wedge \text{final } cs')$   
    (by final = SKIP)

Equivalent:

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

# May versus Must

→ 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  $cs \rightarrow cs' \rightarrow \dots$



# 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:*

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

Some obvious facts:

*Consequence rule:*

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

$wp$  of equivalent programs is equal

$$c \sim c' \implies wp\ c = wp\ c'$$

Correctness of *exp*

Correctness of *exp*?

Correctness of *exp*?

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

Correctness of *exp*?

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

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



Correctness of *exp*?

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

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

In general:  $P \ s \implies wp \ c \ Q \ s$

How to prove correctness of programs?

$$P \text{ } s \Longrightarrow wp \text{ } c \text{ } Q \text{ } s$$

How to prove correctness of programs?

$$P \text{ } s \Longrightarrow wp \text{ } c \text{ } Q \text{ } s$$

$$wp \text{ } SKIP \text{ } Q \text{ } s =$$

How to prove correctness of programs?

$$P \text{ } s \Longrightarrow wp \text{ } c \text{ } Q \text{ } s$$

$$wp \text{ } SKIP \text{ } Q \text{ } s = Q \text{ } s$$

How to prove correctness of programs?

$$P \text{ } s \Longrightarrow wp \text{ } c \text{ } Q \text{ } s$$

$$wp \text{ } SKIP \text{ } Q \text{ } s = Q \text{ } s$$

$$wp \text{ } (x ::= a) \text{ } Q \text{ } s =$$

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?

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

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

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

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

How to prove correctness of programs?

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

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

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

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



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?

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

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

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

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

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

How to prove correctness of programs?

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

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

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

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

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

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*?

$$\begin{aligned} & wp \ (WHILE \ b \ DO \ c) \ Q \ s \\ & = if \ bval \ b \ s \ then \ wp \ c \ (wp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \\ & \quad Q \ s \end{aligned}$$

That was easy! But what about *While*?

$$\begin{aligned} & wp \ (WHILE \ b \ DO \ c) \ Q \ s \\ &= if \ bval \ b \ s \ then \ wp \ c \ (wp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \\ & \quad Q \ s \end{aligned}$$

Unfolding will continue forever!



That was easy! But what about *While*?

$$\begin{aligned} & wp \ (WHILE \ b \ DO \ c) \ Q \ s \\ & = if \ bval \ b \ s \ then \ wp \ c \ (wp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \\ & \quad Q \ s \end{aligned}$$

Unfolding will continue forever!

Obviously, need some inductive argument!

That was easy! But what about *While*?

$$\begin{aligned} & wp \ (WHILE \ b \ DO \ c) \ Q \ s \\ & = if \ bval \ b \ s \ then \ wp \ c \ (wp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \\ & \quad Q \ s \end{aligned}$$

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:

$$c \sim c' \implies wlp\ c = wlp\ c'$$

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

Some obvious facts:

$$c \sim c' \implies wlp\ c = wlp\ c'$$

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

Relation between  $wp$  and  $wlp$

$$wp\ c\ Q\ s \implies wlp\ c\ Q\ s$$

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

Some obvious facts:

$$c \sim c' \implies wlp\ c = wlp\ c'$$

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

Relation between *wp* and *wlp*

$$wp\ c\ Q\ s \implies wlp\ c\ Q\ s$$

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

Unfold rules still hold:

$$wlp\ SKIP\ Q\ s = Q\ s$$

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

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

$$wlp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s = \\ (if\ bval\ b\ s\ then\ wlp\ c_1\ Q\ s\ else\ wlp\ c_2\ Q\ s)$$



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

$$\begin{aligned} \text{wlp } (\text{WHILE } b \text{ DO } c) \ Q \ s = \\ (\text{if } \text{bval } b \ s \text{ then } \text{wlp } c \ (\text{wlp } (\text{WHILE } b \text{ DO } c) \ Q) \ s \text{ else } \\ Q \ s) \end{aligned}$$

Let's try to find predicate  $I$ , such that

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

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

Let's try to find predicate  $I$ , such that

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

and  $I$  holds for start state.

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

Let's try to find predicate  $I$ , such that

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

and  $I$  holds for start state.

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

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

Let's try to find predicate  $I$ , such that

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

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

$$\begin{aligned} & \llbracket I \ s_0; \bigwedge s. I \ s \implies \textit{if } b \textit{val } b \ s \textit{ then } wlp \ c \ I \ s \textit{ else } Q \ s \rrbracket \\ & \implies wlp \ ( \textit{WHILE } b \ \textit{DO } c ) \ Q \ s_0 \end{aligned}$$

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 ...

$$P \ s \Longrightarrow \textit{wlp} \ c \ Q \ s$$

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

Now we can start proving programs ...

$$P \ s \Longrightarrow \textit{wlp} \ c \ Q \ s$$

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

Otherwise, use unfold rules.

Now we can start proving programs ...

$$P \ s \Longrightarrow \textit{wlp} \ c \ Q \ s$$

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

Otherwise, use unfold rules.

Iterate, until all *wlps* gone!

$wlp\_if\_eq$  and  $wlp\_whileI'$  produce *if-then-else*

$wlp\_if\_eq$  and  $wlp\_whileI'$  produce *if-then-else* which we have to split.

$wlp\_if\_eq$  and  $wlp\_whileI'$  produce *if-then-else* which we have to split.

Combine rule with splitting!

# Wp\_Demo.thy

Proving Partial Correctness



But how about termination?

But how about termination?

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

But how about termination?

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

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

But how about termination?

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

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

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  $x_1 > x_2 > \dots$

Well-foundedness implies induction principle

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

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

$$\bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ c\ (\lambda s'. I\ s' \wedge s' < s)\ s \\ \text{else } Q\ s$$



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

$$\bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ c\ (\lambda s'. I\ s' \wedge s' < s)\ s \\ \text{else } Q\ s$$

Then use wf-induction to prove:

$$\begin{aligned} & \llbracket wf\ R; I\ s_0; \\ & \bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ c\ (\lambda s'. I\ s' \wedge (s', s) \in \\ & R)\ s \text{ else } Q\ s \rrbracket \\ & \implies wp\ (WHILE\ b\ DO\ c)\ Q\ s_0 \end{aligned}$$

Or, equivalently

**assumes**  $WF: wf\ R$

**assumes**  $INIT: I\ s_0$

**assumes**  $STEP: \bigwedge s. \llbracket I\ s; bval\ b\ s \rrbracket$   
 $\implies wp\ c\ (\lambda s'. I\ s' \wedge (s', s) \in R)\ s$

**assumes**  $FINAL: \bigwedge s. \llbracket I\ s; \neg bval\ b\ s \rrbracket \implies Q\ s$

**shows**  $wp\ (WHILE\ b\ DO\ c)\ Q\ s_0$

Or, equivalently

**assumes**  $WF: wf\ R$

**assumes**  $INIT: I\ s_0$

**assumes**  $STEP: \bigwedge s. \llbracket I\ s; bval\ b\ s \rrbracket$   
 $\implies wp\ c\ (\lambda s'. I\ s' \wedge (s', s) \in R)\ s$

**assumes**  $FINAL: \bigwedge s. \llbracket I\ s; \neg bval\ b\ s \rrbracket \implies Q\ s$

**shows**  $wp\ (WHILE\ b\ DO\ c)\ Q\ s_0$

Now we can prove total correctness ...

Wp\_Demo.thy

Total Correctness

**lemma** *ASSUME\_Θ<sub>alt</sub>*:

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

**unfolding** *ASSUME\_Θ\_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

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

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

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

$BBinop::(bool \Rightarrow bool \Rightarrow bool) \Rightarrow bexp \Rightarrow bexp \Rightarrow bexp$



# 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  $s \text{ ''}x\text{''}$  by (Isabelle) variable  $x$ .

Similar:  $s_0 \text{ ''}x\text{''}$  by  $x_0$ .

## More Readable VCs

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

Similar:  $s_0 \text{ ''}x\text{'}$  by  $x_0$ .

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

$$(\bigwedge x. P \ x) \Longrightarrow P \ (s \text{ ''}x\text{'})$$

# IMP2/Introduction.thy

More Readable VCs

# More Readable Annotations

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



# More Readable Annotations

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

E.g. write  $c \leq n_0 \wedge a = c * c$  for

$$s \text{ ''}c\text{''} \leq s_0 \text{ ''}n\text{''} \wedge s \text{ ''}a\text{''} = s \text{ ''}c\text{''} * s \text{ ''}c\text{''}$$

# 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

$$s \text{ ''}c\text{''} \leq s_0 \text{ ''}n\text{''} \wedge s \text{ ''}a\text{''} = s \text{ ''}c\text{''} * s \text{ ''}c\text{''}$$

Which variables to interpret?

# 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

$$s \text{ ''}c\text{''} \leq s_0 \text{ ''}n\text{''} \wedge s \text{ ''}a\text{''} = s \text{ ''}c\text{''} * s \text{ ''}c\text{''}$$

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

$$s \text{ "c"} \leq s_0 \text{ "n"} \wedge s \text{ "a"} = s \text{ "c"} * s \text{ "c"}$$

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  $c \leq n_0 \wedge a = c * c$  for

$$s \text{ ''}c\text{''} \leq s_0 \text{ ''}n\text{''} \wedge s \text{ ''}a\text{''} = s \text{ ''}c\text{''} * s \text{ ''}c\text{''}$$

Which variables to interpret? over which states?

All variables that occur in the program!

Precondition:  $x$  interpreted as  $s \text{ ''}x\text{''}$

# 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

$$s \text{ ''}c\text{''} \leq s_0 \text{ ''}n\text{''} \wedge s \text{ ''}a\text{''} = s \text{ ''}c\text{''} * s \text{ ''}c\text{''}$$

Which variables to interpret? over which states?

All variables that occur in the program!

Precondition:  $x$  interpreted as  $s \text{ ''}x\text{''}$

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

# 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:

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

Compute operation by iterating weaker operation

# Common Loop Patterns

We've seen a few loop's already:

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

Compute operation by iterating weaker operation

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

# Common Loop Patterns

We've seen a few loop's already:

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

Compute operation by iterating weaker operation

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

Use accumulator  $a$  and increment counter (count-up)

# Common Loop Patterns

We've seen a few loop's already:

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

Compute operation by iterating weaker operation

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

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:

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

Compute operation by iterating weaker operation

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

Use accumulator  $a$  and increment counter (count-up)

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

Invariant:  $a = 2^c \wedge \dots$  (accumulator = f(iterations))

# Common Loop Patterns

We've seen a few loop's already:

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

Compute operation by iterating weaker operation

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

Use accumulator  $a$  and increment counter (count-up)

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

Invariant:  $a = 2^c \wedge \dots$  (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* ( $r*r \leq n$ ) {  $r=r+1$  };  $r=r-1$

# Approximate Naively

Invert monotonic function, by naively trying all values:

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

What does this compute

# Approximate Naively

Invert monotonic function, by naively trying all values:

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

What does this compute? square root, rounded down!

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

# Approximate Naively

Invert monotonic function, by naively trying all values:

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

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:

$r=1$ ; *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:

$r=1$ ; *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:  $?$   $(r-1)^2 \leq n \wedge \dots$  ( $r-1$  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:  $?$   $(r-1)^2 \leq n \wedge \dots$  ( $r-1$  below or equal result)

Applications: sqrt, log, ...

# IMP2/Examples.thy

Approximate from Below

# Bisection

We can compute  $\text{sqrt}$  more efficiently.

# Bisection

We can compute sqrt more efficiently.

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

# Bisection

We can compute sqrt more efficiently.

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

Idea: Half range in each step

# Bisection

We can compute sqrt more efficiently.

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

Idea: Half range in each step

Invariant



# Bisection

We can compute sqrt more efficiently.

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

Idea: Half range in each step

Invariant?

# Bisection

We can compute sqrt more efficiently.

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

Idea: Half range in each step

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

# Bisection

We can compute sqrt more efficiently.

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

Idea: Half range in each step

Invariant?  $l^2 \leq n < h^2 \wedge \dots$  (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:  $(b \text{ dvd } a) = (\exists k. a = b * k)$

Greatest Common Divisor:  $gcd::int \Rightarrow int \Rightarrow int$  such that

$gcd\ a\ b\ \text{dvd}\ a$  and  $gcd\ a\ b\ \text{dvd}\ b$  and

$\llbracket a \neq 0; b \neq 0; c\ \text{dvd}\ a; c\ \text{dvd}\ b \rrbracket \implies c \leq gcd\ a\ b$

# 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:  $\gcd \ x \ y = \gcd \ y \ (x \bmod y)$

# IMP2/Examples.thy

Euclid

- ④ Weakest Preconditions
- ⑤ Towards Simpler Verification of Programs
- ⑥ Example Verifications
- ⑦ Advanced Verification

# Modified Variables

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

## Modified Variables

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

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

## Modified Variables

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

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

## Modified Variables

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

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

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

## Modified Variables

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

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

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

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



## Modified Variables

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

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

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

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

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

# Modified Variables

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

Pre:  $n \geq 0$  Post:  $a = 2^{i_0}$  and only  $a, i$  changed.

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

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

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

Program modifies at most variables it assigns to

$\pi: (c, s) \Rightarrow t \implies \text{modifies} (\text{lhsv } \pi \ c) \ t \ s$

## 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)

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

For while-rule, we get

**lemma** *wp\_whileI\_modset*:

**fixes** *c*

**defines** [*simp*]: *modset*  $\equiv$  *lhsv c*

**assumes** *WF*: *wf R*

**assumes** *INIT*: *I s<sub>0</sub>*

**assumes** *STEP*:  $\bigwedge s. \llbracket \text{modifies modset } s \ s_0; I \ s; bval \ b \ s \rrbracket$

$\Longrightarrow wp \ c \ (\lambda s'. I \ s' \wedge (s', s) \in R) \ s$

**assumes** *FINAL*:  $\bigwedge s. \llbracket \text{modifies modset } s \ s_0; I \ s; \neg bval \ b \ s \rrbracket$

$\Longrightarrow Q \ s$

**shows** *wp (WHILE b DO c) Q s<sub>0</sub>*

# Modified Variables

The VCG will automatically rewrite with rule

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

# Modified Variables

The VCG will automatically rewrite with rule

$$\llbracket \text{modifies } vs \ s \ s'; x \notin vs \rrbracket \implies s \ x = s' \ x$$

**program\_spec** computes *lhs*-variables:

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

# 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:  $2^{2^{\cdot^{\cdot^2}}}$  ( $m$  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:

$$\begin{aligned} & \llbracket HT \pi P c Q; \bigwedge s. P' s \implies P s; \bigwedge s_0 s. \llbracket P s_0; P' s_0; Q \\ & s_0 s \rrbracket \implies Q' s_0 s \rrbracket \\ & \implies HT \pi P' c Q' \end{aligned}$$

# Modular Proofs

Re-using proofs:

$$\begin{aligned} & \llbracket HT \ \pi \ P \ c \ Q; \bigwedge s. P' \ s \implies P \ s; \bigwedge s_0 \ s. \llbracket P \ s_0; P' \ s_0; Q \\ & \ s_0 \ s \rrbracket \implies Q' \ s_0 \ s \rrbracket \\ & \implies HT \ \pi \ P' \ c \ Q' \end{aligned}$$

with modified sets:

$$\begin{aligned} & \llbracket HT\_mods \ \pi \ mods \ P \ c \ Q; P \ s; \bigwedge s'. \llbracket modifies \ mods \ s' \\ & \ s; Q \ s \ s' \rrbracket \implies Q' \ s' \rrbracket \\ & \implies wp \ \pi \ c \ Q' \ s \end{aligned}$$

# Modular Proofs

Re-using proofs:

$$\begin{aligned} & \llbracket HT \ \pi \ P \ c \ Q; \bigwedge s. P' \ s \implies P \ s; \bigwedge s_0 \ s. \llbracket P \ s_0; P' \ s_0; Q \\ & \ s_0 \ s \rrbracket \implies Q' \ s_0 \ s \rrbracket \\ & \implies HT \ \pi \ P' \ c \ Q' \end{aligned}$$

with modified sets:

$$\begin{aligned} & \llbracket HT\_mods \ \pi \ mods \ P \ c \ Q; P \ s; \bigwedge s'. \llbracket modifies \ mods \ s' \\ & \ s; Q \ s \ s' \rrbracket \implies Q' \ s' \rrbracket \\ & \implies wp \ \pi \ c \ Q' \ s \end{aligned}$$

VCG will automatically use this rule.

# Modular Proofs

Re-using proofs:

$$\begin{aligned} & \llbracket HT \ \pi \ P \ c \ Q; \bigwedge s. P' \ s \implies P \ s; \bigwedge s_0 \ s. \llbracket P \ s_0; P' \ s_0; Q \\ & \ s_0 \ s \rrbracket \implies Q' \ s_0 \ s \rrbracket \\ & \implies HT \ \pi \ P' \ c \ Q' \end{aligned}$$

with modified sets:

$$\begin{aligned} & \llbracket HT\_mods \ \pi \ mods \ P \ c \ Q; P \ s; \bigwedge s'. \llbracket modifies \ mods \ s' \\ & \ s; Q \ s \ s' \rrbracket \implies Q' \ s' \rrbracket \\ & \implies wp \ \pi \ c \ Q' \ s \end{aligned}$$

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:

$AssignIdx::char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$   
 $\pi: (x[i] ::= a, s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$

# 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:

$AssignIdx::char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$   
 $\pi: (x[i] ::= a, s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$   
 $ArrayCpy::char\ list \Rightarrow char\ list \Rightarrow com$   
 $\pi: (x[] ::= y, s) \Rightarrow s(x := s\ y)$

# 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:

$AssignIdx::char\ list \Rightarrow aexp \Rightarrow aexp \Rightarrow com$   
 $\pi: (x[i] ::= a, s) \Rightarrow s(x := (s\ x)(aval\ i\ s := aval\ a\ s))$

$ArrayCpy::char\ list \Rightarrow char\ list \Rightarrow com$   
 $\pi: (x[] ::= y, s) \Rightarrow s(x := s\ y)$

$ArrayClear::char\ list \Rightarrow com$   
 $\pi: (CLEAR\ x[], s) \Rightarrow s(x := \lambda_. 0)$

# Arrays

By default, we use index 0.

# Arrays

By default, we use index 0.

Abbreviations:

$$V\ x = \text{Vidx}\ x\ (N\ 0)$$

# Arrays

By default, we use index 0.

Abbreviations:

$$V\ x = \text{Vidx}\ x\ (N\ 0)$$

$$\text{Assign}\ x\ a = \text{AssignIdx}\ x\ (N\ 0)\ a$$

# Arrays

By default, we use index 0.

Abbreviations:

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

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



# Arrays

By default, we use index 0.

Abbreviations:

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

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

VCG: Guess type from variable usage

# Arrays

By default, we use index 0.

Abbreviations:

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

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:

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

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

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:

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

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

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

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

# 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

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

# Reasoning about Arrays

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

Set interval notation:

$\{l..h\}$ ,  $\{l..<h\}$ ,  $\{l<..h\}$ ,  $\{l<..<h\}$

Examples:

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

Elements 0 to 41 are positive

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

Elements  $l$  to  $<h$  are sorted

# Reasoning about Arrays

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

Set interval notation:

$\{l..h\}$ ,  $\{l..<h\}$ ,  $\{l<..h\}$ ,  $\{l<..<h\}$

Examples:

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

Elements 0 to 41 are positive

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

Elements  $l$  to  $<h$  are sorted

# Reasoning about Arrays

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

Set interval notation:

$\{l..h\}$ ,  $\{l..<h\}$ ,  $\{l<..h\}$ ,  $\{l<..<h\}$

Examples:

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

Elements 0 to 41 are positive

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

Elements  $l$  to  $<h$  are sorted

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

# IMP2/Examples.thy

Sortedness Check



# Binary Search Algorithm

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

# Binary Search Algorithm

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

Idea: Halve interval in each step.

# Binary Search Algorithm

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

Idea: Halve interval in each step.

This algorithm is **tricky** to implement correctly!

# Binary Search Algorithm

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

Idea: Halve interval in each step.

This algorithm is *tricky* to implement correctly!

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

— Donald Knuth

# Binary Search Algorithm

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

Idea: Halve interval in each step.

This algorithm is **tricky** to implement correctly!

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

— Donald Knuth

Only 5 out of 20 surveyed textbooks had correct implementations

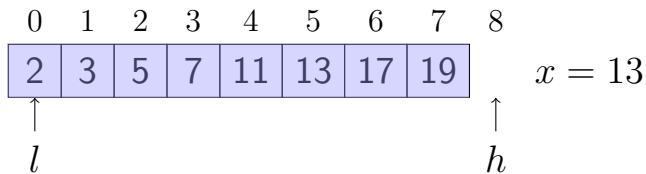
— Richard E. Pattis, 1988

# Binary Search Algorithm

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

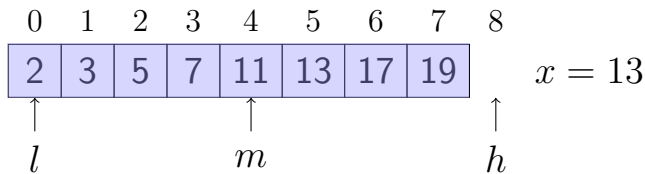
```
while (l < h) {  
    m = (l + h) / 2;  
    if (a[m] < x) l = m + 1  
    else h = m  
}
```

# Binary Search Algorithm



```
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  
}
```

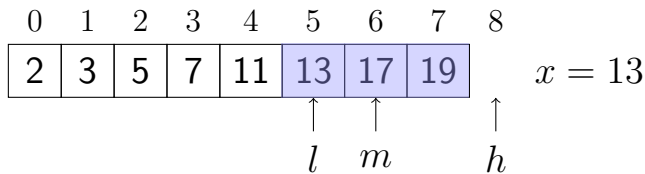


# Binary Search Algorithm

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

```
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

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

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

# Binary Search Algorithm

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

$\uparrow$        $\uparrow$   
 $l$        $h$

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

# Binary Search Algorithm

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

↑  
 $l, h$

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

# Binary Search Algorithm

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

↑  
 $l, h$

```
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

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

Invariant:

# Proving Binary Search

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

Invariant:

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

# Proving Binary Search

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

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

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

Invariant:

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

# IMP2/Examples.thy

Binary Search

# Insertion Sort

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

Idea: Build sorted array from start.

In each iteration, move next element to its position



# Specifying Sorting Algorithms

Precondition:  $l \leq h$

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

- Array is sorted

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

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

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

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

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

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

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

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

# Specifying Sorting Algorithms

Precondition:  $l \leq h$

Postcondition:

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

where

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

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



# Multisets in Isabelle

**imports** *HOL–Library.Multiset*

# Multisets in Isabelle

**imports** *HOL–Library.Multiset*

*'a multiset*: **Finite** multiset

# Multisets in Isabelle

**imports** *HOL–Library.Multiset*

*'a multiset*: **Finite** multiset

Some functions and syntax:

# Multisets in Isabelle

**imports** *HOL–Library.Multiset*

*'a multiset*: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

# Multisets in Isabelle

**imports** *HOL-Library.Multiset*

*'a multiset*: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

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

# Multisets in Isabelle

**imports** *HOL-Library.Multiset*

*'a multiset*: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

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

$m_1 + m_2$  — union of multisets

# Multisets in Isabelle

**imports** *HOL-Library.Multiset*

'a multiset: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

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

$m_1 + m_2$  — union of multisets

$a \in\# m$  — membership query

# Multisets in Isabelle

**imports** *HOL-Library.Multiset*

'a multiset: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

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

$m_1 + m_2$  — union of multisets

$a \in\# m$  — membership query

$\{\#a, b, c, c\#\}$  — Syntax for *add\_mset* and  $\{\#\}$



# Multisets in Isabelle

**imports** *HOL-Library.Multiset*

*'a multiset*: **Finite** multiset

Some functions and syntax:

$\{\#\}$  — empty multiset

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

$m_1 + m_2$  — union of multisets

$a \in\# m$  — membership query

$\{\#a, b, c, c\#\}$  — Syntax for *add\_mset* and  $\{\#\}$

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

**Multiset** of elements at indexes in finite **set**  $r$

# Proving Insertion Sort

Separate proof for inner loop!

# Proving Insertion Sort

Separate proof for inner loop!

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

# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop:

# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop: ?

# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop: ?

**assumes** *ran\_sorted a l j*

# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop: ?

**assumes** *ran\_sorted a l j*

**ensures** *ran\_sorted a l (j + 1)*

# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop: ?

**assumes** *ran\_sorted a l j*

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



# Proving Insertion Sort

Separate proof for inner loop!

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

Specification of inner loop: ?

**assumes** *ran\_sorted a l j*

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

**ensures** *mset\_ran a {l..j} = mset\_ran a<sub>0</sub> {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<sub>0</sub> {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**  $\text{ran\_sorted } a \ l \ j$

**ensures**  $\text{ran\_sorted } a \ l \ (j + 1)$  and

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

Invariant of outer loop:

$\text{ran\_sorted } a \ l \ j$

# 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**  $\text{ran\_sorted } a \ l \ j$

**ensures**  $\text{ran\_sorted } a \ l \ (j + 1)$  and

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

Invariant of outer loop:

$\text{ran\_sorted } a \ l \ j$

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

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && a[i] > key) {  
    a[i + 1] = a[i];  
    i = i - 1;  
};  
a[i + 1] = key
```

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && a[i] > key) {  
    a[i + 1] = a[i];  
    i = i - 1;  
};  
a[i + 1] = key
```

Intuition:

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && a[i] > key) {  
    a[i + 1] = a[i];  
    i = i - 1;  
};  
a[i + 1] = key
```

Intuition: ?

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && a[i] > key) {  
    a[i + 1] = a[i];  
    i = i - 1;  
};  
a[i + 1] = key
```

Intuition: ?

$a[j]$  is moved backwards



## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && a[i] > key) {  
    a[i + 1] = a[i];  
    i = i - 1;  
};  
a[i + 1] = key
```

Intuition: ?

$a[j]$  is moved backwards until

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && 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  $\leq a[j]$

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && 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

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && 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

## Insert: Inner Loop

```
key = a[j];  
i = j - 1;  
while (i >= 0 && 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.

## Insert: Inner Loop

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

# Insert: Inner Loop

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

Let's specify this intuition!

# Insert: Inner Loop

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

Let's specify this intuition!

It implies sortedness and mset-preservation



# Insert: 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

# Insert: 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!

## Insert: Inner Loop

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

## Insert: Inner Loop

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

assumes  $l < j$ , let  $key = a[j]$

# Insert: Inner Loop

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

assumes  $l < j$ , let  $key = a[j]$

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

# Insert: 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

# Insert: Inner Loop

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

assumes  $l < j$ , let  $key = a[j]$

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

ensures  $\forall k \in \{l..i\}. a[k] \geq key$  and  
 $a[i + 1] = key$  and

# Insert: 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

$\forall k \in \{i + 2..j\}. a\ k = a_0\ (k - 1)$



# Insert: 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  $l \leq i \longrightarrow a i \leq key$  and

# Insert: 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  $l \leq i \longrightarrow a i \leq key$  and

$\forall k \in \{i + 2..j\}. key < a k$

## Insert: Finding Invariant

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

## Insert: Finding Invariant

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

## Insert: Finding Invariant

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

## Insert: Finding Invariant

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

## Insert: Finding Invariant

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

Consider intermediate situation

## Insert: Finding Invariant

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

Consider intermediate situation



## Insert: Finding Invariant

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

Consider intermediate situation

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

## Insert: Finding Invariant

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

Consider intermediate situation

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

## Insert: Finding Invariant

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

Consider intermediate situation

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

## Insert: Finding Invariant

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

Consider intermediate situation

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

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle$



# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle =$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

$is\_local\ x \implies \langle s | t \rangle x =$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

$is\_local\ x \implies \langle s | t \rangle\ x = s\ x$

$is\_global\ x \implies \langle s | t \rangle\ x = t\ x$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle \ n = (\text{if } is\_local \ n \text{ then } s \ n \text{ else } t \ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

$is\_local \ x \implies \langle s | t \rangle \ x = s \ x$

$is\_global \ x \implies \langle s | t \rangle \ x = t \ x$

$is\_local \ x \implies \langle s | t \rangle (x := v) =$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle \ n = (\text{if } is\_local \ n \text{ then } s \ n \text{ else } t \ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

$is\_local \ x \implies \langle s | t \rangle \ x = s \ x$

$is\_global \ x \implies \langle s | t \rangle \ x = t \ x$

$is\_local \ x \implies \langle s | t \rangle (x := v) = \langle s(x := v) | t \rangle$

# State Combination

$\langle s_1 | s_2 \rangle$  – State with locals from  $s_1$ , globals from  $s_2$

$\langle s | t \rangle n = (\text{if } is\_local\ n \text{ then } s\ n \text{ else } t\ n)$

Some rules:  $\langle s | s \rangle = s$

$\langle s | \langle s' | t \rangle \rangle = \langle s | t \rangle$

$\langle \langle s | t' \rangle | t \rangle = \langle s | t \rangle$

$is\_local\ x \implies \langle s | t \rangle x = s\ x$

$is\_global\ x \implies \langle s | t \rangle x = t\ x$

$is\_local\ x \implies \langle s | t \rangle (x := v) = \langle s(x := v) | t \rangle$

$is\_global\ x \implies \langle s | t \rangle (x := v) = \langle s | t(x := v) \rangle$

# 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:

$$\pi: (c, \langle \langle \rangle \rangle | s \rangle) \Rightarrow s' \implies \pi: (SCOPE\ c, s) \Rightarrow \langle s | s' \rangle$$

# Scope Command

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

Semantics:

$$\pi: (c, \langle \langle \rangle \rangle | s \rangle) \Rightarrow s' \implies \pi: (\text{SCOPE } c, s) \Rightarrow \langle s | s' \rangle$$

$$\text{Unfold rule: } wp \ \pi \ (\text{SCOPE } c) \ Q \ s \\ =$$

# Scope Command

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

Semantics:

$$\pi: (c, \langle \langle \rangle \rangle | s \rangle) \Rightarrow s' \implies \pi: (\text{SCOPE } c, s) \Rightarrow \langle s | s' \rangle$$

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

# Scope Command

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

Semantics:

$$\pi: (c, \langle \langle \rangle \rangle | s \rangle) \Rightarrow s' \implies \pi: (SCOPE\ c, s) \Rightarrow \langle s | s' \rangle$$

$$\text{Unfold rule: } wp\ \pi\ (SCOPE\ c)\ Q\ s \\ =$$

# Scope Command

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

Semantics:

$$\pi: (c, \langle \langle \rangle \rangle | s \rangle) \Rightarrow s' \implies \pi: (\text{SCOPE } c, s) \Rightarrow \langle s | s' \rangle$$

$$\begin{aligned} \text{Unfold rule: } wp \ \pi \ (\text{SCOPE } c) \ Q \ s \\ = wp \ \pi \ c \ (\lambda s'. Q \ \langle s | s' \rangle) \ \langle \langle \rangle \rangle | s \rangle \end{aligned}$$

# Parameter Passing

Pass information over scope boundaries by globals

# Parameter Passing

Pass information over scope boundaries by globals

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

# Parameter Passing

Pass information over scope boundaries by globals

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

$G_1 = a_1; \dots; G_n = a_n; \text{inline } f; r = G$



# Parameter Passing

Pass information over scope boundaries by globals

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

$G_1 = a_1; \dots; G_n = a_n; \text{inline } f; r = G$

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

# Parameter Passing

Pass information over scope boundaries by globals

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

$G_1 = a_1; \dots; G_n = a_n; \text{inline } f; r = G$

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

*scope*  $\{ p_1 = G_1; \dots; p_n = G_n; \text{body}; G = x \}$

# Wrapping Specification

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

# Wrapping Specification

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

How to derive specification for procedure?

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

# Wrapping Specification

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

How to derive specification for procedure?

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

Recall:

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

# Prologue

$$\begin{aligned} HT \pi P \text{ body } Q &\implies \\ HT \pi (wp \pi \text{ prologue } P) (\text{prologue};; \text{body}) \\ &(\lambda s_0 s. wp \pi \text{ prologue } (\lambda s_0. Q s_0 s) s_0) \end{aligned}$$

Intuition: Weakest precondition to enforce  $P$  after prologue

# Epilogue

$$\begin{aligned} & \llbracket HT \pi P \textit{body} Q; \forall s. \exists t. \pi: (\textit{epilogue}, s) \Rightarrow t \rrbracket \\ & \implies HT \pi P (\textit{body};; \textit{epilogue}) (\lambda s_0. sp \pi (Q s_0) \\ & \textit{epilogue}) \end{aligned}$$

Intuition: Strongest postcondition we get from  $Q$  after epilogue

# Strongest Postconditions

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



# Strongest Postconditions

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

Selected rules:

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

# Strongest Postconditions

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

Selected rules:

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

# Strongest Postconditions

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

Selected rules:

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

# Strongest Postconditions

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

Selected rules:

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

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

# Strongest Postconditions

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

Selected rules:

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

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

# Strongest Postconditions

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

Selected rules:

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

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

# Strongest Postconditions

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

Selected rules:

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

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

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

# Strongest Postconditions

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

Selected rules:

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

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

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



# Strongest Postconditions

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

Selected rules:

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

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

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

# Wrapping Specification

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

# Wrapping Specification

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

Derive specification for

# Wrapping Specification

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

Derive specification for

Parameter assignments:  $HT\ \pi\ P\ c\ Q \implies$

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

# Wrapping Specification

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

Derive specification for

Parameter assignments:  $HT\ \pi\ P\ c\ Q \implies$

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

Return value assignment:  $HT\ \pi\ P\ c\ Q \implies$

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

# Wrapping Specification

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

Derive specification for

Parameter assignments:  $HT\ \pi\ P\ c\ Q \implies$

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

Return value assignment:  $HT\ \pi\ P\ c\ Q \implies$

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

Scope:  $HT\ \pi\ P\ c\ Q \implies$

$$HT\ \pi\ (\lambda s. P\ <<>|s>)\ (SCOPE\ c)\ (\lambda s_0\ s. \exists l. Q\ <<>|s_0> \\ <l|s>)$$

# IMP2/Examples.thy

Merge as Procedure

## ⑦ Advanced Verification

Arrays

Data Refinement

Local Variables

Recursion



# Recursive Procedures

Program is map  $pname \rightarrow com$

# Recursive Procedures

Program is map  $pname \rightarrow com$

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

# Recursive Procedures

Program is map  $pname \rightarrow com$

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

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

# Recursive Procedures

Program is map  $pname \rightarrow com$

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

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

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

# Recursive Procedures

Program is map  $pname \rightarrow com$

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

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

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

Note: Gets stuck if procedure does not exist!

# Recursive Procedures

Program is map  $pname \rightarrow com$

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

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

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

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

## Proof Rules for Recursion

Unfolding:  $\pi \ p = \textit{Some } c \implies \textit{wp } \pi \ (\textit{PCall } p) \ Q \ s = \textit{wp } \pi \ c \ Q \ s$

## Proof Rules for Recursion

Unfolding:  $\pi \ p = \text{Some } c \implies wp \ \pi \ (PCall \ p) \ Q \ s = wp \ \pi \ c \ Q \ s$

Idea: Well-Founded induction on state



# Proof Rules for Recursion

Unfolding:  $\pi \ p = \text{Some } c \implies wp \ \pi \ (PCall \ p) \ Q \ s = wp \ \pi \ c \ Q \ s$

Idea: Well-Founded induction on state

**assumes**  $wf \ R$

$$\bigwedge s. \llbracket HT \ \pi \ (\lambda s'. (s', s) \in R \wedge P \ s') \ (PCall \ p) \ Q; P \ s \rrbracket \\ \implies wp \ \pi \ (PCall \ p) \ (Q \ s) \ s$$

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

# Proof Rules for Recursion

Unfolding:  $\pi \ p = \text{Some } c \implies wp \ \pi \ (PCall \ p) \ Q \ s = wp \ \pi \ c \ Q \ s$

Idea: Well-Founded induction on state

**assumes**  $wf \ R$

$$\bigwedge s. \llbracket HT \ \pi \ (\lambda s'. (s', s) \in R \wedge P \ s') \ (PCall \ p) \ Q; P \ s \rrbracket \\ \implies wp \ \pi \ (PCall \ p) \ (Q \ s) \ s$$

**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  $\Theta$  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  $\Theta$  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  $\Theta$  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**  $RL: \bigwedge f_0\ s_0. \llbracket ASSUME\_Θ\ \pi\ f_0\ s_0\ R\ Θ \rrbracket \implies$   
 $PROVE\_Θ\ \pi\ f_0\ s_0\ Θ$   
**shows**  $HT'set\ \pi\ Θ$

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*:  $\bigwedge f_0 \ s_0. \llbracket \text{ASSUME\_}\Theta \ \pi \ f_0 \ s_0 \ R \ \Theta \rrbracket \Longrightarrow$   
*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.

$$\llbracket \pi \ p = \text{Some } c; \text{HT\_mods } \pi \ \text{mods } P \ c \ Q \rrbracket \Longrightarrow \text{HT\_mods } \pi \ \text{mods } P \ (P\text{Call } p) \ Q$$

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!

$$\pi': (c, s) \Rightarrow t \implies \pi: (P\textit{Scope } \pi' c, s) \Rightarrow t$$

Call procedure with local procedure environment

# Local Procedures

Idea: Recursive procedure names only valid locally!

No need to worry about name clashes!

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

Call procedure with local procedure environment

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

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

$$\begin{aligned} & \llbracket I \ s_0; \bigwedge s. I \ s \implies \textit{if } b \textit{val } b \ s \textit{ then } wlp \ \pi \ c \ I \ s \textit{ else } Q \ s \rrbracket \\ & \implies wlp \ \pi \ (\textit{WHILE } b \ \textit{DO } c) \ Q \ s_0 \end{aligned}$$

# Partial Correctness

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

What invariant shall we use?

# Partial Correctness

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

What invariant shall we use?

$$wlp \ \pi \ c \ Q!$$

# Total Correctness

$$\begin{aligned} & \llbracket wf\ R; \ I\ s_0; \\ & \bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ \pi\ c\ (\lambda s'. I\ s' \wedge (s', s) \\ & \in R) \ s \text{ else } Q\ s \rrbracket \\ & \implies wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0 \end{aligned}$$

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



# Total Correctness

$$\begin{aligned} & \llbracket wf\ R; \ I\ s_0; \\ & \bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ \pi\ c\ (\lambda s'. I\ s' \wedge (s', s) \\ & \in R) \ s \text{ else } Q\ s \rrbracket \\ & \implies wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0 \end{aligned}$$

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

Variant?

# Total Correctness

$$\begin{aligned} & \llbracket wf\ R; \ I\ s_0; \\ & \bigwedge s. I\ s \implies \text{if } bval\ b\ s \text{ then } wp\ \pi\ c\ (\lambda s'. I\ s' \wedge (s', s) \\ & \in R) \ s \text{ else } Q\ s \rrbracket \\ & \implies wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0 \end{aligned}$$

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* =

*LOADI int*

load value

| *LOAD vname*

load var

| *ADD*

add top of stack

# Stack Machine

Instructions:

**datatype** *instr* =

*LOADI int*

load value

| *LOAD vname*

load var

| *ADD*

add top of stack

| *STORE vname*

store var

# Stack Machine

Instructions:

**datatype** *instr* =

<i>LOADI int</i>	load value
<i>LOAD vname</i>	load var
<i>ADD</i>	add top of stack
<i>STORE vname</i>	store var
<i>JMP int</i>	jump

# Stack Machine

Instructions:

**datatype** *instr* =

<i>LOADI int</i>	load value
<i>LOAD vname</i>	load var
<i>ADD</i>	add top of stack
<i>STORE vname</i>	store var
<i>JMP int</i>	jump
<i>JMPLESS int</i>	jump if <

# Stack Machine

Instructions:

**datatype** *instr* =

*LOADI int*

load value

| *LOAD vname*

load var

| *ADD*

add top of stack

| *STORE vname*

store var

| *JMP int*

jump

| *JMPLESS int*

jump if <

| *JMPGE int*

jump if  $\geq$



# Semantics

Type synonyms:

$$\textit{stack} = \textit{int list}$$
$$\textit{config} = \textit{int} \times \textit{state} \times \textit{stack}$$

# Semantics

Type synonyms:

$stack = int\ list$

$config = int \times state \times stack$



Execution of 1 instruction:

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



# Instruction execution

$iexec\ instr\ (i, s, stk) =$   
(**case**  $instr$  of  $LOADI\ n \Rightarrow (i + 1, s, n \# stk)$   
|  $LOAD\ x \Rightarrow (i + 1, s, s\ x \# stk)$   
|  $ADD \Rightarrow (i + 1, s, (hd2\ stk + hd\ stk) \# tl2\ stk)$   
|  $STORE\ x \Rightarrow (i + 1, s(x := hd\ stk), tl\ stk)$   
|  $JMP\ n \Rightarrow (i + 1 + n, s, stk)$   
|  $JMPLESS\ n \Rightarrow$   
    ( $if\ hd2\ stk < hd\ stk\ then\ i + 1 + n\ else\ i + 1,$   
       $s, tl2\ stk)$ )  
|  $JMPGE\ n \Rightarrow$   
    ( $if\ hd\ stk \leq hd2\ stk\ then\ i + 1 + n\ else\ i + 1,$   
       $s, tl2\ stk)$ )



# Program execution (1 step)

Programs are instruction lists.

# Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$instr\ list \vdash config \rightarrow config$$

# Program execution (1 step)

Programs are instruction lists.

Executing one program step:

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

$$P \vdash c \rightarrow c' =$$

$$(\exists i \ s \ stk.$$

$$c = (i, s, stk) \wedge$$

$$c' = iexec(P !! i) (i, s, stk) \wedge$$

$$0 \leq i \wedge i < size\ P)$$

# Program execution (1 step)

Programs are instruction lists.

Executing one program step:

$$instr\ list \vdash config \rightarrow config$$

$$P \vdash c \rightarrow c' =$$

$$(\exists i\ s\ stk.$$

$$c = (i, s, stk) \wedge$$

$$c' = iexec\ (P\ !!\ i)\ (i, s, stk) \wedge$$

$$0 \leq i \wedge i < size\ P)$$

where  $'a\ list\ !!\ int$  = nth instruction of list

$size :: 'a\ list \Rightarrow int$  = list size as integer



# Program execution ( $\ast$ 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:

$$acompile (N\ n) = [LOADI\ n]$$

$$acompile (V\ x) = [LOAD\ x]$$

$$acompile (Plus\ a_1\ a_2) = acompile\ a_1\ @\ acompile\ a_2\ @\ [ADD]$$

Correctness theorem:

*acompile a*

$$\vdash (0, s, stk) \rightarrow^* (size\ (acompile\ a), s, aval\ a\ s \neq stk)$$

## Compiling *aexp*

Same as before:

$$acompile (N\ n) = [LOADI\ n]$$

$$acompile (V\ x) = [LOAD\ x]$$

$$acompile (Plus\ a_1\ a_2) = acompile\ a_1\ @\ acompile\ a_2\ @\ [ADD]$$

Correctness theorem:

*acompile a*

$$\vdash (0, s, stk) \rightarrow^* (size\ (acompile\ a), s, aval\ a\ s \neq stk)$$

Proof by induction on *a*

## Compiling *aexp*

Same as before:

$$acompile (N\ n) = [LOADI\ n]$$

$$acompile (V\ x) = [LOAD\ x]$$

$$acompile (Plus\ a_1\ a_2) = acompile\ a_1\ @\ acompile\ a_2\ @\ [ADD]$$

Correctness theorem:

*acompile a*

$$\vdash (0, s, stk) \rightarrow^* (size\ (acompile\ a), s, eval\ a\ s \neq stk)$$

Proof by induction on *a* (with arbitrary *stk*).

# Compiling *aexp*

Same as before:

$$acompile (N\ n) = [LOADI\ n]$$

$$acompile (V\ x) = [LOAD\ x]$$

$$acompile (Plus\ a_1\ a_2) = acompile\ a_1\ @\ acompile\ a_2\ @\ [ADD]$$

Correctness theorem:

*acompile a*

$$\vdash (0, s, stk) \rightarrow^* (size\ (acompile\ a), s, eval\ a\ s \neq stk)$$


Proof by induction on *a* (with arbitrary *stk*).

Needs lemmas!

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$\begin{aligned} P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk') \end{aligned}$$



$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk')$$

Proofs by rule induction on  $\rightarrow^*$ ,

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk')$$

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

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk')$$

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

$$P \vdash c \rightarrow c' \implies P @ P' \vdash c \rightarrow c'$$

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$

$$\begin{aligned} P \vdash (i, s, stk) \rightarrow^* (i', s', stk') &\implies \\ P' @ P \vdash (size P' + i, s, stk) &\rightarrow^* (size P' + i', s', stk') \end{aligned}$$

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

$$P \vdash c \rightarrow c' \implies P @ P' \vdash c \rightarrow c'$$

$$\begin{aligned} P \vdash (i, s, stk) \rightarrow (i', s', stk') &\implies \\ P' @ P \vdash (size P' + i, s, stk) &\rightarrow (size P' + i', s', stk') \end{aligned}$$

$$P \vdash c \rightarrow^* c' \implies P @ P' \vdash c \rightarrow^* c'$$



$$P \vdash (i, s, stk) \rightarrow^* (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow^* (size P' + i', s', stk')$$

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

$$P \vdash c \rightarrow c' \implies P @ P' \vdash c \rightarrow c'$$

$$P \vdash (i, s, stk) \rightarrow (i', s', stk') \implies \\ P' @ P \vdash (size P' + i, s, stk) \rightarrow (size P' + i', s', stk')$$



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  $b = \text{And } (\text{Less } (V \text{ ''x''}) (V \text{ ''y''}))$   
 $(\text{Not } (\text{Less } (V \text{ ''z''}) (V \text{ ''a''})))$ .

## Example

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

$bcomp \ b \ \text{False } 3 =$



## Example

Let  $b = \text{And } (\text{Less } (V \text{ "x"}) (V \text{ "y"}))$   
 $(\text{Not } (\text{Less } (V \text{ "z"}) (V \text{ "a"})))$ .

$bcomp \ b \ \text{False} \ 3 =$

$[\text{LOAD } \text{"x"},$

## Example

Let  $b = \text{And } (\text{Less } (V \text{ "x"}) (V \text{ "y"}))$   
 $(\text{Not } (\text{Less } (V \text{ "z"}) (V \text{ "a"})))$ .

$bcomp \ b \ \text{False} \ 3 =$

$[LOAD \ \text{"x"},$   
 $LOAD \ \text{"y"},$

## Example

Let  $b = \text{And} \ (\text{Less} \ (V \ "x") \ (V \ "y"))$   
 $(\text{Not} \ (\text{Less} \ (V \ "z") \ (V \ "a")))$ .

$bcomp \ b \ \text{False} \ 3 =$

$[LOAD \ "x",$   
 $LOAD \ "y",$   
 $,$

## Example

Let  $b = \text{And} \ (\text{Less} \ (V \ "x") \ (V \ "y"))$   
 $(\text{Not} \ (\text{Less} \ (V \ "z") \ (V \ "a")))$ .

$bcomp \ b \ \text{False} \ 3 =$

$[LOAD \ "x",$   
 $LOAD \ "y",$   
 $LOAD \ "z",$

## Example

Let  $b = \text{And} \ (\text{Less} \ (V \ "x") \ (V \ "y"))$   
 $(\text{Not} \ (\text{Less} \ (V \ "z") \ (V \ "a")))$ .

$bcomp \ b \ \text{False} \ 3 =$

[ $LOAD \ "x",$   
 $LOAD \ "y",$   
 $,$   
 $LOAD \ "z",$   
 $LOAD \ "a",$

## Example

Let  $b = \text{And } (\text{Less } (V \text{ "x"}) (V \text{ "y"}))$   
 $(\text{Not } (\text{Less } (V \text{ "z"}) (V \text{ "a"})))$ .

$bcomp \ b \ False \ 3 =$

[ $LOAD \text{ "x"}$ ,  
 $LOAD \text{ "y"}$ ,

$\text{'}$   
 $LOAD \text{ "z"}$ ,  
 $LOAD \text{ "a"}$ ,

]

## Example

Let  $b = \text{And} \ (\text{Less} \ (V \ "x") \ (V \ "y"))$   
 $(\text{Not} \ (\text{Less} \ (V \ "z") \ (V \ "a")))$ .

$bcomp \ b \ False \ 3 =$

[ $LOAD \ "x",$   
 $LOAD \ "y",$   
 $JMPGE \ 6,$   
 $LOAD \ "z",$   
 $LOAD \ "a",$   
 $]$

## Example

Let  $b = \text{And} \ (\text{Less} \ (V \ "x") \ (V \ "y"))$   
 $(\text{Not} \ (\text{Less} \ (V \ "z") \ (V \ "a")))$ .

$bcomp \ b \ False \ 3 =$

[*LOAD "x",*  
*LOAD "y",*  
*JMPGE 6,*  
*LOAD "z",*  
*LOAD "a",*  
*JMPLESS 3*  
]





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

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

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

*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 :: 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<sub>1</sub> a<sub>2</sub>) 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<sub>1</sub> a<sub>2</sub>) f n =*

*acomp a<sub>1</sub> @*

*acomp a<sub>2</sub> @ (if f then [JMPLESS n] else [JMPGE 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<sub>1</sub> a<sub>2</sub>) f n =*

*acompa a<sub>1</sub> @*

*acompa a<sub>2</sub> @ (if f then [JMPLESS n] else [JMPGE n])*

*bcomp (And b<sub>1</sub> b<sub>2</sub>) f n =*

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

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

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

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

$acomp\ a_1\ @$

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

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

$let\ cb_2 = bcomp\ b_2\ f\ n;$

$m = if\ f\ then\ size\ cb_2\ else\ size\ cb_2 + n;$

$cb_1 = bcomp\ b_1\ False\ m$

$in\ cb_1\ @\ cb_2$



# Correctness of *bcomp*



# Correctness of *bcomp*

$0 \leq n \implies$   
 $bcomp\ b\ f\ n$   
 $\vdash (0, s, stk) \rightarrow^*$   
 $(size\ (bcomp\ b\ f\ n) + (if\ f = bval\ b\ s\ then\ n\ else\ 0),$   
 $s, stk)$



# 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*) = *acom* *a* @ [*STORE* *x*]

# Compiling *com*

*ccomp* :: *com*  $\Rightarrow$  *instr list*

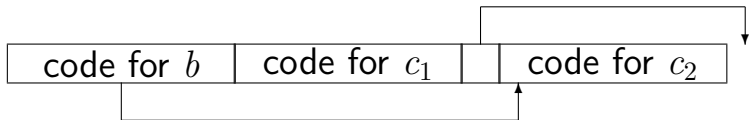
*ccomp* *SKIP* = []

*ccomp* (*x* ::= *a*) = *acom* *a* @ [*STORE* *x*]

*ccomp* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *ccomp* *c*<sub>1</sub> @ *ccomp* *c*<sub>2</sub>

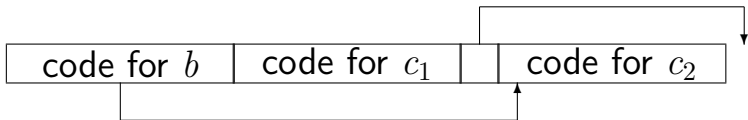
$$ccomp \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) =$$

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



$c_{comp} (IF\ b\ THEN\ c_1\ ELSE\ c_2) =$

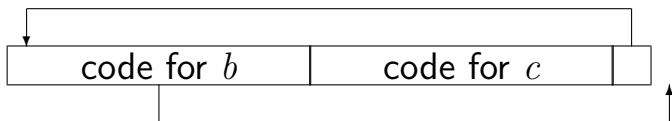
*let*  $cc_1 = c_{comp}\ c_1; cc_2 = c_{comp}\ c_2;$   
     $cb = b_{comp}\ b\ False\ (size\ cc_1 + 1)$   
*in*  $cb\ @\ cc_1\ @\ JMP\ (size\ cc_2)\ \# \ cc_2$





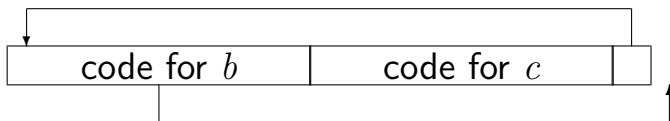
$$ccomp \ (WHILE\ b\ DO\ c) =$$

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



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

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



## 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:

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

# Correctness of *ccomp*

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

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

Proof by rule induction.

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*source code simulates compiled code?*



## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*source code simulates compiled code?*

If *ccomp* *c* with start state *s* produces result *t*,

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*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 “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*source code simulates compiled code?*

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

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*

How about “ $\impliedby$ ”:

*source code simulates compiled code?*

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

## The other direction

We have only shown “ $\implies$ ”:

*compiled code simulates source code.*



How about “ $\impliedby$ ”:

*source code simulates compiled code?*

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

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



# 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 expressions:

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

## 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*)

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

# 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*)

$$\frac{\textit{taval } a_1 \textit{ s (Iv } i_1\text{)} \quad \textit{taval } a_2 \textit{ s (Iv } i_2\text{)}}{\textit{taval (Plus } a_1 \textit{ } a_2\text{) s (Iv (} i_1 + i_2\text{))}}$$
$$\frac{\textit{taval } a_1 \textit{ s (Rv } r_1\text{)} \quad \textit{taval } a_2 \textit{ s (Rv } r_2\text{)}}{\textit{taval (Plus } a_1 \textit{ } a_2\text{) s (Rv (} r_1 + r_2\text{))}}$$

Example: evaluation of  $Plus(V \text{ "x"})(Ic \ 1)$

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

If  $s \text{ ''}x'' = Iv\ i$ :

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

If  $s \text{ ''}x'' = Iv \ i$ :

---


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



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

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

If  $s \text{''}x\text{'}' = Iv \ i$ :

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

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

If  $s \text{''}x\text{'}' = Iv\ i$ :

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

If  $s \text{''}x\text{'}' = Rv\ r$ :



Example: evaluation of  $Plus (V \text{ ''}x''^{\wedge}) (Ic \ 1)$

If  $s \text{ ''}x'' = Iv \ i$ :

$$\frac{taval (V \text{ ''}x''^{\wedge}) \ s \ (Iv \ i) \quad taval (Ic \ 1) \ s \ (Iv \ 1)}{taval (Plus (V \text{ ''}x''^{\wedge}) (Ic \ 1)) \ s \ (Iv(i + 1))}$$

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

# 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:

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

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

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

# Boolean expressions

Syntax as before. Semantics:

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

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

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

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

# Boolean expressions

Syntax as before. Semantics:

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

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

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

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

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



*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 × state ⇒ state option ⇒ 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 <sup>type</sup> error and nontermination.

Could introduce an extra error-element, e.g.

*big\_step :: com × state ⇒ state option ⇒ 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

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

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

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

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

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

# Example

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

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

## 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  $Plus (V \text{ "x"}) (V \text{ "y"})$  ?

# Type system

There are two types:

**datatype**  $ty = Ity \mid Rty$

What is the type of  $Plus (V \text{"x"}) (V \text{"y"})$  ?

Depends on the type of  $\text{"x"}$  and  $\text{"y"}$  !

# Type system

There are two types:

**datatype**  $ty = Ity \mid Rty$

What is the type of  $Plus (V \text{"x"}) (V \text{"y"})$  ?

Depends on the type of  $\text{"x"}$  and  $\text{"y"}$  !

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  $Plus (V \text{"x"}) (V \text{"y"})$  ?

Depends on the type of  $\text{"x"}$  and  $\text{"y"}$  !

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

$tyenv = vname \Rightarrow ty$  

The type of an expression is always relative to a type environment  $\Gamma$ . Standard notation:

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

Read: *In the context of  $\Gamma$ ,  $e$  has type  $\tau$*



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*

$$\begin{array}{l} \Gamma \vdash a : \tau \\ \text{tyenv} \vdash \text{aexp} : \text{ty} \end{array}$$

The rules:

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



# The type of an *aexp*

$$\begin{array}{l} \Gamma \vdash a : \tau \\ \text{tyenv} \vdash \text{aexp} : \text{ty} \end{array}$$

The rules:

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

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

# The type of an *aexp*

$$\begin{array}{l} \Gamma \vdash a : \tau \\ \text{tyenv} \vdash \text{aexp} : \text{ty} \end{array}$$

The rules:

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

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

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



## The type of an *aexp*

$$\begin{array}{l} \Gamma \vdash a : \tau \\ \text{tyenv} \vdash \text{aexp} : \text{ty} \end{array}$$

The rules:

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

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

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

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



## Example

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

where  $\Gamma \text{ ''}x'' = Ity$ .

# Well-typed *bexp*

Notation:

$$\Gamma \vdash b$$

# Well-typed *bexp*

Notation:

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

# Well-typed *bexp*

Notation:

$$\begin{array}{c} \Gamma \vdash b \\ \text{tyenv} \vdash \text{bexp} \end{array}$$

Read: *In context  $\Gamma$ ,  $b$  is well-typed.*

The rules:

$$\Gamma \vdash Bc\ v$$



The rules:

$$\Gamma \vdash Bc \ v$$

$$\Gamma \vdash b$$

$$\hline \Gamma \vdash Not \ b$$

The rules:

$$\Gamma \vdash Bc\ v$$

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

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

The rules:

$$\Gamma \vdash Bc\ v$$

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

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

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

The rules:



$$\Gamma \vdash Bc\ v$$

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

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

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

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  $\Gamma$ ,  $c$  is well-typed.*

The rules:

$$\Gamma \vdash \textit{SKIP}$$



The rules:

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

The rules:

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

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

The rules:

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

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

$$\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \textit{IF } b \textit{ THEN } c_1 \textit{ ELSE } c_2}$$

The rules:

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

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



$$\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2}$$

$$\frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \text{WHILE } b \text{ DO } c}$$

# 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*, *::=*, ...).

# Syntax-directedness

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

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

# 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  $t_1, \dots, t_n$ .

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*,  $::=$ ,  $\dots$ ).
- Well-typedness of a term  $C\ t_1 \dots t_n$  depends only on the well-typedness of its subterms  $t_1, \dots, t_n$ .



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*.

## 10 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  $s \Vdash x'' = Rv\ r$

then  $(x'' ::= Plus\ (V\ x'')\ (Ic\ 1),\ s) \not\rightarrow$



# Well-typed states

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

Remember:

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

then  $(x'' ::= Plus\ (V\ x'')\ (Ic\ 1),\ s) \not\rightarrow$

The state must be well-typed w.r.t.  $\Gamma$ .

The type of a value:

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

The type of a value:

$$\textit{type} \ (Iv \ i) = Ity$$

$$\textit{type} \ (Rv \ r) = Rty$$

Well-typed state:

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

# 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

Progress  $\wedge$  Preservation  $\implies$  Type safety

# The slogan

Progress  $\wedge$  Preservation  $\implies$  Type safety

Progress Well-typed programs do not get stuck.

# The slogan

Progress  $\wedge$  Preservation  $\implies$  Type safety

Progress Well-typed programs do not get stuck.

Preservation Well-typedness is preserved by reduction.

# The slogan

Progress  $\wedge$  Preservation  $\implies$  Type safety

**Progress** Well-typed programs do not get stuck.

**Preservation** Well-typedness is preserved by reduction.

Preservation: Well-typedness is an *invariant*.



Progress:

$$\llbracket \Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP} \rrbracket \implies \exists cs'. (c, s) \rightarrow cs'$$

Progress:

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

Preservation:

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s \rrbracket \Longrightarrow \Gamma \vdash s'$$

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c \rrbracket \Longrightarrow \Gamma \vdash c'$$



Progress:

$$\llbracket \Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP} \rrbracket \implies \exists cs'. (c, s) \rightarrow cs'$$

Preservation:

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s \rrbracket \implies \Gamma \vdash s'$$

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c \rrbracket \implies \Gamma \vdash c'$$

Type soundness:

$$\begin{aligned} &\llbracket (c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq \text{SKIP} \rrbracket \\ &\implies \exists cs''. (c', s') \rightarrow cs'' \end{aligned}$$

*bexp*

Progress:

$$\llbracket \Gamma \vdash b; \Gamma \vdash s \rrbracket \Longrightarrow \exists v. \textit{tbval } b \ s \ v$$

Progress:

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

Preservation:

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

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*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>

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

*vars* :: *aexp*  $\Rightarrow$  *vname set*

*vars* (*N* *n*) = {}

*vars* (*V* *x*) = {*x*}

*vars* (*Plus* *a*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>

*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*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>

*vars* :: *bexp*  $\Rightarrow$  *vname set*

*vars* (*Bc* *v*) = {}

*vars* (*Not* *b*) = *vars* *b*



*vars* (*And* *b*<sub>1</sub> *b*<sub>2</sub>) = *vars* *b*<sub>1</sub>  $\cup$  *vars* *b*<sub>2</sub>

*vars* (*Less* *a*<sub>1</sub> *a*<sub>2</sub>) = *vars* *a*<sub>1</sub>  $\cup$  *vars* *a*<sub>2</sub>

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.



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

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

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

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

$D A c A'$  should imply:

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



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

$D A c A'$  should imply:

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

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

$D A c A'$  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*

$$\begin{array}{c}
 D\ A\ SKIP\ A \\
 vars\ a \subseteq A \\
 \hline
 D\ A\ (x ::= a)\ (insert\ x\ A)
 \end{array}$$

$$\begin{array}{c}
D \ A \ SKIP \ A \\
\frac{vars \ a \subseteq \ A}{D \ A \ (x ::= a) \ (insert \ x \ A)} \\
\frac{D \ A_1 \ c_1 \ A_2 \quad D \ A_2 \ c_2 \ A_3}{D \ A_1 \ (c_1;; \ c_2) \ A_3}
\end{array}$$

$$\begin{array}{c}
D \ A \ SKIP \ A \\
\\
\frac{vars \ a \subseteq A}{D \ A \ (x ::= a) \ (insert \ x \ A)} \\
\\
\frac{D \ A_1 \ c_1 \ A_2 \quad D \ A_2 \ c_2 \ A_3}{D \ A_1 \ (c_1;; c_2) \ A_3} \\
\\
\frac{vars \ b \subseteq A \quad D \ A \ c_1 \ A_1 \quad D \ A \ c_2 \ A_2}{D \ A \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ (A_1 \cap A_2)}
\end{array}$$



$D \ A \ SKIP \ A$

$vars \ a \subseteq A$

---

$D \ A \ (x ::= a) \ (insert \ x \ A)$

$D \ A_1 \ c_1 \ A_2 \quad D \ A_2 \ c_2 \ A_3$

---

$D \ A_1 \ (c_1;; \ c_2) \ A_3$

$vars \ b \subseteq A \quad D \ A \ c_1 \ A_1 \quad D \ A \ c_2 \ A_2$

---

$D \ A \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ (A_1 \cap A_2)$

$vars \ b \subseteq A \quad D \ A \ c \ A'$

---

$D \ A \ (WHILE \ b \ DO \ c) \ A$

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

## Correctness of $D$

- Things can go wrong:  
execution may access uninitialized variable.  
 $\implies$  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.  
 $\implies$  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.  
 $\implies$  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 | Some 'a*



*state = vname  $\Rightarrow$  val option*

where

**datatype** *'a option = None | Some 'a*

Notation: *s(x  $\mapsto$  y)* means *s(x := Some y)*

*state = vname  $\Rightarrow$  val option*

where

**datatype** *'a option = None | 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<sub>1</sub> a<sub>2</sub>) s =  
(case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of  
  (Some i<sub>1</sub>, Some i<sub>2</sub>)  $\Rightarrow$  Some(i<sub>1</sub>+i<sub>2</sub>)  
  | \_  $\Rightarrow$  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 =*

*(case bval b s of None  $\Rightarrow$  None*

*| Some bv  $\Rightarrow$  Some ( $\neg$  bv))*

*bval :: bexp  $\Rightarrow$  state  $\Rightarrow$  bool option*

*bval (Bc v) s = Some v*

*bval (Not b) s =*

*(case bval b s of None  $\Rightarrow$  None*  
*| Some bv  $\Rightarrow$  Some ( $\neg$  bv))*

*bval (And b<sub>1</sub> b<sub>2</sub>) s =*

*(case (bval b<sub>1</sub> s, bval b<sub>2</sub> s) of*  
*(Some bv<sub>1</sub>, Some bv<sub>2</sub>)  $\Rightarrow$  Some(bv<sub>1</sub>  $\wedge$  bv<sub>2</sub>)*  
*| \_  $\Rightarrow$  None)*

*bval :: bexp  $\Rightarrow$  state  $\Rightarrow$  bool option*

*bval (Bc v) s = Some v*



*bval (Not b) s =  
(case bval b s of None  $\Rightarrow$  None  
| Some bv  $\Rightarrow$  Some ( $\neg$  bv))*

*bval (And b<sub>1</sub> b<sub>2</sub>) s =  
(case (bval b<sub>1</sub> s, bval b<sub>2</sub> s) of  
  (Some bv<sub>1</sub>, Some bv<sub>2</sub>)  $\Rightarrow$  Some(bv<sub>1</sub>  $\wedge$  bv<sub>2</sub>)  
  | \_  $\Rightarrow$  None)*

*bval (Less a<sub>1</sub> a<sub>2</sub>) s =  
(case (aval a<sub>1</sub> s, aval a<sub>2</sub> s) of  
  (Some i<sub>1</sub>, Some i<sub>2</sub>)  $\Rightarrow$  Some(i<sub>1</sub> < i<sub>2</sub>)  
  | \_  $\Rightarrow$  None)*

# Big step semantics

$(com, state) \Rightarrow state\ option$

# Big step semantics

$$(com, state) \Rightarrow state\ option$$

A small complication:

$$\frac{(c_1, s_1) \Rightarrow Some\ s_2 \quad (c_2, s_2) \Rightarrow s}{(c_1;; c_2, s_1) \Rightarrow s}$$

# Big step semantics

$$(com, state) \Rightarrow state\ option$$

A small complication:

$$\frac{(c_1, s_1) \Rightarrow Some\ s_2 \quad (c_2, s_2) \Rightarrow s}{(c_1;; c_2, s_1) \Rightarrow s}$$
$$\frac{(c_1, s_1) \Rightarrow None}{(c_1;; c_2, s_1) \Rightarrow None}$$

# Big step semantics

$$(com, state) \Rightarrow state\ option$$



A small complication:

$$\frac{(c_1, s_1) \Rightarrow Some\ s_2 \quad (c_2, s_2) \Rightarrow s}{(c_1;; c_2, s_1) \Rightarrow s}$$
$$\frac{(c_1, s_1) \Rightarrow None}{(c_1;; c_2, s_1) \Rightarrow None}$$

More convenient, because compositional:

$$(com, state\ option) \Rightarrow state\ option$$



Error (*None*) propagates:

$$(c, \textit{None}) \Rightarrow \textit{None}$$



Error (*None*) propagates:

$$(c, \textit{None}) \Rightarrow \textit{None}$$

Execution starting in (mostly) normal states (*Some s*):

$$(\textit{SKIP}, s) \Rightarrow s$$

Error (*None*) propagates:

$$(c, \textit{None}) \Rightarrow \textit{None}$$

Execution starting in (mostly) normal states (*Some s*):

$$(\textit{SKIP}, s) \Rightarrow s$$

$$\frac{\textit{aval } a \ s = \textit{Some } i}{(x ::= a, \textit{Some } s) \Rightarrow \textit{Some } (s(x \mapsto i))}$$

Error (*None*) propagates:

$$(c, \textit{None}) \Rightarrow \textit{None}$$

Execution starting in (mostly) normal states (*Some s*):

$$(\textit{SKIP}, s) \Rightarrow s$$

$$\frac{\textit{aval } a \textit{ } s = \textit{Some } i}{(x ::= a, \textit{Some } s) \Rightarrow \textit{Some } (s(x \mapsto i))}$$

$$\frac{\textit{aval } a \textit{ } s = \textit{None}}{(x ::= a, \textit{Some } s) \Rightarrow \textit{None}}$$

Error (*None*) propagates:

$$(c, \text{None}) \Rightarrow \text{None}$$

Execution starting in (mostly) normal states (*Some s*):

$$(\text{SKIP}, s) \Rightarrow s$$

$$\frac{\text{aval } a \ s = \text{Some } i}{(x ::= a, \text{Some } s) \Rightarrow \text{Some } (s(x \mapsto i))}$$

$$\frac{\text{aval } a \ s = \text{None}}{(x ::= a, \text{Some } s) \Rightarrow \text{None}}$$

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



$$\frac{bval\ b\ s = Some\ True \quad (c_1, Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ True \quad (c_1, Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \quad (c_2, Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ True \quad (c_1, Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \quad (c_2, Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = None}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, Some\ s) \Rightarrow None}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$



$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{\begin{array}{l} bval\ b\ s = Some\ True \\ (c,\ Some\ s) \Rightarrow s' \end{array} \quad (WHILE\ b\ DO\ c,\ s') \Rightarrow s''}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{\begin{array}{l} bval\ b\ s = Some\ True \\ (c,\ Some\ s) \Rightarrow s' \quad (WHILE\ b\ DO\ c,\ s') \Rightarrow s'' \end{array}}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = None}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow None}$$

## 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.*

*If  $D(\text{dom } s) \text{ c } A'$  and  $(c, \text{Some } s) \Rightarrow s'$   
then  $s' \neq \text{None}$ .*

## Correctness of $D$ w.r.t. $\Rightarrow$

We want in the end:

*Well-initialized programs cannot go wrong.*

*If  $D(\text{dom } s) \text{ c } A'$  and  $(c, \text{Some } s) \Rightarrow s'$   
then  $s' \neq \text{None}$ .*

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.*

*If  $D(\text{dom } s) \text{ c } A'$  and  $(c, \text{Some } s) \Rightarrow s'$   
then  $s' \neq \text{None}$ .*


We need to prove a generalized statement:

*If  $(c, \text{Some } s) \Rightarrow s'$  and  $D A \text{ c } A'$  and  $A \subseteq \text{dom } s$   
then  $\exists t. s' = \text{Some } t \wedge A' \subseteq \text{dom } t$ .*

## Correctness of $D$ w.r.t. $\Rightarrow$

We want in the end:

*Well-initialized programs cannot go wrong.*

 *If  $D(\text{dom } s) \subseteq A'$  and  $(c, \text{Some } s) \Rightarrow s'$   
then  $s' \neq \text{None}$ .*

We need to prove a generalized statement:

*If  $(c, \text{Some } s) \Rightarrow s'$  and  $D A \subseteq A'$  and  $A \subseteq \text{dom } s$   
then  $\exists t. s' = \text{Some } t \wedge A' \subseteq \text{dom } t$ .*



By rule induction on  $(c, \text{Some } s) \Rightarrow s'$ .

Proof needs some easy lemmas:



$vars\ a \subseteq dom\ s \implies \exists i. aval\ a\ s = Some\ i$

$vars\ b \subseteq dom\ s \implies \exists bv. bval\ b\ s = Some\ bv$

$D\ A\ c\ A' \implies A \subseteq A'$



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

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X = \text{live before } c \text{ relative to } X$



# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X =$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X = \text{live before } c \text{ relative to } X$

$L\ SKIP\ X = X$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X = \text{live before } c \text{ relative to } X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X =$



# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$



# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X =$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$   
 $vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X =$  live before  $c$  relative to  $X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$   
 $vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$

Example:

$L\ ("y" ::= V\ "z";; "x" ::= Plus\ (V\ "y")\ (V\ "z"))$   
 $\{"x"\} =$

# Liveness analysis

$L :: com \Rightarrow vname\ set \Rightarrow vname\ set$

$L\ c\ X = \text{live before } c \text{ relative to } X$

$L\ SKIP\ X = X$

$L\ (x ::= a)\ X = vars\ a \cup (X - \{x\})$

$L\ (c_1;; c_2)\ X = L\ c_1\ (L\ c_2\ X)$

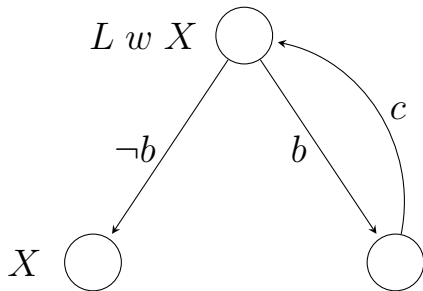
$L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$   
 $vars\ b \cup L\ c_1\ X \cup L\ c_2\ X$



Example:

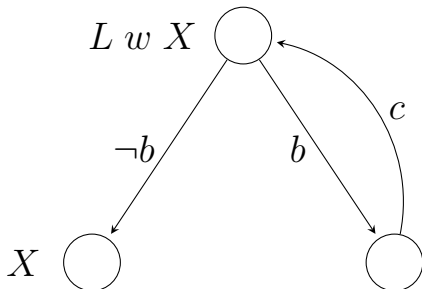
$L\ ("y" ::= V\ "z";; "x" ::= Plus\ (V\ "y")\ (V\ "z"))$   
 $\{"x"\} = \{"z"\}$

*WHILE*  $b$  *DO*  $c$



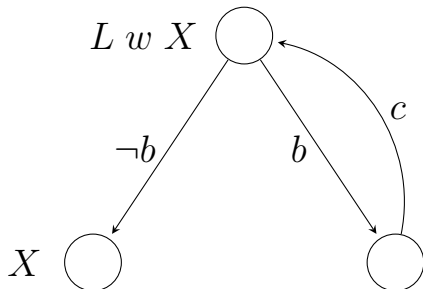


*WHILE*  $b$  *DO*  $c$



$L \ w \ X$  must satisfy

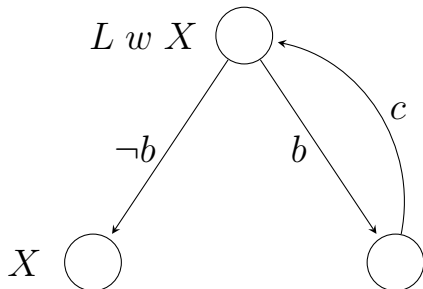
*WHILE*  $b$  *DO*  $c$



$L w X$  must satisfy

$vars\ b \subseteq L w X$  (evaluation of  $b$ )

# *WHILE* $b$ *DO* $c$

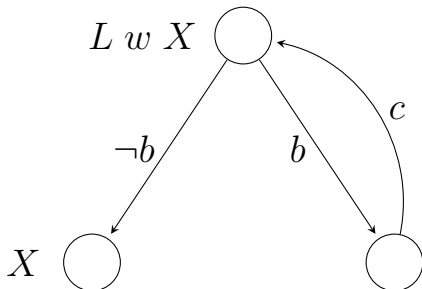


$L \ w \ X$  must satisfy

$vars \ b \quad \subseteq \quad L \ w \ X$  (evaluation of  $b$ )

$X \quad \subseteq \quad L \ w \ X$  (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

$$L ( \textit{WHILE } b \textit{ DO } c ) X = \textit{vars } b \cup X \cup L c X$$



We define

$$L ( WHILE\ b\ DO\ c )\ X = vars\ b \cup X \cup L\ c\ X$$

$\implies$

$$vars\ b \subseteq L\ w\ X \quad \checkmark$$

$$X \subseteq L\ w\ X \quad \checkmark$$

We define

$$L (WHILE\ b\ DO\ c)\ X = vars\ b \cup X \cup L\ c\ X$$

$\implies$

$$vars\ b \subseteq L\ w\ X \quad \checkmark$$

$$X \subseteq L\ w\ X \quad \checkmark$$

$$L\ c\ (L\ w\ X) \subseteq L\ w\ X \quad ?$$



$$\begin{aligned}
L \text{ SKIP } X &= X \\
L (x ::= a) X &= \text{vars } a \cup (X - \{x\}) \\
L (c_1;; c_2) X &= L c_1 (L c_2 X) \\
L (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) X &= \text{vars } b \cup L c_1 X \cup L c_2 X \\
L (\text{WHILE } b \text{ DO } c) X &= \text{vars } b \cup X \cup L c X
\end{aligned}$$

Example:

$$\begin{aligned}
L (\text{WHILE Less } (V \text{ "x"}) (V \text{ "x"}) \text{ DO "y" } ::= V \text{ "z"}) \\
\{ \text{"x"} \} &=
\end{aligned}$$



$$\begin{aligned}
L \text{ SKIP } X &= X \\
L (x ::= a) X &= \text{vars } a \cup (X - \{x\}) \\
L (c_1;; c_2) X &= L c_1 (L c_2 X) \\
L (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) X &= \text{vars } b \cup L c_1 X \cup L c_2 X \\
L (\text{WHILE } b \text{ DO } c) X &= \text{vars } b \cup X \cup L c X
\end{aligned}$$

Example:

$$\begin{aligned}
L (\text{WHILE Less } (V \text{ "x"}) (V \text{ "x"}) \text{ DO "y" } ::= V \text{ "z"}) \\
\{\text{"x"}\} &= \{\text{"x"}, \text{"z"}\}
\end{aligned}$$

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**  
if there are functions *gen* and *kill* such that

$$A \ c \ X = X - \textit{kill} \ c \cup \textit{gen} \ c$$

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**  
if there are functions *gen* and *kill* such that

$$A \ c \ X = X - \textit{kill} \ c \cup \textit{gen} \ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$X_1 \subseteq X_2 \implies A \ c \ X_1 \subseteq A \ c \ X_2$$

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**  
if there are functions *gen* and *kill* such that

$$A \ c \ X = X - \textit{kill} \ c \cup \textit{gen} \ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$\begin{aligned} X_1 \subseteq X_2 &\implies A \ c \ X_1 \subseteq A \ c \ X_2 \\ A \ c \ (X_1 \cap X_2) &= A \ c \ X_1 \cap A \ c \ X_2 \end{aligned}$$

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**  
if there are functions *gen* and *kill* such that

$$A \ c \ X = X - \textit{kill} \ c \cup \textit{gen} \ c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$\begin{aligned} X_1 \subseteq X_2 &\implies A \ c \ X_1 \subseteq A \ c \ X_2 \\ A \ c \ (X_1 \cap X_2) &= A \ c \ X_1 \cap A \ c \ X_2 \end{aligned}$$

Many standard data-flow analyses are gen/kill.

# Gen/kill analyses

A data-flow analysis  $A :: com \Rightarrow \tau \text{ set} \Rightarrow \tau \text{ set}$   
is called **gen/kill analysis**  
if there are functions *gen* and *kill* such that

$$A \ c \ X = (X - \text{kill } c) \cup \text{gen } c$$

Gen/kill analyses are extremely well-behaved, e.g.

$$\begin{aligned} X_1 \subseteq X_2 &\implies A \ c \ X_1 \subseteq A \ c \ X_2 \\ A \ c \ (X_1 \cap X_2) &= A \ c \ X_1 \cap A \ c \ X_2 \end{aligned}$$

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* =

# Liveness via gen/kill

*kill* :: *com*  $\Rightarrow$  *vname set*

*kill SKIP* =  $\{\}$

# Liveness via gen/kill

*kill* :: *com*  $\Rightarrow$  *vname set*

*kill* *SKIP* =  $\{\}$

*kill* (*x* ::= *a*) =

# Liveness via gen/kill

*kill* :: *com*  $\Rightarrow$  *vname set*

<i>kill</i> <i>SKIP</i>	=	$\{\}$
<i>kill</i> ( <i>x</i> ::= <i>a</i> )	=	$\{x\}$

# Liveness via gen/kill

*kill* :: *com*  $\Rightarrow$  *vname set*

$$\begin{aligned} \textit{kill SKIP} &= \{\} \\ \textit{kill } (x ::= a) &= \{x\} \\ \textit{kill } (c_1;; c_2) &= \end{aligned}$$

# Liveness via gen/kill

*kill* :: *com*  $\Rightarrow$  *vname set*

*kill* *SKIP*

=  $\{\}$

*kill* (*x* ::= *a*)

=  $\{x\}$

*kill* (*c*<sub>1</sub>;; *c*<sub>2</sub>)

= *kill* *c*<sub>1</sub>  $\cup$  *kill* *c*<sub>2</sub>

# Liveness via gen/kill

*kill :: com  $\Rightarrow$  vname set*

$$\begin{aligned} \textit{kill SKIP} &= \{\} \\ \textit{kill } (x ::= a) &= \{x\} \\ \textit{kill } (c_1;; c_2) &= \textit{kill } c_1 \cup \textit{kill } c_2 \\ \textit{kill } (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= \end{aligned}$$

# Liveness via gen/kill

*kill :: com  $\Rightarrow$  vname set*

*kill SKIP* =  $\{\}$

*kill (x ::= a)* =  $\{x\}$

*kill (c<sub>1</sub>;; c<sub>2</sub>)* = *kill c<sub>1</sub>*  $\cup$  *kill c<sub>2</sub>*

*kill (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>)* = *kill c<sub>1</sub>*  $\cap$  *kill c<sub>2</sub>*



# Liveness via gen/kill

*kill :: com  $\Rightarrow$  vname set*

$$\begin{aligned} \textit{kill SKIP} &= \{\} \\ \textit{kill } (x ::= a) &= \{x\} \\ \textit{kill } (c_1;; c_2) &= \textit{kill } c_1 \cup \textit{kill } c_2 \\ \textit{kill } (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= \textit{kill } c_1 \cap \textit{kill } c_2 \\ \textit{kill } (WHILE\ b\ DO\ c) &= \end{aligned}$$

# Liveness via gen/kill



*kill :: com  $\Rightarrow$  vname set*

*kill SKIP*  $= \{\}$

*kill (x ::= a)*  $= \{x\}$

*kill (c<sub>1</sub>;; c<sub>2</sub>)*  $= \textit{kill } c_1 \cup \textit{kill } c_2$

*kill (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>)*  $= \textit{kill } c_1 \cap \textit{kill } c_2$

*kill (WHILE b DO c)*  $= \{\}$

*gen :: com  $\Rightarrow$  vname set*

*gen :: com  $\Rightarrow$  vname set*

*gen SKIP =*

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen SKIP* =  $\{\}$

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) =

*gen :: com  $\Rightarrow$  vname set*

*gen SKIP           = {}*  
*gen (x ::= a)   = vars a*

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP*           =    {}

*gen* (*x* ::= *a*)    =   *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>)   =



*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> − *kill c*<sub>1</sub>)



*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> − *kill c*<sub>1</sub>)

*gen* (*IF b THEN c*<sub>1</sub> *ELSE c*<sub>2</sub>) =

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> − *kill c*<sub>1</sub>)

*gen* (*IF b THEN c*<sub>1</sub> *ELSE c*<sub>2</sub>) =  
*vars b*  $\cup$  *gen c*<sub>1</sub>  $\cup$  *gen c*<sub>2</sub>

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen* *SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> − *kill c*<sub>1</sub>)

*gen* (*IF b THEN c*<sub>1</sub> *ELSE c*<sub>2</sub>) =  
*vars b*  $\cup$  *gen c*<sub>1</sub>  $\cup$  *gen c*<sub>2</sub>

*gen* (*WHILE b DO c*) =

*gen* :: *com*  $\Rightarrow$  *vname set*

*gen SKIP* =  $\{\}$

*gen* (*x* ::= *a*) = *vars a*

*gen* (*c*<sub>1</sub>;; *c*<sub>2</sub>) = *gen c*<sub>1</sub>  $\cup$  (*gen c*<sub>2</sub> - *kill c*<sub>1</sub>)



*gen* (*IF b THEN c*<sub>1</sub> *ELSE c*<sub>2</sub>) =  
*vars b*  $\cup$  *gen c*<sub>1</sub>  $\cup$  *gen c*<sub>2</sub>

*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  $c$ .

$$L\ c\ X = \text{gen}\ c \cup (X - \text{kill}\ c)$$

Proof by induction on  $c$ .

$\Rightarrow$

$$L\ c\ (L\ w\ X) \subseteq L\ w\ X$$





## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :

## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :  
 $\textit{gen}\ \textit{SKIP} =$

## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :  
 $\textit{gen}\ \textit{SKIP} = \{\}$

## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$gen\ SKIP = \{\}$$

$$gen\ (x ::= a) =$$

## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :

$$\begin{aligned}\textit{gen}\ \textit{SKIP} &= \{\} \\ \textit{gen}\ (x ::= a) &= \{x\}\end{aligned}$$

## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$\begin{array}{ll} gen\ SKIP & = \{\} \\ gen\ (x ::= a) & = \{x\} \\ gen\ (c_1;; c_2) & = \end{array}$$

## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :

$$\begin{aligned}\textit{gen}\ \textit{SKIP} &= \{\} \\ \textit{gen}\ (x ::= a) &= \{x\} \\ \textit{gen}\ (c_1;; c_2) &= \textit{gen}\ c_1 \cup \textit{gen}\ c_2\end{aligned}$$



## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$\begin{aligned} gen\ SKIP &= \{\} \\ gen\ (x ::= a) &= \{x\} \\ gen\ (c_1;; c_2) &= gen\ c_1 \cup gen\ c_2 \\ gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= \end{aligned}$$

## 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  $A \ c \ X = X - \textit{kill } c \cup \textit{gen } c$ :

$$\begin{aligned}\textit{gen } SKIP &= \{\} \\ \textit{gen } (x ::= a) &= \{x\} \\ \textit{gen } (c_1;; c_2) &= \textit{gen } c_1 \cup \textit{gen } c_2 \\ \textit{gen } (IF \ b \ THEN \ c_1 \ ELSE \ c_2) &= \textit{gen } c_1 \cap \textit{gen } c_2\end{aligned}$$

## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$\begin{aligned} gen\ SKIP &= \{\} \\ gen\ (x ::= a) &= \{x\} \\ gen\ (c_1;; c_2) &= gen\ c_1 \cup gen\ c_2 \\ gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= gen\ c_1 \cap gen\ c_2 \\ gen\ (WHILE\ b\ DO\ c) &= \end{aligned}$$

## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$\begin{aligned} gen\ SKIP &= \{\} \\ gen\ (x ::= a) &= \{x\} \\ gen\ (c_1;; c_2) &= gen\ c_1 \cup gen\ c_2 \\ gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= gen\ c_1 \cap gen\ c_2 \\ gen\ (WHILE\ b\ DO\ c) &= \{\} \end{aligned}$$

## 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  $A\ c\ X = X - \textit{kill}\ c \cup \textit{gen}\ c$ :

$$\begin{aligned}\textit{gen}\ \textit{SKIP} &= \{\} \\ \textit{gen}\ (x ::= a) &= \{x\} \\ \textit{gen}\ (c_1;; c_2) &= \textit{gen}\ c_1 \cup \textit{gen}\ c_2 \\ \textit{gen}\ (\textit{IF}\ b\ \textit{THEN}\ c_1\ \textit{ELSE}\ c_2) &= \textit{gen}\ c_1 \cap \textit{gen}\ c_2 \\ \textit{gen}\ (\textit{WHILE}\ b\ \textit{DO}\ c) &= \{\}\end{aligned}$$

$$\textit{kill}\ c =$$

## 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  $A\ c\ X = X - kill\ c \cup gen\ c$ :

$$\begin{aligned} gen\ SKIP &= \{\} \\ gen\ (x ::= a) &= \{x\} \\ gen\ (c_1;; c_2) &= gen\ c_1 \cup gen\ c_2 \\ gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) &= gen\ c_1 \cap gen\ c_2 \\ gen\ (WHILE\ b\ DO\ c) &= \{\} \\ kill\ c &= \{\} \end{aligned}$$



## 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  $L \subset X$ .*



$(.,.) \Rightarrow .$  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  $L \subset X$ .*

Put differently:

*If two initial states agree on  $L \subset X$   
then the corresponding final states agree on  $X$ .*



# Equality on

An abbreviation:

$$f = g \text{ on } X \equiv \forall x \in X. f\ x = g\ x$$

# Equality on

An abbreviation:

$$f = g \text{ on } X \equiv \forall x \in X. f\ x = g\ x$$

Two easy theorems (in theory *Vars*):

$$s_1 = s_2 \text{ on vars } a \implies \text{aval } a\ s_1 = \text{aval } a\ s_2$$

# Equality on

An abbreviation:

$$f = g \text{ on } X \equiv \forall x \in X. f\ x = g\ x$$

Two easy theorems (in theory *Vars*):

$$s_1 = s_2 \text{ on vars } a \implies \text{aval } a\ s_1 = \text{aval } a\ s_2$$

$$s_1 = s_2 \text{ on vars } b \implies \text{bval } b\ s_1 = \text{bval } b\ s_2$$

## 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  $L\ c\ X = gen\ c \cup (X - kill\ c)$ :

$vars\ b \cup X \cup L\ c\ P \subseteq P \implies$   
 $L\ (WHILE\ b\ DO\ c)\ X \subseteq P$



## 12 Live Variable Analysis

Correctness of  $L$

Dead Variable Elimination

True Liveness

Comparisons

Bury all assignments to dead variables:

$$\textit{bury} :: \textit{com} \Rightarrow \textit{vname set} \Rightarrow \textit{com}$$

Bury all assignments to dead variables:

$$\textit{bury} :: \textit{com} \Rightarrow \textit{vname set} \Rightarrow \textit{com}$$
$$\textit{bury SKIP X} \quad =$$

Bury all assignments to dead variables:

$$\textit{bury} :: \textit{com} \Rightarrow \textit{vname set} \Rightarrow \textit{com}$$
$$\textit{bury SKIP X} = \textit{SKIP}$$

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury* *SKIP* *X* = *SKIP*

*bury* (*x* ::= *a*) *X* =



Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if x*  $\in$  *X* *then* *x ::= a* *else SKIP*

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if x*  $\in$  *X* *then* *x ::= a* *else SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* =

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if x*  $\in$  *X* *then* *x ::= a* *else SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury c<sub>1</sub> (L c<sub>2</sub> X)*;; *bury c<sub>2</sub> X*

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if x*  $\in$  *X* *then* *x ::= a* *else SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury c<sub>1</sub> (L c<sub>2</sub> X)*;; *bury c<sub>2</sub> X*

*bury (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X* =

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if* *x*  $\in$  *X* *then* *x ::= a* *else SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury c<sub>1</sub> (L c<sub>2</sub> X)*;; *bury c<sub>2</sub> X*

*bury (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X* =  
*IF b THEN bury c<sub>1</sub> X ELSE bury c<sub>2</sub> X*

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if* *x*  $\in$  *X* *then* *x ::= a* *else* *SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury* *c<sub>1</sub>* (*L c<sub>2</sub> X*);; *bury* *c<sub>2</sub>* *X*

*bury (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X* =  
*IF b THEN* *bury* *c<sub>1</sub>* *X* *ELSE* *bury* *c<sub>2</sub>* *X*

*bury (WHILE b DO c) X* =

Bury all assignments to dead variables:

*bury* :: *com*  $\Rightarrow$  *vname set*  $\Rightarrow$  *com*

*bury SKIP X* = *SKIP*

*bury (x ::= a) X* = *if*  $x \in X$  *then*  $x ::= a$  *else* *SKIP*

*bury (c<sub>1</sub>;; c<sub>2</sub>) X* = *bury*  $c_1$  (*L*  $c_2$  *X*);; *bury*  $c_2$  *X*

*bury (IF b THEN c<sub>1</sub> ELSE c<sub>2</sub>) X* =  
*IF*  $b$  *THEN* *bury*  $c_1$  *X* *ELSE* *bury*  $c_2$  *X*

*bury (WHILE b DO c) X* =  
*WHILE*  $b$  *DO* *bury*  $c$  (*L* (*WHILE*  $b$  *DO*  $c$ ) *X*)

# Correctness of *bury*



## Correctness of *bury*

$$\textit{bury } c \textit{ } UNIV \sim c$$

where *UNIV* is the set of all variables.

# Correctness of *bury*



$$\textit{bury } c \textit{ UNIV} \sim c$$

where *UNIV* is the set of all variables.

The two directions need to be proved separately.

$$(c, s) \Rightarrow s' \Longrightarrow (\textit{bury } c \textit{ UNIV}, s) \Rightarrow s'$$

$$(c, s) \Rightarrow s' \implies (\text{bury } c \text{ } UNIV, s) \Rightarrow s'$$

Follows from generalized statement:

*If  $(c, s) \Rightarrow s'$  and  $s = t$  on  $L \ c \ X$   
 then  $\exists t'. (\text{bury } c \ X, t) \Rightarrow t' \wedge s' = t'$  on  $X$ .*



$$(c, s) \Rightarrow s' \implies (\text{bury } c \text{ UNIV}, s) \Rightarrow s'$$

Follows from generalized statement:

*If  $(c, s) \Rightarrow s'$  and  $s = t$  on  $L \ c \ X$   
then  $\exists t'. (\text{bury } c \ X, t) \Rightarrow t' \wedge s' = t'$  on  $X$ .*

Proof by rule induction, like for correctness of  $L$ .

$$(bury\ c\ UNIV, s) \Rightarrow s' \Longrightarrow (c, s) \Rightarrow s'$$

$$(bury\ c\ UNIV, s) \Rightarrow s' \implies (c, s) \Rightarrow s'$$

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  $X$ .*

$$(bury\ c\ UNIV, s) \Rightarrow s' \implies (c, s) \Rightarrow s'$$

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,



$$(bury\ c\ UNIV, s) \Rightarrow s' \implies (c, s) \Rightarrow s'$$

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.

$$(bc_1;; bc_2 = bury\ c\ X) =$$

$$(\exists c_1\ c_2.$$

$$c = c_1;; c_2 \wedge$$

$$bc_2 = bury\ c_2\ X \wedge bc_1 = bury\ c_1\ (L\ c_2\ X))$$

## 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  $\tau$ , 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  $\tau$ , 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  $\tau$ , eg  $\subseteq$  on sets.

If  $f\ x \leq x$  then  $x$  is a *pre-fixpoint* (*pfp*) of  $f$ .

If  $x \leq y \implies f\ x \leq f\ y$  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  $L \ w$ :

$$\text{vars } b \cup X \cup L \ c \ (L \ w \ X) \subseteq L \ w \ X$$

This is the same as saying that  $L \ w \ X$  should be a pfp of

$$\lambda P. \text{vars } b \cup X \cup L \ c \ P$$

## 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:

$$L (x ::= a) X = \\ (if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X)$$

## 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:

$$L (x ::= a) X = \\ (if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X)$$



But then

$$L (WHILE\ b\ DO\ c) X = vars\ b \cup X \cup L\ c\ X$$

is not correct anymore.

$$\begin{aligned}
L(x ::= a) X = \\
& (if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X) \\
\\
L(WHILE\ b\ DO\ c) X = vars\ b \cup X \cup L\ c\ X
\end{aligned}$$



$$L(x ::= a) X =$$

$$(if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X)$$

$$L(WHILE\ b\ DO\ c) X = vars\ b \cup X \cup L\ c\ X$$

Let  $w = WHILE\ b\ DO\ c$

where  $b = Less\ (N\ 0)\ (V\ y)$

and  $c = y ::= V\ x;;\ x ::= V\ z$

and  $distinct\ [x,\ y,\ z]$

$L (x ::= a) X =$   
*(if  $x \in X$  then vars  $a \cup (X - \{x\})$  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  $w$  !

$L (x ::= a) X =$   
*(if  $x \in X$  then vars  $a \cup (X - \{x\})$  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  $w$  !

$y ::= V\ x \qquad x ::= V\ z\ \{y\}$

$L (x ::= a) X =$   
*(if  $x \in X$  then vars  $a \cup (X - \{x\})$  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  $w$  !

$y ::= V\ x\ \{y\}\quad x ::= V\ z\ \{y\}$

$L (x ::= a) X =$   
*(if  $x \in X$  then vars  $a \cup (X - \{x\})$  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  $w$  !

$\{x\}\ y ::= V\ x\ \{y\}\ x ::= V\ z\ \{y\}$

$$L (x ::= a) X =$$

$$(if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X)$$

$$L (WHILE\ b\ DO\ c) X = vars\ b \cup X \cup L\ c\ X$$

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  $w$  !

$$\{x\}\ y ::= V\ x\ \{y\}\ x ::= V\ z\ \{y\}$$

$$\implies L\ w\ \{y\} = \{y\} \cup \{y\} \cup \{x\}$$


$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;; x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;; x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$y ::= V\ x \qquad x ::= V\ z \quad \{x, y\}$$



$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;;\ x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$y ::= V\ x\ \{y, z\}\ x ::= V\ z\ \{x, y\}$$

$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;; x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$\{x, z\} \quad y ::= V\ x \quad \{y, z\} \quad x ::= V\ z \quad \{x, y\}$$

$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;; x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$\{x, z\} \quad y ::= V\ x \quad \{y, z\} \quad x ::= V\ z \quad \{x, y\}$$

$$L\ c\ \{x, y\} = \{x, z\}$$

$$b = \text{Less } (N\ 0) \ (V\ y)$$

$$c = y ::= V\ x;; x ::= V\ z$$

$L\ w\ \{y\} = \{x, y\}$  is not a pfp of  $L\ c$ :

$$\{x, z\} \quad y ::= V\ x \quad \{y, z\} \quad x ::= V\ z \quad \{x, y\}$$

$$L\ c\ \{x, y\} = \{x, z\} \not\subseteq \{x, y\}$$

## $L w$ for true liveness

Define  $L w X$  as the least pfp of  
 $\lambda P. \text{vars } b \cup X \cup L c P$



# Existence of least fixpoints

**Theorem** (Knaster-Tarski) Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ .

# Existence of least fixpoints

**Theorem** (Knaster-Tarski) Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ .  
If  $f$  is monotone ( $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$ )

# Existence of least fixpoints

**Theorem** (Knaster-Tarski) Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ .  
If  $f$  is monotone ( $X \subseteq Y \implies f(X) \subseteq f(Y)$ )  
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  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  is monotone then  
 $lfp(f) := \bigcap \{P \mid f(P) \subseteq P\}$  is the least pre-fixpoint.

# Proof of Knaster-Tarski

**Theorem** If  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  is monotone then  $lfp(f) := \bigcap \{P \mid f(P) \subseteq P\}$  is the least pre-fixpoint.

**Proof** •  $f(lfp f) \subseteq lfp f$

# Proof of Knaster-Tarski

**Theorem** If  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  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  $f$

# Proof of Knaster-Tarski

**Theorem** If  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  is monotone then  $\text{lfp}(f) := \bigcap \{P \mid f(P) \subseteq P\}$  is the least pre-fixpoint.

**Proof**

- $f(\text{lfp } f) \subseteq \text{lfp } f$
- $\text{lfp } f$  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  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  is monotone then  $\text{lfp}(f) := \bigcap \{P \mid f(P) \subseteq P\}$  is the least pre-fixpoint.

**Proof**

- $f(\text{lfp } f) \subseteq \text{lfp } f$
- $\text{lfp } f$  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**

- $f p \leq p \implies f p = p$

# Proof of Knaster-Tarski

**Theorem** If  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$  is monotone then  $\text{lfp}(f) := \bigcap \{P \mid f(P) \subseteq P\}$  is the least pre-fixpoint.

**Proof**

- $f(\text{lfp } f) \subseteq \text{lfp } f$
- $\text{lfp } f$  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**

- $f p \leq p \implies f p = p$
- $p$  is the least fixpoint



## Definition of $L$

$L (x ::= a) X =$   
 $(\text{if } x \in X \text{ then vars } a \cup (X - \{x\}) \text{ else } X)$

$L (WHILE\ b\ DO\ c) X = lfp\ f_w$   
**where**  $f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$

## Definition of $L$

$L (x ::= a) X =$   
 $(\text{if } x \in X \text{ then } \text{vars } a \cup (X - \{x\}) \text{ else } X)$

$L (WHILE\ b\ DO\ c) X = \text{lfp } f_w$   
where  $f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$

**Lemma**  $L\ c$  is monotone.



## Definition of $L$

$L (x ::= a) X =$   
 $(\text{if } x \in X \text{ then vars } a \cup (X - \{x\}) \text{ else } X)$

$L (WHILE\ b\ DO\ c) X = lfp\ f_w$   
where  $f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$

**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 $L$

$L (x ::= a) X =$   
 $(\text{if } x \in X \text{ then vars } a \cup (X - \{x\}) \text{ else } X)$

$L (WHILE\ b\ DO\ c) X = lfp\ f_w$   
where  $f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$

**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  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ .

# Computation of $lfp$

**Theorem** Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ . If

- $f$  is monotone:  $X \subseteq Y \implies f(X) \subseteq f(Y)$

# Computation of *lfp*

**Theorem** Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ . If

- $f$  is monotone:  $X \subseteq Y \implies f(X) \subseteq f(Y)$
- 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  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ . If

- $f$  is monotone:  $X \subseteq Y \implies f(X) \subseteq f(Y)$
- 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  $lfp(f) = f^k(\{\})$ .

# Computation of $lfp$



**Theorem** Let  $f :: \tau \text{ set} \Rightarrow \tau \text{ set}$ . If

- $f$  is monotone:  $X \subseteq Y \implies f(X) \subseteq f(Y)$
- 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  $lfp(f) = f^k(\{\})$ .

**Proof** Show  $f^i(\{\}) \subseteq p$  for any pfp  $p$  of  $f$  (by induction on  $i$ ).

## Computation of $lfp\ f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$$



## Computation of $\text{lfp } f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

## Computation of $\text{lfp } f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

## Computation of $\text{lfp } f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq$

## Computation of $\text{lfp } f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

## Computation of $lfp f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

## Computation of $lfp f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

## Computation of $lfp f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

**Corollary**  $P \subseteq V_w \implies f_w P \subseteq V_w$

## Computation of $\text{lfp } f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

**Corollary**  $P \subseteq V_w \implies f_w P \subseteq V_w$

Hence  $f_w^k \{\}$  stabilizes for some  $k \leq$



## Computation of $lfp f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

**Corollary**  $P \subseteq V_w \implies f_w P \subseteq V_w$

Hence  $f_w^k \{\}$  stabilizes for some  $k \leq |V_w|$ .

## Computation of $lfp\ f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L\ c\ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L\ c\ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

**Corollary**  $P \subseteq V_w \implies f_w\ P \subseteq V_w$

Hence  $f_w^k \{\}$  stabilizes for some  $k \leq |V_w|$ .

More precisely:  $k \leq |\text{vars } c| + 1$

## Computation of $lfp f_w$

$$f_w = (\lambda P. \text{vars } b \cup X \cup L \ c \ P)$$

The chain  $\{\} \subseteq f_w \{\} \subseteq f_w^2 \{\} \subseteq \dots$  must stabilize:

Let  $\text{vars } c$  be the variables in  $c$ .

**Lemma**  $L \ c \ X \subseteq \text{vars } c \cup X$

**Proof** by induction on  $c$

Let  $V_w = \text{vars } b \cup \text{vars } c \cup X$

**Corollary**  $P \subseteq V_w \implies f_w P \subseteq V_w$

Hence  $f_w^k \{\}$  stabilizes for some  $k \leq |V_w|$ .

More precisely:  $k \leq |\text{vars } c| + 1$

because  $f_w \{\} \supseteq \text{vars } b \cup X$ .



## Example

Let  $w = \textit{WHILE } b \textit{ DO } c$   
where  $b = \textit{Less } (N\ 0) \ (V\ y)$   
and  $c = y ::= V\ x;; x ::= V\ z$

## Example

Let  $w = \text{WHILE } b \text{ DO } c$

where  $b = \text{Less } (N\ 0) \ (V\ y)$

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

where  $b = \text{Less } (N\ 0) \ (V\ y)$

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  $w = \text{WHILE } b \text{ DO } c$   
where  $b = \text{Less } (N\ 0) \ (V\ y)$   
and  $c = y ::= V\ x;; x ::= V\ z$

To compute  $L\ w\ \{y\}$  we iterate  $f_w\ P = \{y\} \cup L\ c\ P$ :

$$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}:$$

$$\{\} \quad y ::= V\ x \quad \{\} \quad x ::= V\ z \quad \{\}$$



## Example

Let  $w = \text{WHILE } b \text{ DO } c$   
where  $b = \text{Less } (N\ 0) \ (V\ y)$   
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  $w = \text{WHILE } b \text{ DO } c$   
where  $b = \text{Less } (N\ 0) \ (V\ y)$   
and  $c = y ::= V\ x;; x ::= V\ z$

To compute  $L\ w\ \{y\}$  we iterate  $f_w\ P = \{y\} \cup L\ c\ P$ :

$$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}:$$

$$\{\} \quad y ::= V\ x \quad \{\} \quad x ::= V\ z \quad \{\}$$

$$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x, y\}:$$

$$\{x\} \quad y ::= V\ x \quad \{y\} \quad x ::= V\ z \quad \{y\}$$

## Example

Let  $w = \text{WHILE } b \text{ DO } c$   
where  $b = \text{Less } (N\ 0) \ (V\ y)$   
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  $w = \text{WHILE } b \text{ DO } c$   
where  $b = \text{Less } (N\ 0) \ (V\ y)$   
and  $c = y ::= V\ x;; x ::= V\ z$

To compute  $L\ w\ \{y\}$  we iterate  $f_w\ P = \{y\} \cup L\ c\ P$ :

$$f_w\ \{\} = \{y\} \cup L\ c\ \{\} = \{y\}:$$

$$\{\} \quad y ::= V\ x \quad \{\} \quad x ::= V\ z \quad \{\}$$

$$f_w\ \{y\} = \{y\} \cup L\ c\ \{y\} = \{x, y\}:$$

$$\{x\} \quad y ::= V\ x \quad \{y\} \quad x ::= V\ z \quad \{y\}$$

$$f_w\ \{x, y\} = \{y\} \cup L\ c\ \{x, y\} = \{x, y, z\}:$$

$$\{x, z\} \quad y ::= V\ x \quad \{y, z\} \quad x ::= V\ z \quad \{x, y\}$$



# 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`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ s)$



# Computation of *lfp* in Isabelle

From the library theory While\_Combinator:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ s)$

**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`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

$while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ s)$

**Lemma** Let  $f :: \tau\ set \Rightarrow \tau\ set$ . If

- $f$  is monotone:  $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- 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`:

$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$   
 $while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ s)$

**Lemma** Let  $f :: \tau\ set \Rightarrow \tau\ set$ . If

- $f$  is monotone:  $X \subseteq Y \Longrightarrow f(X) \subseteq f(Y)$
- and bounded by some finite set  $C$ :  
 $X \subseteq C \Longrightarrow f\ X \subseteq C$

then  $lfp\ f = while\ (\lambda X. f\ X \neq X)\ f\ \{\}$



# 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)  $\implies$   
 $\forall P. f_w \ P \subseteq g_w \ P$

## 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:

$$\begin{aligned} \text{If } Lb\ w\ X = g_w^i\ \{\}: \quad & \forall P. L\ c\ P \subseteq Lb\ c\ P \text{ (IH)} \implies \\ & \forall P. f_w\ P \subseteq g_w\ P \implies f_w(g_w^i\ \{\}) = g_w(g_w^i\ \{\}) = g_w^i\ \{\} \end{aligned}$$

# 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:

$$\begin{aligned} \text{If } Lb \ w \ X &= g_w^i \ \{\}: \quad \forall P. L \ c \ P \subseteq Lb \ c \ P \text{ (IH)} \implies \\ \forall P. f_w \ P &\subseteq g_w \ P \implies f_w(g_w^i \ \{\}) = g_w \ (g_w^i \ \{\}) = g_w^i \ \{\} \\ \implies L \ w \ X &= lfp \ f_w \subseteq g_w^i \ \{\} = Lb \ w \ X \end{aligned}$$

# 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)  $\implies$   
 $\forall P. f_w \ P \subseteq g_w \ P \implies f_w(g_w^i \ \{\}) = g_w(g_w^i \ \{\}) = g_w^i \ \{\}$   
 $\implies L \ w \ X = lfp \ f_w \subseteq g_w^i \ \{\} = Lb \ w \ X$

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)  $\implies$   
 $\forall P. f_w\ P \subseteq g_w\ P \implies f_w(g_w^i\ \{\}) = g_w(g_w^i\ \{\}) = g_w^i\ \{\}$   
 $\implies L\ w\ X = lfp\ f_w \subseteq g_w^i\ \{\} = Lb\ w\ X$

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.