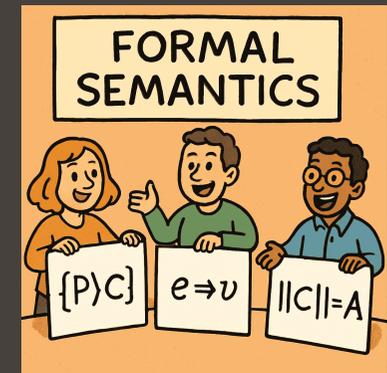


Formal Semantics



Formal Semantics

Semantics of a programming language = formally describe the behavior of programs

Applications of the semantics

- reason on correctness of program optimizations
- reason on correctness of program translations
- reason on correctness of program transformations
- verify program correctness
- ...

[formal semantics of programming languages](#) is an established and active research field in compute science

Approaches to Program Semantics

Main approaches are

- Axiomatic Semantics
- Operational Semantics
- Denotational Semantics

We briefly explain them, before studying details

Axiomatic Semantics



- Define the meaning of programs using **logical axioms**
- Deduce properties of programs using **logical inference rules**
- Usually **not all**, but only **selected properties** are considered

Prominent Example: Hoare calculus



- Triples $\{P\} C \{Q\}$ describe the effect of the programs on the environment: if precondition P holds and command C is executed, then postcondition Q holds.
- Exemplary axioms:

$$\overline{\{x = n\} x := m \{x = m\}} \quad \overline{\{\} x := 0 \{x = 0\}}$$

- An exemplary inference rule:

$$\frac{\{P\} C_1 \{Q\}, \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}$$

- An exemplary derivation

$$\frac{\overline{\{\} x := 0 \{x = 0\}}, \quad \overline{\{x = 0\} x := 10 \{x = 10\}}}{\overline{\{\} x := 0; x := 10 \{x = 10\}}}$$

Operational Semantics



- Defines **how** a program is **executed**, i.e. describes the program evaluation
- Formalisms:
 - **state transition systems**: they describe how states are transformed to eventually reach a final state (for instance, finite automata)
 - **abstract machines**: machine model to evaluate programs (for instance, a universal Turing-machine, RAM-machines, etc.)
 - **rewrite systems**: describe how programs are rewritten to obtain a value (that's what we mainly did in the lambda calculus and KFPT-languages)

Further classification:

- **small-step semantics**: evaluation requires several steps, all of them are fine-grained steps
- **big-step semantics**: evaluation in few or often only one step

Denotational Semantics



- Program is mapped to a mathematical object (the denotation of the program)
- Wide-spread approach for denotational semantics is using domains = partially ordered sets.
- Allows to use mathematics in the domain
- Very elegant but often complicated
- Example notation $\llbracket P \rrbracket \sigma$ is the denotation of program P starting in state σ

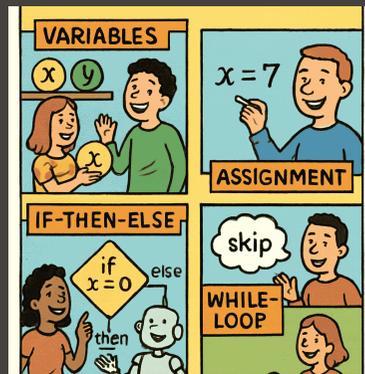
- **Contextual Semantics:** contextual equivalence as an equality notion for programs
Semantics of a program = Equivalence class of the program.
- **Transformational Semantics:** Transform the program in a program of another language and use the semantics of the target language.

Examples:

- removal of syntactic sugar
- expressing recursive supercombinators with the fixpoint operator
- Church-encoding of data
- ...

- a desired property of every semantic description:
the semantics of a **whole program** can be computed by computing the semantics of the **subprograms** and then **joining** them
- E.g. if $\langle \cdot \rangle$ computes the semantics of arithmetic expressions:
 $\langle s + t \rangle = \langle s \rangle + \langle t \rangle$ should hold

The IMP Language



The IMP Language

Definition

Arithmetic Expressions

$$\mathbf{AExp} ::= n \mid V \mid \mathbf{AExp} + \mathbf{AExp} \mid \mathbf{AExp} - \mathbf{AExp} \mid \mathbf{AExp} * \mathbf{AExp}$$

Boolean Expressions

$$\mathbf{BExp} ::= \text{True} \mid \text{False} \mid \mathbf{AExp} = \mathbf{AExp} \mid \mathbf{AExp} \leq \mathbf{AExp} \\ \mid \neg \mathbf{BExp} \mid \mathbf{BExp} \vee \mathbf{BExp} \mid \mathbf{BExp} \wedge \mathbf{BExp}$$

IMP-Programs

$$\mathbf{Cmd} ::= \text{skip} \mid V := \mathbf{AExp} \mid \mathbf{Cmd}; \mathbf{Cmd} \\ \mid \text{if } \mathbf{BExp} \text{ then } \mathbf{Cmd} \text{ else } \mathbf{Cmd} \text{ fi} \mid \text{while } \mathbf{BExp} \text{ do } \mathbf{Cmd} \text{ od}$$

where

- V generates storage locations $\in Loc$
- n, m represent arbitrary integers

Examples

$y := 2; z := 4; x := y + z$

- assigns 2 to storage location y
- assigns 4 to storage location z
- assigns 6 to storage location x

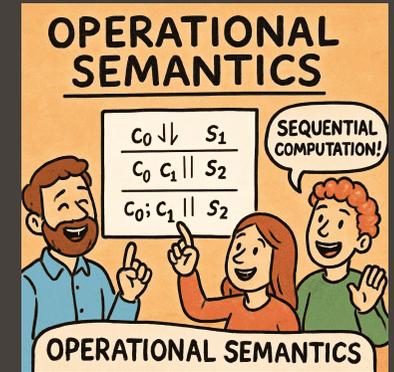
$x := 1; y := 100; \text{while } 1 \leq y \text{ do } x := x * y; y := y - 1 \text{ od}$

- computes 100!

$s := 0; i := 100; \text{while } 1 \leq i \text{ do } s := s + i * i; i := i - 1 \text{ od}$

- computes the sum $\sum_{i=0}^{100} i^2$.

Operational Semantics



Operational Semantics: States

- A **state** is a partial function $\sigma : Loc \rightarrow \mathbb{Z}$ such that $Dom(\sigma)$ is finite.
- Storage locations store numbers, but no boolean values
- Accessing not initialized storage locations is treated as runtime error
- Let Σ be the set of all states, i.e.

$$\Sigma = \{\sigma \mid \sigma : Loc \rightarrow \mathbb{Z} \wedge Dom(\sigma) \text{ is finite}\}$$

- For $\sigma \in \Sigma$ and $x \in Loc$, $\sigma(x) \in \mathbb{Z}$ is the value of storage location x , or if σ is not defined for x , $\sigma(x) = \perp$

A Big-Step Semantics

Definition (Evaluation Relation)

A **configuration** $\langle s, \sigma \rangle$ consists of a command, arithmetic expression, or boolean expression s and a state $\sigma \in \Sigma$.

We use the **evaluation relation** \Downarrow for all three kinds of configurations:

- For arithmetic expression a : $\langle a, \sigma \rangle \Downarrow n$ if a evaluates to number $n \in \mathbb{Z}$ in state σ .
- For boolean expression b : $\langle b, \sigma \rangle \Downarrow v \in \{\text{True}, \text{False}\}$ if b evaluates to v in state σ .
- For command c : $\langle c, \sigma \rangle \Downarrow \sigma'$ if c changes the state σ to state σ' .

Note:

- \Downarrow is a relation and **not necessarily** a function
- If it is a function, then the programming language is **deterministic**
- Sometimes, \Downarrow **must** be a relation:
e.g., if the language can generate random numbers

Derivation Rules

Axioms and derivation rules for the big-step semantics are written as

$$\frac{\text{premises}}{\text{conclusion}}$$

For an axiom, the premises are empty

Rules for Evaluation of Arithmetic Expressions

The rules for evaluation of arithmetic expressions are:

$$\begin{array}{ll} \text{(AxiNum)} \frac{}{\langle n, \sigma \rangle \downarrow n} & \text{(Sum)} \frac{\langle a_1, \sigma \rangle \downarrow n_1 \quad \langle a_2, \sigma \rangle \downarrow n_2 \text{ if } n' = n_1 + n_2}{\langle a_1 + a_2, \sigma \rangle \downarrow n'} \\ \text{(AxiLoc)} \frac{}{\langle x, \sigma \rangle \downarrow \sigma(x)} \text{ if } \sigma(x) \text{ is defined} & \text{(Prod)} \frac{\langle a_1, \sigma \rangle \downarrow n_1 \quad \langle a_2, \sigma \rangle \downarrow n_2 \text{ if } n' = n_1 \cdot n_2}{\langle a_1 * a_2, \sigma \rangle \downarrow n'} \\ & \text{(Diff)} \frac{\langle a_1, \sigma \rangle \downarrow n_1 \quad \langle a_2, \sigma \rangle \downarrow n_2 \text{ if } n' = n_1 - n_2}{\langle a_1 - a_2, \sigma \rangle \downarrow n'} \end{array}$$

Rules for Evaluation of Boolean Expressions

$$\begin{array}{ll} \text{(AxiT)} \frac{}{\langle \text{True}, \sigma \rangle \downarrow \text{True}} & \text{(Eq)} \frac{\langle a_1, \sigma \rangle \downarrow n \quad \langle a_2, \sigma \rangle \downarrow m \text{ if } n = m}{\langle a_1 = a_2, \sigma \rangle \downarrow \text{True}} \\ \text{(AxiF)} \frac{}{\langle \text{False}, \sigma \rangle \downarrow \text{False}} & \text{(NEq)} \frac{\langle a_1, \sigma \rangle \downarrow n \quad \langle a_2, \sigma \rangle \downarrow m \text{ if } n \neq m}{\langle a_1 = a_2, \sigma \rangle \downarrow \text{False}} \\ \text{(Leq)} \frac{\langle a_1, \sigma \rangle \downarrow n \quad \langle a_2, \sigma \rangle \downarrow m \text{ if } n \leq m}{\langle a_1 \leq a_2, \sigma \rangle \downarrow \text{True}} & \text{(NLeq)} \frac{\langle a_1, \sigma \rangle \downarrow n \quad \langle a_2, \sigma \rangle \downarrow m \text{ if } n > m}{\langle a_1 \leq a_2, \sigma \rangle \downarrow \text{False}} \\ \text{(AndT)} \frac{\langle b_1, \sigma \rangle \downarrow \text{True} \quad \langle b_2, \sigma \rangle \downarrow \text{True}}{\langle b_1 \wedge b_2, \sigma \rangle \downarrow \text{True}} & \text{(AndF1)} \frac{\langle b_1, \sigma \rangle \downarrow \text{False}}{\langle b_1 \wedge b_2, \sigma \rangle \downarrow \text{False}} \end{array}$$

Rules for Evaluation of Boolean Expressions (cont'ed)

$$\begin{array}{ll} \text{(AndF2)} \frac{\langle b_1, \sigma \rangle \downarrow \text{True} \quad \langle b_2, \sigma \rangle \downarrow \text{False}}{\langle b_1 \wedge b_2, \sigma \rangle \downarrow \text{False}} & \text{(OrF)} \frac{\langle b_1, \sigma \rangle \downarrow \text{False} \quad \langle b_2, \sigma \rangle \downarrow \text{False}}{\langle b_1 \vee b_2, \sigma \rangle \downarrow \text{False}} \\ \text{(OrT1)} \frac{\langle b_1, \sigma \rangle \downarrow \text{True}}{\langle b_1 \vee b_2, \sigma \rangle \downarrow \text{True}} & \text{(OrT2)} \frac{\langle b_1, \sigma \rangle \downarrow \text{False} \quad \langle b_2, \sigma \rangle \downarrow \text{True}}{\langle b_1 \vee b_2, \sigma \rangle \downarrow \text{True}} \\ \text{(Not1)} \frac{\langle b, \sigma \rangle \downarrow \text{False}}{\langle \neg b, \sigma \rangle \downarrow \text{True}} & \text{(Not2)} \frac{\langle b, \sigma \rangle \downarrow \text{True}}{\langle \neg b, \sigma \rangle \downarrow \text{False}} \end{array}$$

conjunction and disjunction are evaluated “sequentially”, i.e. $\langle \text{True} \vee b, \sigma \rangle \downarrow \text{True}$ and $\langle \text{False} \wedge b, \sigma \rangle \downarrow \text{False}$ for every b , in particular when b is undefined.

Example

hsrm*

For $\sigma = \{x \mapsto 10, y \mapsto 7, z \mapsto 8\}$, we can build the derivation tree for the arithmetic expression $x \leq y + 4 \vee \text{True}$ as follows

$$\frac{\text{AxNum} \frac{\langle x, \sigma \rangle \downarrow 10}{\langle x, \sigma \rangle \downarrow 10} \quad \text{Sum} \frac{\text{AxLoc} \frac{\langle y, \sigma \rangle \downarrow 7}{\langle y, \sigma \rangle \downarrow 7} \quad \text{AxNum} \frac{\langle 4, \sigma \rangle \downarrow 4}{\langle 4, \sigma \rangle \downarrow 4}}{\langle y + 4, \sigma \rangle \downarrow 11} \quad \text{if } 11 = 7 + 4}{\langle x \leq y + 4, \sigma \rangle \downarrow \text{True}} \quad \text{if } 10 \leq 11}{\text{OrT1} \frac{\langle x \leq y + 4, \sigma \rangle \downarrow \text{True}}{\langle x \leq y + 4 \vee \text{True}, \sigma \rangle \downarrow \text{True}}}$$

The construction is done bottom-up, until the top of the tree consists of axioms and thus no more premises have to be shown.

Remarks

hsrm*

- The semantics does not prescribe an exact order of evaluation
- E.g, in $a_1 + a_2$, the semantics does not fix the order of evaluating a_1 and a_2 .
- This is a typical characteristics of a big-step semantics – it leaves some freedom in the implementation.

This could be changed, by replacing rule (Sum) by:

$$\frac{\langle a_1, \sigma \rangle \downarrow n \quad \langle n + a_2, \sigma \rangle \downarrow m}{\langle a_1 + a_2, \sigma \rangle \downarrow m} \quad \frac{\langle a_2, \sigma \rangle \downarrow n}{\langle m + a_2, \sigma \rangle \downarrow n'} \quad \text{if } n' = m + n$$

Rules for Evaluation of Commands

hsrm*

The rules for evaluation of commands have side-effects, i.e. they modify the state σ . We write $\sigma[m/x]$ for the state σ where the value of x is changed to m , i.e.

$$\sigma[m/x](y) = \begin{cases} \sigma(x) & \text{if } y \neq x \\ m & \text{if } y = x \end{cases}$$

Rules for Evaluation of Commands (Cont'd)

hsrm*

$$\text{(AxSkip)} \frac{}{\langle \text{skip}, \sigma \rangle \downarrow \sigma} \quad \text{(Asgn)} \frac{\langle a, \sigma \rangle \downarrow m}{\langle x := a, \sigma \rangle \downarrow \sigma[m/x]} \quad \text{(Seq)} \frac{\langle c_1, \sigma \rangle \downarrow \sigma' \quad \langle c_2, \sigma' \rangle \downarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \downarrow \sigma''}$$

$$\text{(IfT)} \frac{\langle b, \sigma \rangle \downarrow \text{True} \quad \langle c_1, \sigma \rangle \downarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}, \sigma \rangle \downarrow \sigma'} \quad \text{(IfF)} \frac{\langle b, \sigma \rangle \downarrow \text{False} \quad \langle c_2, \sigma \rangle \downarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}, \sigma \rangle \downarrow \sigma'}$$

$$\text{(WhileF)} \frac{\langle b, \sigma \rangle \downarrow \text{False}}{\langle \text{while } b \text{ do } c \text{ od}, \sigma \rangle \downarrow \sigma} \quad \text{(WhileT)} \frac{\langle b, \sigma \rangle \downarrow \text{True} \quad \langle c, \sigma \rangle \downarrow \sigma'}{\langle \text{while } b \text{ do } c \text{ od}, \sigma \rangle \downarrow \sigma''}$$

Evaluation is Deterministic (Cont'd)

Proposition

Let c be a command, $\sigma \in \Sigma$ and $\langle c, \sigma \rangle \downarrow \sigma_1$ and $\langle c, \sigma \rangle \downarrow \sigma_2$ then $\sigma_1 = \sigma_2$

Proof. By induction on the derivation of $\langle c, \sigma \rangle \downarrow \sigma_1$.

- Base case: 1 step. This must be (AxSkip) and $c = \text{skip}$. Then the proof is easy.
- Step: Consider the last rule applied in the derivation of $\langle c, \sigma \rangle \downarrow \sigma_1$.
 - Case (Seq). Then $c = c_1; c_2$, $\langle c_1, \sigma \rangle \downarrow \sigma'$, and $\langle c_2, \sigma' \rangle \downarrow \sigma_1$ for some σ' . For $\langle c, \sigma \rangle \downarrow \sigma_2$, the rule must also be (Seq), i.e. $\langle c_1, \sigma \rangle \downarrow \sigma''$ and $\langle c_2, \sigma'' \rangle \downarrow \sigma_2$. Apply IH to the subderivations: This shows $\sigma' = \sigma''$ and $\sigma_1 = \sigma_2$.
 - Case (WhileT): Then $\langle b, \sigma \rangle \downarrow \text{True}$, $\langle c', \sigma \rangle \downarrow \sigma'$ and $\langle \text{while } b \text{ do } c' \text{ od}, \sigma' \rangle \downarrow \sigma_1$. The previous lemma shows that $\langle b, \sigma \rangle \downarrow \text{False}$ is impossible and thus for $\langle c, \sigma \rangle \downarrow \sigma_2$ also rule (WhileT) was used, i.e. $\langle c, \sigma \rangle \downarrow \sigma''$ and $\langle \text{while } b \text{ do } c' \text{ od}, \sigma'' \rangle \downarrow \sigma_2$ for some σ'' . The IH shows that $\sigma' = \sigma''$ and thus also $\sigma_1 = \sigma_2$.
 - Cases (IfT), (IfF), (WhileF): Similar. □

Equivalence of IMP-Programs

Since evaluation is deterministic, semantics of a program is a partial function on states:

Definition

Let c be a command, then $\llbracket c \rrbracket_{eval} : \Sigma \rightarrow \Sigma$ is the partial function such that

$$\llbracket c \rrbracket_{eval} \sigma = \sigma' \text{ iff } \langle c, \sigma \rangle \downarrow \sigma'$$

There are programs such that $\llbracket c \rrbracket_{eval}$ is undefined for all states. One such program is `while True do skip od`.

Definition (Equivalence \sim on Programs)

The relation \sim is defined as $c_1 \sim c_2$ iff for all $\sigma \in \Sigma$: $\llbracket c_1 \rrbracket_{eval} \sigma = \llbracket c_2 \rrbracket_{eval} \sigma$

Note: \sim means that the input-output behavior is the same.

Example

Lemma

The equivalence

$$(\text{while } b \text{ do } c \text{ od}) \sim (\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ od else skip fi})$$

holds.

This can be shown by a case distinction:

- $\langle b, \sigma \rangle \downarrow \text{False}$
- $\langle b, \sigma \rangle \downarrow \text{True}$
- $\langle b, \sigma \rangle \downarrow v$ does not hold for any v

Remark

- Let \sim_{init} be like \sim , but with the difference, that all variables are initialized with value 0 (i.e. $\sigma(x) = 0$, if σ does not define a value for x).
- $\sim \neq \sim_{init}$:
 - `if b then skip else skip fi` \sim_{init} `skip` holds for every boolean expression b
 - `if b then skip else skip fi` $\not\sim$ `skip` does not hold for all b :
e.g. if b is $x = x$ and $x \notin \sigma$

A Small-Step-Semantics of IMP



- similar to the reduction relations in the lambda-calculus
- rewrite pairs of programs and state (i.e. configurations) until a successful configuration is obtained
- defined by reduction rules and reduction contexts
- alternative definition with labeling to fix the strategy

A Small-Step-Semantics of IMP



Reduction rules \rightarrow operate on configurations $\langle t, \sigma \rangle$

$(skip)$ $\langle skip; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle$	(eqT) $\langle n = n, \sigma \rangle \rightarrow \langle True, \sigma \rangle$ if $n = m$
$(asgn)$ $\langle x := m, \sigma \rangle \rightarrow \langle skip, \sigma[m/x] \rangle$ if $m \in \mathbb{Z}$	(eqF) $\langle n = m, \sigma \rangle \rightarrow \langle False, \sigma \rangle$ if $n \neq m$
(ifT) $\langle \text{if True then } c_1 \text{ else } c_2 \text{ fi}, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle$	(orT) $\langle True \vee b, \sigma \rangle \rightarrow \langle True, \sigma \rangle$
(ifF) $\langle \text{if False then } c_1 \text{ else } c_2 \text{ fi}, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle$	(orF) $\langle False \vee v, \sigma \rangle \rightarrow \langle v, \sigma \rangle$ if $v \in \{True, False\}$
$(while)$ $\langle \text{while } b \text{ do } c \text{ od}, \sigma \rangle$ $\rightarrow \langle \text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ od else skip fi}, \sigma \rangle$	$(andF)$ $\langle False \wedge b, \sigma \rangle \rightarrow \langle False, \sigma \rangle$
(sum) $\langle n + m, \sigma \rangle \rightarrow \langle n', \sigma \rangle$ if $n' = n + m$	$(andT)$ $\langle True \wedge v, \sigma \rangle \rightarrow \langle v, \sigma \rangle$ if $v \in \{True, False\}$
$(prod)$ $\langle n * m, \sigma \rangle \rightarrow \langle n', \sigma \rangle$ if $n' = n * m$	$(notT)$ $\langle \neg True, \sigma \rangle \rightarrow \langle False, \sigma \rangle$
$(diff)$ $\langle n - m, \sigma \rangle \rightarrow \langle n', \sigma \rangle$ if $n' = n - m$	$(notF)$ $\langle \neg False, \sigma \rangle \rightarrow \langle True, \sigma \rangle$
(loc) $\langle x, \sigma \rangle \rightarrow \langle n, \sigma \rangle$ if $\sigma(x) = n$	
$(leqT)$ $\langle n \leq m, \sigma \rangle \rightarrow \langle True, \sigma \rangle$ if $n \leq m$	
$(leqF)$ $\langle n \leq m, \sigma \rangle \rightarrow \langle False, \sigma \rangle$ if $n > m$	

Reduction Contexts



Three classes of reduction contexts

$$\begin{aligned}
 R_A &::= [\cdot] \mid R_A + a \mid R_A * a \mid R_A - a \mid n + R_A \mid n * R_A \mid n - R_A \\
 R_B &::= [\cdot] \mid R_B \vee b \mid R_B \wedge b \mid \text{False} \vee R_B \mid \text{True} \wedge R_B \mid \neg R_B \\
 &\quad \mid R_A \leq a \mid n \leq R_A \mid R_A = a \mid n = R_A \\
 R_C &::= [\cdot] \mid R_C; c \mid \text{if } R_B \text{ then } c_1 \text{ else } c_2 \text{ fi} \mid x := R_A
 \end{aligned}$$

Definition

Reduction relation \xrightarrow{eval} :

If $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ then for every R_C -context: $\langle R_C[s], \sigma \rangle \xrightarrow{eval} \langle R_C[s'], \sigma' \rangle$.

We also write $\xrightarrow{eval, rule}$ where *rule* is the name of the used rule.

Note: **while** is **not** missing in R_C , since rule (*while*) rewrites the whole **while**

Alternative Definition with Labeling Algorithm



For command c , start with c^* and exhaustively apply the shifting rules:

$(c_1; c_2)^*$	$\Rightarrow (c_1^*; c_2)$
$(X := a)^*$	$\Rightarrow (X := a^*)$
$\text{if } b \text{ then } c \text{ else } c' \text{ fi}^*$	$\Rightarrow \text{if } b^* \text{ then } c \text{ else } c' \text{ fi}$
$(a_1 \oplus a_2)^*$	$\Rightarrow (a_1^* \oplus a_2)$ if $\oplus \in \{+, -, *, =, \leq\}$
$(n^* \oplus a)$	$\Rightarrow (n \oplus a^*)$ if $n \in \mathbb{Z}$ and $\oplus \in \{+, -, *, =, \leq\}$
$(b_1 \vee b_2)^*$	$\Rightarrow (b_1^* \vee b_2)$
$(b_1 \wedge b_2)^*$	$\Rightarrow (b_1^* \wedge b_2)$
$(\neg b)^*$	$\Rightarrow (\neg b^*)$
$(\text{False}^* \vee b)$	$\Rightarrow (\text{False} \vee b^*)$
$(\text{True}^* \wedge b)$	$\Rightarrow (\text{True} \wedge b^*)$

Reduction Rules with Label

hsrc*

$$\begin{array}{l}
 \langle C[\text{skip}^*; c], \sigma \rangle \xrightarrow{\text{eval, skip}} \langle C[c], \sigma \rangle \\
 \langle C[x := m^*], \sigma \rangle \xrightarrow{\text{eval, asgn}} \langle C[\text{skip}], \sigma[m/x] \rangle \text{ if } m \in \mathbf{Z} \\
 \langle C[\text{if True}^* \text{ then } c_1 \text{ else } c_2 \text{ fi}], \sigma \rangle \xrightarrow{\text{eval, ifT}} \langle C[c_1], \sigma \rangle \\
 \langle C[\text{if False}^* \text{ then } c_1 \text{ else } c_2 \text{ fi}], \sigma \rangle \xrightarrow{\text{eval, ifF}} \langle C[c_2], \sigma \rangle \\
 \langle C[\text{while } b \text{ do } c \text{ od}], \sigma \rangle^* \xrightarrow{\text{eval, while}} \langle C[\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ od else skip fi}], \sigma \rangle \\
 \langle C[n + m^*], \sigma \rangle \xrightarrow{\text{eval, sum}} \langle C[n'], \sigma \rangle \text{ if } n' = n + m \\
 \langle C[n * m^*], \sigma \rangle \xrightarrow{\text{eval, prod}} \langle C[n'], \sigma \rangle \text{ if } n' = n \cdot m \\
 \langle C[n - m^*], \sigma \rangle \xrightarrow{\text{eval, diff}} \langle C[n'], \sigma \rangle \text{ if } n' = n - m \\
 \langle C[x^*], \sigma \rangle \xrightarrow{\text{eval, loc}} \langle C[n], \sigma \rangle \text{ if } \sigma(x) = n \\
 \langle C[n \leq m^*], \sigma \rangle \xrightarrow{\text{eval, leqT}} \langle C[\text{True}], \sigma \rangle \text{ if } n \leq m \\
 \langle C[n \leq m^*], \sigma \rangle \xrightarrow{\text{eval, leqF}} \langle C[\text{False}], \sigma \rangle \text{ if } n > m \\
 \langle C[n = n^*], \sigma \rangle \xrightarrow{\text{eval, eqT}} \langle C[\text{True}], \sigma \rangle \text{ if } n = m \\
 \langle C[n = m^*], \sigma \rangle \xrightarrow{\text{eval, eqF}} \langle C[\text{False}], \sigma \rangle \text{ if } n \neq m \\
 \langle C[\text{True}^* \vee b], \sigma \rangle \xrightarrow{\text{eval, orT}} \langle C[\text{True}], \sigma \rangle \\
 \langle C[\text{False} \vee v^*], \sigma \rangle \xrightarrow{\text{eval, orF}} \langle C[v], \sigma \rangle \\
 \text{if } v \in \{\text{True}, \text{False}\} \\
 \langle C[\text{False}^* \wedge b], \sigma \rangle \xrightarrow{\text{eval, andF}} \langle C[\text{False}], \sigma \rangle \\
 \langle C[\text{True} \wedge v^*], \sigma \rangle \xrightarrow{\text{eval, andT}} \langle C[v], \sigma \rangle \\
 \text{if } v \in \{\text{True}, \text{False}\} \\
 \langle C[\neg \text{True}^*], \sigma \rangle \xrightarrow{\text{eval, notT}} \langle C[\text{False}], \sigma \rangle \\
 \langle C[\neg \text{False}^*], \sigma \rangle \xrightarrow{\text{eval, notF}} \langle C[\text{True}], \sigma \rangle
 \end{array}$$

Remarks and Notations

hsrc*

- Definitions with reduction contexts and with labeling algorithm are the same
- We write $\xrightarrow{\text{eval}, n}$ for n $\xrightarrow{\text{eval}}$ -steps, $\xrightarrow{\text{eval}, +}$ for the transitive closure and $\xrightarrow{\text{eval}, *}$ for the reflexive-transitive closure of $\xrightarrow{\text{eval}}$
- Small-step evaluation successfully stops if the configuration $\langle \text{skip}, \sigma \rangle$ for some $\sigma \in \Sigma$ is reached.
- For command c and environment σ , we write $\langle c, \sigma \rangle \downarrow_{\text{eval}} \sigma'$ iff $\langle c, \sigma \rangle \xrightarrow{\text{eval}, *} \langle \text{skip}, \sigma' \rangle$.
- There are stuck configurations: E.g. $\langle R_C[x], \sigma \rangle$ where $\sigma(x)$ is undefined.

By inspecting all syntactic cases one can verify:

Lemma

The reduction relation $\xrightarrow{\text{eval}}$ deterministic, i.e. if $\langle c, \sigma \rangle \xrightarrow{\text{eval}} \langle c', \sigma' \rangle$ and $\langle c, \sigma \rangle \xrightarrow{\text{eval}} \langle c'', \sigma'' \rangle$, then $c' = c''$ and $\sigma' = \sigma''$.

Example

hsrc*

$$\begin{array}{l}
 \langle (\text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 3\}) \rangle \\
 \xrightarrow{\text{eval, while}} \langle \text{if } \neg(x \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, loc}} \langle \text{if } \neg(3 \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, leqF}} \langle \text{if } \neg \text{False then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, notF}} \langle \text{if True then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, ifT}} \langle x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, loc}} \langle x := 3 - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, diff}} \langle x := 2; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 3\} \rangle \\
 \xrightarrow{\text{eval, asgn}} \langle \text{skip}; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, skip}} \langle \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, while}} \langle \text{if } \neg(x \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, loc}} \langle \text{if } \neg(1 \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, leqT}} \langle \text{if } \neg \text{True then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, notT}} \langle \text{if False then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, ifF}} \langle \text{skip}, \{x \mapsto 2\} \rangle
 \end{array}$$

Example (Cont'd)

hsrc*

$$\begin{array}{l}
 \xrightarrow{\text{eval, leqF}} \langle \text{if } \neg \text{False then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, notF}} \langle \text{if True then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, ifT}} \langle x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, loc}} \langle x := 2 - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, diff}} \langle x := 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 2\} \rangle \\
 \xrightarrow{\text{eval, asgn}} \langle \text{skip}; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, skip}} \langle \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, while}} \langle \text{if } \neg(x \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, loc}} \langle \text{if } \neg(1 \leq 1) \text{ then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, leqT}} \langle \text{if } \neg \text{True then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, notT}} \langle \text{if False then } x := x - 1; \text{while } \neg(x \leq 1) \text{ do } x := x - 1 \text{ od else skip fi}, \{x \mapsto 1\} \rangle \\
 \xrightarrow{\text{eval, ifF}} \langle \text{skip}, \{x \mapsto 1\} \rangle
 \end{array}$$

Equivalence of Big-Step and Reduction Semantics

hsrcm*

Lemma

Let a be an arithmetic expression and σ be a state. Then $\langle a, \sigma \rangle \downarrow m$ iff $\langle a, \sigma \rangle \xrightarrow{n} \langle m, \sigma \rangle$ by applying the reduction rules.

Proof.

The “if”-direction can be shown by induction on the derivation tree for $\langle a, \sigma \rangle \downarrow m$

The “only-if”-direction can be shown by induction on the number n of steps.

Lemma

Let b be a boolean expression and σ be a state, $v \in \{\text{true}, \text{false}\}$. Then $\langle b, \sigma \rangle \downarrow v$ iff $\langle b, \sigma \rangle \xrightarrow{n} \langle v, \sigma \rangle$ by applying the reduction rules.

Proof.

The “if”-direction can be shown by induction on the derivation tree for $\langle b, \sigma \rangle \downarrow m$.

The “only-if”-direction can be shown by induction on the number n of steps.

Equivalence of Big-Step and Reduction Semantics (2)

hsrcm*

Proposition

For IMP-commands c and states $\sigma \in \Sigma$: $\langle c, \sigma \rangle \downarrow_{eval} \sigma'$ iff $\langle c, \sigma \rangle \downarrow \sigma'$

Proof. We only show one direction:

$$\langle c, \sigma \rangle \xrightarrow{eval, n} \langle \text{skip}, \sigma' \rangle \implies \langle c, \sigma \rangle \downarrow \sigma'$$

By induction on the number n of steps.

Base case: $n = 0$. Then $c = \text{skip}$, $\sigma' = \sigma$, and (AxSkip) shows the claim.

Step: $n > 0$ and $\langle c, \sigma \rangle \xrightarrow{eval} \langle c_1, \sigma_1 \rangle \xrightarrow{eval, n-1} \langle \text{skip}, \sigma' \rangle$.

The induction hypothesis shows that $\langle c_1, \sigma_1 \rangle \downarrow \sigma'$.

Now all cases of the first reduction step (and c, c_1, σ, σ_1) have to be considered.

Equivalence of Big-Step and Reduction Semantics (3)

hsrcm*

• An $\xrightarrow{eval, skip}$ -step, $c = \text{skip}; c_1$, and $\sigma = \sigma_1$. Then $\frac{(AxSkip) \overline{\langle \text{skip} \rangle \downarrow \sigma} \quad \langle c_1, \sigma \rangle \downarrow \sigma'}{(Seq) \overline{\langle \text{skip}; c_1, \sigma \rangle \downarrow \sigma'}}$

• An $\xrightarrow{eval, asgn}$ -step. Two cases:

• $c = x := m$, $c_1 = \text{skip}$ and $\sigma_1 = \sigma[m/x] = \sigma'$. Then $\frac{(AxNum) \overline{\langle m, \sigma \rangle \downarrow m}}{(Asgn) \overline{\langle x := m, \sigma \rangle \downarrow \sigma[m/x]}}$

• $c = x := m; c'$, $c_1 = \text{skip}; c'$, and $\sigma_1 = \sigma[m/x]$. Then $\langle \text{skip}; c', \sigma_1 \rangle \xrightarrow{eval, skip} \langle c', \sigma_1 \rangle$, and the induction hypothesis can also be applied to $\langle c', \sigma_1 \rangle \xrightarrow{eval, n-2} \langle \text{skip}, \sigma' \rangle$

showing $\langle c', \sigma_1 \rangle \downarrow \sigma'$. Then $\frac{(AxNum) \overline{\langle m, \sigma \rangle \downarrow m}}{(Asgn) \overline{\langle x := m, \sigma \rangle \downarrow \sigma_1}} \quad \langle c', \sigma_1 \rangle \downarrow \sigma'$
 $\frac{(Seq) \overline{\langle x := m, \sigma \rangle \downarrow \sigma_1} \quad \langle c', \sigma_1 \rangle \downarrow \sigma'}{(Seq) \overline{\langle x := m; c_1, \sigma \rangle \downarrow \sigma'}}$

• $\xrightarrow{eval, ifT}$ - or $\xrightarrow{eval, ifF}$ -step: similar to the previous one.

Equivalence of Big-Step and Reduction Semantics (4)

hsrcm*

• $\xrightarrow{eval, while}$ -step. Two cases, we only consider one:

• $c = \text{while } b \text{ do } c' \text{ od}$, $c_1 = \text{if } b \text{ then } c'; \text{while } b \text{ do } c' \text{ od else skip fi}$, and $\sigma_1 = \sigma$.

The reduction semantics will evaluate b until it is a boolean value.

Two subcases:

• b evaluates to False. Then

$\langle \text{if } b \text{ then } c'; \text{while } b \text{ do } c' \text{ od else skip fi}, \sigma \rangle$

$\xrightarrow{eval, *} \langle \text{if False then } c'; \text{while } b \text{ do } c' \text{ od else skip fi}, \sigma \rangle$

$\xrightarrow{eval, ifF} \langle \text{skip}, \sigma_2 \rangle$.

Then also $\langle b, \sigma \rangle \xrightarrow{*} \langle \text{False}, \sigma \rangle$ and by the previous lemmas $\langle b, \sigma \rangle \downarrow \langle \text{False}, \sigma \rangle$.

This shows

$$(WhileF) \frac{\langle b, \sigma \rangle \downarrow \text{False}}{\langle \text{while } b \text{ do } c' \text{ od}, \sigma \rangle \downarrow \sigma}$$

Equivalence of Big-Step and Reduction Semantics (5)

hsrcm*

- b evaluates to True. Then

$$\begin{aligned} & \langle \text{if } b \text{ then } c'; \text{while } b \text{ do } c' \text{ od else skip fi}, \sigma \rangle \\ & \xrightarrow{\text{eval},*} \langle \text{if True then } c'; \text{while } b \text{ do } c' \text{ od else skip fi}, \sigma \rangle \\ & \xrightarrow{\text{eval},\text{ifT}} \langle c'; \text{while } b \text{ do } c' \text{ od}, \sigma \rangle \end{aligned}$$

Then also $\langle b, \sigma \rangle \xrightarrow{*} \langle \text{True}, \sigma \rangle$ and by the previous lemmas $\langle b, \sigma \rangle \downarrow \langle \text{False}, \sigma \rangle$.

By the IH: $\langle c'; \text{while } b \text{ do } c' \text{ od}, \sigma \rangle \downarrow \sigma'$ and there exists a derivation tree:

$$\text{(Seq)} \frac{\langle c', \sigma \rangle \downarrow \sigma_2 \quad \langle \text{while } b \text{ do } c' \text{ od}, \sigma_2 \rangle \downarrow \sigma'}{\langle c'; \text{while } b \text{ do } c' \text{ od}, \sigma \rangle \downarrow \sigma'}$$

Thus $\langle c', \sigma \rangle \downarrow \sigma_2$ and $\langle \text{while } b \text{ do } c' \text{ od}, \sigma_2 \rangle \downarrow \sigma'$ must hold

Putting everything together:

$$\text{(WhileT)} \frac{\langle b, \sigma \rangle \downarrow \text{True} \quad \langle c', \sigma \rangle \downarrow \sigma_2 \quad \langle \text{while } b \text{ do } c' \text{ od}, \sigma_2 \rangle \downarrow \sigma'}{\langle \text{while } b \text{ do } c' \text{ od}, \sigma \rangle \downarrow \sigma'}$$

Equivalence of Big-Step and Reduction Semantics (6)

hsrcm*

All other cases: the reduction step operates on a boolean or an arithmetic expression.

We only consider the case $c = R_c[\text{if } b \text{ then } c' \text{ else } c'' \text{ fi}]$.

Then $\langle c, \sigma \rangle \xrightarrow{\text{eval},k} \langle R_c[\text{if } v \text{ then } c' \text{ else } c'' \text{ fi}], \sigma \rangle \xrightarrow{\text{eval},n-k} \langle \text{skip}, \sigma' \rangle$ with $v \in \{\text{True}, \text{False}\}$ and $k \geq 1$.

Then also $\langle b, \sigma \rangle \xrightarrow{k} \langle v, \sigma \rangle$, and the previous lemmas show $\langle b, \sigma \rangle \downarrow v$.

We only consider the case $v = \text{True}$: Then

$$\langle R_c[\text{if } v \text{ then } c' \text{ else } c'' \text{ fi}], \sigma \rangle \xrightarrow{\text{eval}} \langle R_c[c'], \sigma \rangle \xrightarrow{\text{eval},n-k-1} \langle \text{skip}, \sigma' \rangle$$

Since $k > 0$ the IH applied to $\langle R_c[c'], \sigma \rangle \xrightarrow{\text{eval},n-k-1} \langle \text{skip}, \sigma' \rangle$ shows $\langle R_c[c'], \sigma \rangle \downarrow \sigma'$.

...

Equivalence of Big-Step and Reduction Semantics (7)

hsrcm*

...

If $R_c = [\cdot]$, then this shows

$$\text{(IFT)} \frac{\langle b, \sigma \rangle \downarrow \text{True} \quad \langle c', \sigma \rangle \downarrow \sigma'}{\langle \text{if } b \text{ then } c' \text{ else } c'' \text{ fi}, \sigma \rangle \downarrow \sigma'}$$

If $R_c = [\cdot]; c_0$ then $\langle R_c[c'], \sigma \rangle \downarrow \sigma'$ implies $\langle c', \sigma \rangle \downarrow \sigma_0$ and $\langle c_0, \sigma_0 \rangle \downarrow \sigma'$ for some σ_0 .

$$\text{(Seq)} \frac{\text{(IFT)} \frac{\langle b, \sigma \rangle \downarrow \text{True} \quad \langle c', \sigma \rangle \downarrow \sigma_0}{\langle \text{if } b \text{ then } c' \text{ else } c'' \text{ fi}, \sigma \rangle \downarrow \sigma_0} \quad \langle c_0, \sigma_0 \rangle \downarrow \sigma'}{\langle \text{if } b \text{ then } c' \text{ else } c'' \text{ fi}; c_0, \sigma \rangle \downarrow \sigma'}$$

All other cases are similar.

Sketch of Turing Completeness of IMP

hsrcm*

- Turing completeness can be shown by simulating a Turing machine with an IMP-program
- Proof in Schoening's book: While-programs and Goto-programs compute the same functions, and Goto-programs are shown to be Turing complete
- We give a sketch of a direct proof

Sketch of Turing Completeness of IMP(Cont'd)



- Turing machine configuration wqw' : Encode w , w' and state q by numbers to a base large enough to capture the tape alphabet. Then recode as integers
- The three numbers are stored in locations $x_w, x_{w'}, x_q$ of the IMP-program.
- Operations of a Turing machine (i.e. replacing the current symbol and moving the read-/write-head) are operations on the numbers (implemented using division with reminders, subtraction, addition, and multiplication.)

Sketch of Turing Completeness of IMP(Cont'd)



Assume that $\Gamma = \{a_1, \dots, a_n\}$, $Q = \{q_1, \dots, q_m\}$ and F are final states of the TM. State transition of the TM is simulated by a single while-loop, written in pseudo-code:

```
while decode( $x_q$ )  $\notin F$  do
  if decode( $x_q$ ) =  $q_1 \wedge$  decode( $x_w$ ) =  $a_1v$  then adjust  $x_q, x_w, x'_w$  for  $\delta(q_1, a_1)$  else
  if decode( $x_q$ ) =  $q_1 \wedge$  decode( $x_w$ ) =  $a_2v$  then adjust  $x_q, x_w, x'_w$  for  $\delta(q_1, a_2)$  else
  ...
  if decode( $x_q$ ) =  $q_m \wedge$  decode( $x_w$ ) =  $a_nv$  then adjust  $x_q, x_w, x'_w$  for  $\delta(q_m, a_n)$  else
  skip
fi...fi
od
```

Abstract Machine Semantics of IMP



- operational semantics as an abstract machine
- abstract means independent from real hardware
- usually easy to implement on real hardware
- we define an abstract machine for IMP

States of the IMP Machine



The **state** of the IMP machine is a triple (E, T, S) with

- **Environment E** : maps storage locations to numbers
- **Task T** : a command or an (arithmetic or boolean) expression
- **Stack S** : Contains numbers, booleans, commands, etc.

Notation:

- $s_1; s_2; \dots; s_n$ = stack with n -elements, where s_1 is on the top
- $s_1; S$ = stack with top element s_1 and S is the remaining stack.
- $[]$ = empty stack.

Start state for program c : $(\emptyset, c, [])$.

Start state for program c in environment E : $(E, c, [])$.

Final state = any state of the form $(E, \text{skip}, [])$

Stack Entries

hsrm*

- commands c
- branches $[T : c_1, F : c_2]$ to continue the evaluation of a conditional or a while loop
- $x :=$ means that x has to be updated in the environment
- $(\oplus t)$ means that the current task evaluates the left argument of operator $\oplus \in \{+, -, *, =, \leq, \wedge, \vee\}$ where t is the right argument
- \neg to negate the result of the current task
- $(n\oplus)$ means that the right argument of $\oplus \in \{+, -, *, =, \leq\}$ is currently evaluated

Transition Relation \rightsquigarrow of the IMP Machine (1)

hsrm*

$(E, (c_1; c_2), S)$	\rightsquigarrow	$(E, c_1, c_2; S)$
$(E, x := a, S)$	\rightsquigarrow	$(E, a, x :=; S)$
$(E, n, x :=; S)$	\rightsquigarrow	$(E[n/x], \text{skip}, S)$
(E, x, S)	\rightsquigarrow	(E, n, S) if $E(x) = n$
$(E, \text{while } b \text{ do } c \text{ od}, S)$	\rightsquigarrow	$(E, b, [T : c; \text{while } b \text{ do } c \text{ od}, F : \text{skip}]; S)$
$(E, \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}, S)$	\rightsquigarrow	$(E, b, [T : c_1, F : c_2]; S)$
$(E, \text{skip}, c; S)$	\rightsquigarrow	(E, c, S)
$(E, \text{True}, [T : c_1, F : c_2]; S)$	\rightsquigarrow	(E, c_1, S)
$(E, \text{False}, [T : c_1, F : c_2]; S)$	\rightsquigarrow	(E, c_2, S)

Transition Relation \rightsquigarrow of the IMP Machine (2)

hsrm*

$(E, a_1 + a_2, S)$	\rightsquigarrow	$(E, a_1, (+a_2); S)$
$(E, n, (+a); S)$	\rightsquigarrow	$(E, a, (n+); S)$
$(E, m, (n+); S)$	\rightsquigarrow	(E, m', S) if $m' = n + m$
$(E, a_1 - a_2, S)$	\rightsquigarrow	$(E, a_1, (-a_2); S)$
$(E, n, (-a); S)$	\rightsquigarrow	$(E, a, (n-); S)$
$(E, m, (n-); S)$	\rightsquigarrow	(E, m', S) if $m' = n - m$
$(E, a_1 * a_2, S)$	\rightsquigarrow	$(E, a_1, (*a_2); S)$
$(E, n, (*a); S)$	\rightsquigarrow	$(E, a, (n*); S)$
$(E, m, (n*); S)$	\rightsquigarrow	(E, m', S) if $m' = n \cdot m$

Transition Relation \rightsquigarrow of the IMP Machine (3)

hsrm*

$(E, b_1 \wedge b_2, S)$	\rightsquigarrow	$(E, b_2, (\wedge b_2); S)$
$(E, \text{True}, (\wedge b_2); S)$	\rightsquigarrow	(E, b_2, S)
$(E, \text{False}, (\wedge b_2); S)$	\rightsquigarrow	(E, False, S)
$(E, b_1 \vee b_2, S)$	\rightsquigarrow	$(E, b_2, (\vee b_2); S)$
$(E, \text{True}, (\vee b_2); S)$	\rightsquigarrow	(E, True, S)
$(E, \text{False}, (\vee b_2); S)$	\rightsquigarrow	(E, b_2, S)
$(E, \neg b, S)$	\rightsquigarrow	$(E, b, \neg; S)$
$(E, \text{True}, \neg; S)$	\rightsquigarrow	(E, False, S)
$(E, \text{False}, \neg; S)$	\rightsquigarrow	(E, True, S)

Transition Relation \rightsquigarrow of the IMP Machine (4)

hsrcm*

$(E, a_1 = a_2, S)$	$\rightsquigarrow (E, a_1, (= a_2); S)$
$(E, n, (= a); S)$	$\rightsquigarrow (E, a, (n =); S)$
$(E, m, (n =); S)$	$\rightsquigarrow (E, \text{True}, S)$ if $m = n$
$(E, m, (n \neq); S)$	$\rightsquigarrow (E, \text{False}, S)$ if $m \neq n$
$(E, a_1 \leq a_2, S)$	$\rightsquigarrow (E, a_1, (\leq a_2); S)$
$(E, n, (\leq a); S)$	$\rightsquigarrow (E, a, (n \leq); S)$
$(E, m, (n \leq); S)$	$\rightsquigarrow (E, \text{True}, S)$ if $n \leq m$
$(E, m, (n >); S)$	$\rightsquigarrow (E, \text{False}, S)$ if $n > m$

Example

hsrcm*

$(\emptyset, x := 2; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, [])$
$\rightsquigarrow (\emptyset, x := 2, \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\emptyset, 2, x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, \text{skip}, \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, [])$
$\rightsquigarrow (\{x \mapsto 2\}, 2 \leq x, [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 2\}, 2, \leq x; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 2\}, x, 2 \leq; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 2\}, 2, 2 \leq; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 2\}, \text{True}; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 2\}, x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$

Example (Cont'd')

hsrcm*

$\rightsquigarrow (\{x \mapsto 2\}, x - 1, x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, x, -1; x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, 2, -1; x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, 1, 2 -; x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 2\}, 1, x :=; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 1\}, \text{skip}, \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od})$
$\rightsquigarrow (\{x \mapsto 1\}, \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, [])$
$\rightsquigarrow (\{x \mapsto 1\}, 2 \leq x, [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 1\}, 2, \leq x; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 1\}, x, 2 \leq; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 1\}, 1, 2 \leq; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$

Example (Cont'd')

hsrcm*

$\rightsquigarrow (\{x \mapsto 1\}, \text{False}; [T : x := x - 1; \text{while } 2 \leq x \text{ do } x := x - 1 \text{ od}, F : \text{skip}])$
$\rightsquigarrow (\{x \mapsto 1\}, \text{skip}, [])$

Definition

Let \rightsquigarrow^* be the reflexive-transitive closure of \rightsquigarrow and let \rightsquigarrow^n be n steps of \rightsquigarrow .
For a program c and an environment σ , we write

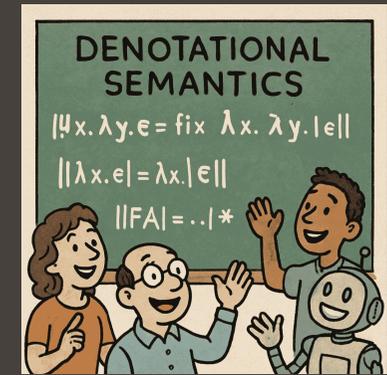
$$\langle c, \sigma \rangle \downarrow_{absm} \sigma' \text{ iff } (\sigma, c, \emptyset) \rightsquigarrow^* (\sigma', \text{skip}, [])$$

Theorem

The abstract machine is equivalent to the big-step semantics and to the reduction semantics, i.e. $\langle c, \sigma \rangle \downarrow_{absm} \sigma$ iff $\langle c, \sigma \rangle \downarrow_{eval} \sigma$ and $\langle c, \sigma \rangle \downarrow_{absm} \sigma$ iff $\langle c, \sigma \rangle \downarrow \sigma$ and

Proof Sketch. It suffices to show the claim for the reduction semantics. It is straight-forward by an induction on the number of \xrightarrow{eval} -steps for one direction, and another induction on the number of \rightsquigarrow -steps for the other direction.

Denotational Semantics



Denotational Semantics

- Goal: define a denotational semantics for IMP
- Idea of denotational semantics:
map each program construct to a mathematical object
- Since the meaning of an arithmetic or boolean expression or command depends on the state, the mathematical objects are
relations between states and values.
- Since evaluation in IMP is **deterministic**, these relations are **(partial) functions**
- Partiality is necessary, since programs may loop, etc.

Denotation

- We write \mathcal{A}, \mathcal{B} , and \mathcal{C} for the denotation of arithmetic expressions, boolean expressions, and commands.
- The syntactic argument is written in $\llbracket \cdot \rrbracket$ brackets

This means, we write:

- for arithmetic expression a , $\mathcal{A}[\llbracket a \rrbracket] : \Sigma \rightarrow \mathbb{Z}$
- for boolean expression b , $\mathcal{B}[\llbracket b \rrbracket] : \Sigma \rightarrow \{\text{True}, \text{False}\}$
- for command c , $\mathcal{C}[\llbracket c \rrbracket] : \Sigma \rightarrow \Sigma$

where the images are partial functions.

To describe partial functions, we use λ -notation and write $\lambda\sigma \in \Sigma. e$ to explicitly note that σ must be state.

- A partial function $f : M \rightarrow N$ is not necessarily defined for all elements of M (we write $f(x) = \perp$ if f is not defined for $x \in M$.)
- The **domain** of partial function f , is denoted as $Dom(f)$ ($Dom(f) = \{x \in M \mid f(x) \neq \perp\}$)
- Function f with $Dom(f) = \emptyset$ is never defined (f is called the **empty function**, and written as \emptyset)

Definition

$$\begin{aligned} \mathcal{A}[[n]] &:= \lambda\sigma \in \Sigma. n, \text{ if } n \in \mathbb{Z} \\ \mathcal{A}[[x]] &:= \lambda\sigma \in \Sigma. \sigma(x) \text{ if } x \in Loc \\ \mathcal{A}[[a_1 + a_2]] &:= \lambda\sigma \in \Sigma. (\mathcal{A}[[a_1]]\sigma) + (\mathcal{A}[[a_2]]\sigma) \\ \mathcal{A}[[a_1 - a_2]] &:= \lambda\sigma \in \Sigma. (\mathcal{A}[[a_1]]\sigma) - (\mathcal{A}[[a_2]]\sigma) \\ \mathcal{A}[[a_1 * a_2]] &:= \lambda\sigma \in \Sigma. (\mathcal{A}[[a_1]]\sigma) \cdot (\mathcal{A}[[a_2]]\sigma) \end{aligned}$$

Remarks:

- If $\sigma(x)$ is not defined, then $\sigma \notin Dom(\lambda\sigma \in \Sigma. \sigma(x))$.
- Numbers n , operators $+$, $-$ have a different meaning on the lhs and the rhs of $:=$
 - on the left hand side, they are syntax of IMP
 - on the right hand side, they are integers and mathematical operations
- $\mathcal{A}[[\cdot]]$ is also called a **semantic function**. The domains are arithmetic IMP expressions, the co-domain are sets of partial functions from states to integers

Definition

$$\begin{aligned} \mathcal{B}[[\text{True}]] &:= \lambda\sigma \in \Sigma. \text{True} \\ \mathcal{B}[[\text{False}]] &:= \lambda\sigma \in \Sigma. \text{False} \\ \mathcal{B}[[a_1 = a_2]] &:= \lambda\sigma \in \Sigma. \mathcal{A}[[a_1]]\sigma = \mathcal{A}[[a_2]]\sigma \\ \mathcal{B}[[a_1 \leq a_2]] &:= \lambda\sigma \in \Sigma. \mathcal{A}[[a_1]]\sigma \leq \mathcal{A}[[a_2]]\sigma \\ \mathcal{B}[[\neg b]] &:= \lambda\sigma \in \Sigma. \neg(\mathcal{B}[[b]]\sigma) \\ \mathcal{B}[[b_1 \vee b_2]] &:= \lambda\sigma \in \Sigma. (\mathcal{B}[[b_1]]\sigma) \vee (\mathcal{B}[[b_2]]\sigma) \\ \mathcal{B}[[b_1 \wedge b_2]] &:= \lambda\sigma \in \Sigma. (\mathcal{B}[[b_1]]\sigma) \wedge (\mathcal{B}[[b_2]]\sigma) \end{aligned}$$

Again True, False, =, \leq , \vee , \wedge , \neg on the lhs and rhs of $:=$ have a different meaning

For the denotational semantics of commands, we introduce a helper function

$$\text{cond} : (\Sigma \rightarrow \{\text{True}, \text{False}\}) \rightarrow (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma) \rightarrow \Sigma \rightarrow \Sigma$$

defined as

$$(\text{cond } f \ g_1 \ g_2) \sigma = \begin{cases} g_1 \sigma, & \text{if } f \sigma = \text{True} \\ g_2 \sigma, & \text{if } f \sigma = \text{False} \end{cases}$$

This is like an if-then-else on the semantic level.

We also define the identity:

$$\text{id}_\Sigma := \lambda\sigma \in \Sigma. \sigma$$

Denotation of Commands (without while)

hsrm*

Definition

$$\begin{aligned}
 \mathcal{C}[\text{skip}] &:= \text{id}_\Sigma \\
 \mathcal{C}[x := a] &:= \lambda\sigma \in \Sigma. \sigma[(\mathcal{A}[a]\sigma)/x] \\
 \mathcal{C}[c_0; c_1] &:= \mathcal{C}[c_1] \circ \mathcal{C}[c_0] \\
 &= \lambda\sigma \in \Sigma. (\mathcal{C}[c_1])(\mathcal{C}[c_0]\sigma) \\
 \mathcal{C}[\text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi}] &:= \lambda\sigma \in \Sigma. (\text{cond } (\mathcal{B}[b]) (\mathcal{C}[c_0]) (\mathcal{C}[c_1])) \sigma
 \end{aligned}$$

Note that $\sigma \notin \text{Dom}(\mathcal{C}[x := a])$ if $(\mathcal{A}[a]\sigma)$ is undefined.

Denotation of While

hsrm*

Defining the denotation of while is **not** straight-forward

A first approach is to use the equivalence

$$\text{while } b \text{ do } c_0 \text{ od} \sim \text{if } b \text{ then } c_0; \text{while } b \text{ do } c_0 \text{ od else skip fi}$$

This results in

$$\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}] := \lambda\sigma \in \Sigma. (\text{cond } (\mathcal{B}[b]) (\mathcal{C}[c_0; \text{while } b \text{ do } c_0 \text{ od}]) \text{id}_\Sigma) \sigma$$

which can be simplified to

$$\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}] := \text{cond } (\mathcal{B}[b]) (\mathcal{C}[c_0; \text{while } b \text{ do } c_0 \text{ od}]) \text{id}_\Sigma$$

Computing the denotation for the sequence $c_0; \text{while } b \text{ do } c_0 \text{ od}$:

$$\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}] := \text{cond } (\mathcal{B}[b]) ((\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}]) \circ (\mathcal{C}[c_0])) \text{id}_\Sigma$$

➔ the lhs $\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}]$ of the defining equation occurs in the rhs.

➔ This is a circular description and not a well-formed definition!

Denotation of While (Cont'd)

hsrm*

Use the “circular description” to find the definition:

$$\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}] = \text{cond } (\mathcal{B}[b]) ((\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}]) \circ (\mathcal{C}[c_0])) \text{id}_\Sigma$$

Let $\varphi = \mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}]$, then we can write

$$\varphi = \text{cond } (\mathcal{B}[b]) (\varphi \circ (\mathcal{C}[c_0])) \text{id}_\Sigma$$

Let Γ be the function that does the computation on φ on the rhs:

$$\Gamma = \lambda u \in (\Sigma \rightarrow \Sigma). \text{cond } (\mathcal{B}[b]) (u \circ (\mathcal{C}[c_0])) \text{id}_\Sigma$$

Note that $\Gamma : (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)$ takes a function of type $\Sigma \rightarrow \Sigma$ and returns a function of type $\Sigma \rightarrow \Sigma$.

Using Γ , the equation becomes

$$\varphi = \Gamma(\varphi)$$

Hence, φ is a **fixpoint** of Γ – the denotation of $\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}]$ is a fixpoint of Γ !

Denotation of While (Cont'd)

hsrm*

We use the **least fixpoint** of Γ , and omit the proof that it **exists** and that it **can always be constructed** (see literature)

Let us write $\text{Fix}(\Gamma)$ for least fixpoint of Γ .

Definition (Denotation of while)

$$\mathcal{C}[\text{while } b \text{ do } c_0 \text{ od}] := \text{Fix}(\Gamma)$$

$$\text{where } \Gamma := \lambda u : \Sigma \rightarrow \Sigma. \text{cond } (\mathcal{B}[b]) (u \circ (\mathcal{C}[c_0])) \text{id}_\Sigma$$

But: How can we compute the fixpoint?

Computing the Fixpoint

hsrm*

An idea to compute the least fixpoint:

- compute the partial functions $F_n \llbracket \text{while } b \text{ do } c_0 \text{ od} \rrbracket$ that represent the denotation of `while b do c0 od`
 - where **only n iterations are allowed**
 - for states σ that require more than n iterations, F_n is undefined
- the denotation $\mathcal{C} \llbracket \text{while } b \text{ do } c_0 \text{ od} \rrbracket$ is the **union** of all functions $F_n \llbracket \text{while } b \text{ do } c_0 \text{ od} \rrbracket$

Computing the Fixpoint (Cont'd)

hsrm*

- For $n = 0$: function $F_0 \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket$ is only defined for states σ where **no iteration of the loop** is necessary, i.e. states σ with $\mathcal{B} \llbracket b \rrbracket \sigma = \text{False}$
This shows

$$F_0 \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket = \{ \sigma \mapsto \sigma \mid \mathcal{B} \llbracket b \rrbracket (\sigma) = \text{False} \}$$

- For $n = 1$: function $F_1 \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket$ is defined for states σ where **at most 1 iteration of the loop** is necessary:

$$F_1 \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket = \{ \sigma \mapsto \sigma \mid \mathcal{B} \llbracket b \rrbracket (\sigma) = \text{False} \} \cup \{ \sigma \mapsto \sigma' \mid \mathcal{B} \llbracket b \rrbracket (\sigma) = \text{True}, \mathcal{C} \llbracket c \rrbracket (\sigma) = \sigma' \text{ and } \mathcal{B} \llbracket b \rrbracket (\sigma') = \text{False} \}$$

Computing the Fixpoint (Cont'd)

hsrm*

- general case, $n \geq 1$:

$$F_n \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket = F_{n-1} \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket \cup \{ \sigma \mapsto \sigma' \mid F_{n-1} \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket (\sigma) = \sigma', \mathcal{B} \llbracket b \rrbracket (\sigma') = \text{True}, \mathcal{C} \llbracket c \rrbracket (\sigma') = \sigma'', \text{ and } \mathcal{B} \llbracket b \rrbracket (\sigma'') = \text{False} \}$$

Since

$$F_i \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket \subseteq F_{i+1} \llbracket \text{while } b \text{ do } c \text{ od} \rrbracket$$

for all $i \in \mathbb{N}_0$, the infinite union can be built:

$$\mathcal{C} \llbracket \text{while } b \text{ do } c_0 \text{ od} \rrbracket = \bigcup_{n \in \mathbb{N}_0} F_n \llbracket \text{while } b \text{ do } c_0 \text{ od} \rrbracket$$

It can be shown that this union is a fixpoint of Γ and that it is the least fixpoint.

Computing the Fixpoint: Iterative Construction

hsrm*

Approach:

- start with the “smallest” function and then iteratively apply Γ and union all results
- the “smallest” function is the empty function \emptyset which is undefined for all states.
- Write φ_i for the i -fold application of Γ to \emptyset , i.e.

$$\varphi_0 = \emptyset \text{ and } \varphi_i = \Gamma(\varphi_{i-1}) \text{ for } i > 0.$$

- Then

$$\text{Fix}(\Gamma) = \bigcup_{i \in \mathbb{N}_0} \varphi_i$$

- This union can be built, since the chain $\varphi_0 \subseteq \varphi_1 \subseteq \varphi_2 \subseteq \dots$ is increasing w.r.t. \subseteq .

Example

hsrm*

We compute the denotation of `while x = 0 do skip od`:

$$\begin{aligned} \mathcal{C}[\text{while } x = 0 \text{ do skip od}] &= \text{Fix}(\Gamma) \text{ where} \\ \Gamma &:= \lambda u \in \Sigma \rightarrow \Sigma. (\text{cond } (\mathcal{B}[x = 0]) (u \circ \text{id}) \text{id}) \\ &= \lambda u \in \Sigma \rightarrow \Sigma. (\text{cond } (\lambda \sigma \in \Sigma. \sigma(x) = 0) u \text{id}) \end{aligned}$$

We compute $\varphi_0, \varphi_1, \dots$

- $\varphi_0 = \emptyset$
- $\varphi_1 = \Gamma(\varphi_0) = \text{cond } (\lambda \sigma \in \Sigma. \sigma(x) = 0) \emptyset \text{id}$.
This can be expressed as $\varphi_1 = \{\sigma \mapsto \sigma \mid x \in \text{Dom}(\sigma) \text{ and } \sigma(x) \neq 0\}$
- $\varphi_2 = \text{cond } (\lambda \sigma \in \Sigma. \sigma(x) = 0) \varphi_1 \text{id}$
If $\sigma(x) \neq 0$, then it is `id` and otherwise it is φ_1 . This can be expressed as

$$\varphi_2 = \{\sigma \mapsto \sigma \mid x \in \text{Dom}(\sigma) \text{ and } \sigma(x) \neq 0\} \cup \{\sigma \mapsto \varphi_1 \sigma \mid \sigma(x) = 0\}$$

But $\{\sigma \mapsto \varphi_1 \sigma \mid \sigma(x) = 0\} = \emptyset$, since $\varphi_1 \sigma$ is undefined for $\sigma(x) = 0$.

This shows $\varphi_2 = \varphi_1$.

Example (Cont'd)

hsrm*

- Since $\varphi_2 = \varphi_1$, we also have $\varphi_i = \varphi_1$ for all $i \geq 1$
- thus $\text{Fix}(\Gamma) = \varphi_1 = \{\sigma \mapsto \sigma \mid x \in \text{Dom}(\sigma) \text{ and } \sigma(x) \neq 0\}$.
- matches the intuition that the program terminates (with unchanged state), if x is defined and $x \neq 0$ holds in the initial state

A Second Example

hsrm*

Our goal is to compute:

$$\mathcal{C}[\text{while } 1 \leq X \text{ do } Y := Y * 2; X := X - 1 \text{ od}]$$

We make some subcalculations:

- $\mathcal{B}[1 \leq X]$
- $\mathcal{C}[Y := Y * 2; X := X - 1]$

A Second Example (2)

hsrm*

$$\begin{aligned} \mathcal{B}[1 \leq X] &= \lambda \sigma \in \Sigma. \mathcal{A}[1] \sigma \leq \mathcal{A}[X] \sigma \\ &= \lambda \sigma \in \Sigma. (\lambda \sigma \in \Sigma. 1) \sigma \leq (\lambda \sigma \in \Sigma. \sigma(X)) \sigma \\ &= \lambda \sigma \in \Sigma. 1 \leq \sigma(X) \end{aligned}$$

A Second Example (3)

hsrm*

$$\begin{aligned}
 & \mathcal{C}[\![Y := Y * 2; X : X - 1]\!] \\
 &= \lambda\sigma \in \Sigma. \mathcal{C}[\![X : X - 1]\!](\mathcal{C}[\![Y := Y * 2]\!]\sigma) \\
 &= \lambda\sigma \in \Sigma. (\lambda\sigma \in \Sigma. \sigma[\mathcal{A}[\![X - 1]\!]\sigma/X]((\lambda\sigma \in \Sigma. \sigma[\mathcal{A}[\![Y * 2]\!]\sigma/Y])) \\
 &= \lambda\sigma \in \Sigma. (\lambda\sigma \in \Sigma. \sigma[\mathcal{A}[\![X - 1]\!]\sigma/X](\sigma[\mathcal{A}[\![Y * 2]\!]\sigma/Y])) \\
 &= \lambda\sigma \in \Sigma. (\lambda\sigma \in \Sigma. \sigma[\sigma(X) - 1/X](\sigma[\sigma(Y) * 2/Y])) \\
 &= \lambda\sigma \in \Sigma. \sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X]
 \end{aligned}$$

A Second Example (4)

hsrm*

$$\mathcal{C}[\![\text{while } 1 \leq X \text{ do } Y := Y * 2; X := X - 1 \text{ od}]\!] = \text{Fix}(\Gamma)$$

with

$$\begin{aligned}
 \Gamma &= \lambda u \in \Sigma \rightarrow \Sigma. \text{cond } \mathcal{B}[1 \leq X] (u \circ \mathcal{C}[\![Y := Y * 2; X : X - 1]\!] \text{ id}) \\
 &= \lambda u \in \Sigma \rightarrow \Sigma. \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (u \circ (\lambda\sigma \in \Sigma. \sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id} \\
 &= \lambda u \in \Sigma \rightarrow \Sigma. \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (\lambda\sigma \in \Sigma. u(\sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id}
 \end{aligned}$$

$$\text{Fix}(\Gamma) = \bigcup \varphi_i \text{ with } \varphi_i = \varphi^i(\emptyset)$$

A Second Example (5)

hsrm*

$$\begin{aligned}
 \Gamma &= \lambda u \in \Sigma \rightarrow \Sigma. \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (\lambda\sigma \in \Sigma. u(\sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id} \\
 \varphi_0 &= \emptyset = \lambda\sigma \in \Sigma. \perp \\
 \varphi_1 &= \Gamma(\varphi_0) = \Gamma(\emptyset) \\
 &= \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (\lambda\sigma \in \Sigma. (\lambda\sigma \in \Sigma. \perp) (\sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id} \\
 &= \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) (\lambda\sigma \in \Sigma. \perp) \text{ id} \\
 &= \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \emptyset \text{ id} \\
 &= \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\}
 \end{aligned}$$

A Second Example (6)

hsrm*

$$\begin{aligned}
 \varphi_2 &= \Gamma(\varphi_1) \\
 &= \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (\lambda\sigma \in \Sigma. (\text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \emptyset \text{ id}) (\sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id} \\
 &= \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\} \cup \{\sigma \rightarrow \sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X] \mid 1 \leq \sigma(X) \wedge 1 > \sigma(X) - 1\} \\
 &= \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\} \cup \{\sigma \rightarrow \sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X] \mid 1 \leq \sigma(X) \wedge 2 > \sigma(X)\} \\
 \varphi_3 &= \Gamma(\varphi_2) \\
 &= \text{cond } (\lambda\sigma \in \Sigma. 1 \leq \sigma(X)) \\
 &\quad (\lambda\sigma \in \Sigma. (\varphi_2 (\sigma[\sigma(Y) * 2/Y, \sigma(X) - 1/X])) \\
 &\quad \text{id} \\
 &= \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\} \cup \{\sigma \rightarrow \sigma[\sigma(Y) * 2 * 2/Y, \sigma(X) - 1 - 1/X] \mid 1 \leq \sigma(X) \wedge 1 > \sigma(X) - 1 - 1\} \\
 &= \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\} \cup \{\sigma \rightarrow \sigma[\sigma(Y) * 2^2/Y, \sigma(X) - 2/X] \mid 1 \leq \sigma(X) \wedge 3 > \sigma(X)\}
 \end{aligned}$$

We could proceed with $\varphi_4, \varphi_5, \dots$ but this will not stop.

A Second Example (7)

hsrcm*

Solution: guess the loop-invariant:

$$\varphi_n = \{\sigma \rightarrow \sigma \mid 1 > \sigma(X)\} \cup \{\sigma \rightarrow \sigma[\sigma(Y) * 2^{\sigma(X)}/Y, 0/X] \mid 1 \leq \sigma(X) \text{ and } n > \sigma(X)\}$$

Then prove the invariant by induction on n (We skip this)

$\mathcal{C}[\text{while } 1 \leq X \text{ do } Y := Y * 2; X := X - 1 \text{ od}] = \bigcup \varphi_n = f$ with

$$f\sigma = \begin{cases} \sigma, & \text{if } \sigma(X) \leq 1 \\ \sigma[Y * 2^{\sigma(X)}/Y, 0/X], & \text{if } \sigma(X) > 1 \\ \perp, & \text{if } \sigma(X) = \perp \text{ or } \sigma(Y) = \perp \end{cases}$$

Equivalence: Big-Step & Denotational Semantics

hsrcm*

Lemma 6.4.1

For all arithmetic expressions a and states $\sigma \in \Sigma$: $\mathcal{A}[[a]]\sigma = n$ iff $\langle a, \sigma \rangle \downarrow n$

Proof. We use structural induction on a .

- Base case $a = n$: Then $\mathcal{A}[[n]]\sigma = n$ and $\langle n, \sigma \rangle \downarrow n$ by axiom (AxNum).
- Base case $a = x$: Then $\mathcal{A}[[x]]\sigma = \sigma(x)$ and $\langle x, \sigma \rangle \downarrow \sigma(x)$ by axiom (AxLoc).
- Step case $a = a_1 + a_2$:

$$\begin{aligned} \mathcal{A}[[a_1 + a_2]]\sigma = n &= (\mathcal{A}[[a_1]]\sigma) + (\mathcal{A}[[a_2]]\sigma) \\ \text{iff } \mathcal{A}[[a_1]]\sigma = n_1 \text{ and } \mathcal{A}[[a_2]]\sigma \text{ and } n &= n_1 + n_2 \\ \text{iff } \langle a_1, \sigma \rangle \downarrow n_1 \text{ and } \langle a_2, \sigma \rangle \downarrow n_2 & \quad \text{(by the IH)} \\ \text{iff } \langle a_1 + a_2, \sigma \rangle \downarrow n & \quad \text{(rule (Sum))} \end{aligned}$$

- The cases $a = a_1 - a_2$ and $a = a_1 * a_2$ are analogous. \square

Equivalence: Big-Step & Denotational Semantics (2)

hsrcm*

Lemma 6.4.2

For all boolean expressions b , states $\sigma \in \Sigma$, and $v \in \{\text{True}, \text{False}\}$:

$$\mathcal{B}[[b]]\sigma = v \text{ iff } \langle a, \sigma \rangle \downarrow v$$

Proof (Sketch).

- The proof is by structural induction on b .
- It is similar to the previous proof.
- It uses Lemma 6.4.1 if b requires the value of an arithmetic expression.

Equivalence: Big-Step & Denotational Semantics (3)

hsrcm*

Theorem

For all commands c of IMP and $\sigma \in \Sigma$: $\mathcal{C}[[c]]\sigma = \sigma'$ iff $\langle c, \sigma \rangle \downarrow \sigma'$

Proof.

- The “only-if” direction can be proved by induction on the derivation tree for $\langle c, \sigma \rangle \downarrow \sigma'$ (we omit the details)
- We show the “if”-direction by structural induction on c
- Base cases:
 - $\mathcal{C}[[\text{skip}]]\sigma = \sigma$ and $\langle \text{skip}, \sigma \rangle \downarrow \sigma$.
 - $\mathcal{C}[[x := a]]\sigma$: Assume $\mathcal{A}[[a]]\sigma = n$. Then $\mathcal{C}[[x := a]]\sigma = \sigma[n/x]$, and Lemma 6.4.1 shows $\langle a, \sigma \rangle \downarrow n$ and thus $\langle x := a, \sigma \rangle \downarrow \sigma[n/x]$ by (Asgn).

Equivalence: Big-Step & Denotational Semantics (4)

hsrcm*

Step cases:

- $\mathcal{C}[[c_0; c_1]]\sigma = \mathcal{C}[[c_1]](\mathcal{C}[[c_0]]\sigma) = \sigma'$. Let $\sigma'' = \mathcal{C}[[c_0]]\sigma$. The IH shows $\mathcal{C}[[c_0]]\sigma = \sigma''$ implies $\langle c_0, \sigma \rangle \downarrow \sigma''$ and $\mathcal{C}[[c_1]]\sigma'' = \sigma'$ implies $\langle c_1, \sigma'' \rangle \downarrow \sigma'$.
Now rule (Seq) shows $\langle c_0; c_1, \sigma \rangle \downarrow \sigma'$.
- $\mathcal{C}[[\text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi}]]\sigma = (\text{cond } \mathcal{B}[[b]] \mathcal{C}[[c_0]] \mathcal{C}[[c_1]])\sigma = \sigma'$.
By Lemma 6.4.2 and $v \in \{\text{True}, \text{False}\}$: If $\mathcal{B}[[b]]\sigma = v$, then $\langle b, \sigma \rangle \downarrow v$.
Let $\sigma' = \begin{cases} \mathcal{C}[[c_0]], & \text{if } \mathcal{B}[[b]]\sigma = \text{True} \\ \mathcal{C}[[c_1]], & \text{if } \mathcal{B}[[b]]\sigma = \text{False} \end{cases}$
The induction hypothesis shows $\langle c_0, \sigma \rangle \downarrow \sigma'$ or $\langle c_1, \sigma \rangle \downarrow \sigma'$ resp.
Now rule (IfT) or (IfF), resp. can be applied, showing $\langle \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi}, \sigma \rangle \downarrow \sigma'$.

Equivalence: Big-Step & Denotational Semantics (5)

hsrcm*

- $\mathcal{C}[[\text{while } b \text{ do } c_0 \text{ od}]]\sigma = \text{Fix}(\Gamma)\sigma = \sigma'$ where

$$\Gamma = \lambda u \in (\Sigma \rightarrow \Sigma). \text{cond } \mathcal{B}[[b]] (u \circ \mathcal{C}[[c_0]]) \text{id}$$

Then

$$\Gamma(\varphi) = \{\sigma_1 \mapsto \sigma_1 \mid \mathcal{B}[[b]]\sigma_1 = \text{False}\} \cup \{\sigma_1 \mapsto \sigma_2 \mid \mathcal{B}[[b]]\sigma_1 = \text{True} \text{ and } (\sigma_1 \mapsto \sigma_2) \in \varphi \circ \mathcal{C}[[c_0]]\}$$

Let $\varphi_n := \Gamma^n(\emptyset)$. Then

$$\varphi_{n+1} = \{\sigma_1 \mapsto \sigma_1 \mid \mathcal{B}[[b]]\sigma_1 = \text{False}\} \cup \{\sigma_1 \mapsto \sigma_2 \mid \mathcal{B}[[b]]\sigma_1 = \text{True} \text{ and } (\sigma_1 \mapsto \sigma_2) \in \varphi_n \circ \mathcal{C}[[c_0]]\}$$

By induction on n , we show

$$(\sigma_1 \mapsto \sigma_2) \in \varphi_n \implies \langle \text{while } b \text{ do } c_0 \text{ od}, \sigma_1 \rangle \downarrow \sigma_2$$

Equivalence: Big-Step & Denotational Semantics (6)

hsrcm*

By induction on n , we show that $(\sigma_1 \mapsto \sigma_2) \in \varphi_n \implies \langle \text{while } b \text{ do } c_0 \text{ od}, \sigma_1 \rangle \downarrow \sigma_2$.

- $n = 0$: $\varphi_0 = \emptyset$. The lhs of the implication is false and the implication is true.
 - $n > 0$: Let the claim hold for n and let $(\sigma_1 \mapsto \sigma_2) \in \varphi_{n+1}$.
 - If $(\mathcal{B}[[b]]\sigma_1) = \text{False}$ and $\sigma_2 = \sigma_1$, then Lemma 6.4.2 shows $\langle b, \sigma_1 \rangle \downarrow \text{False}$.
Rule (WhileF) shows $\langle \text{while } b \text{ do } c_0 \text{ od}, \sigma_1 \rangle \downarrow \sigma_1$.
 - If $\mathcal{B}[[b]]\sigma_1 = \text{True}$, then Lemma 6.4.2 shows $\langle b, \sigma_1 \rangle \downarrow \text{True}$.
Since $\sigma_1 \mapsto \sigma_2 \in \varphi_{n+1}$, there exists σ_3 with $\mathcal{C}[[c_0]]\sigma_1 = \sigma_3$ and $(\sigma_3 \mapsto \sigma_2) \in \varphi_n$.
By the outer IH we get $\langle c_0, \sigma_1 \rangle \downarrow \sigma_3$.
By the inner IH we have $\langle \text{while } b \text{ do } c_0 \text{ od}, \sigma_3 \rangle \downarrow \sigma_2$.
Now rule (WhileT) shows $\langle \text{while } b \text{ do } (c_0; \text{while } b \text{ do } c_0 \text{ od}), \sigma_1 \rangle \downarrow \sigma_2$.
- Since $\text{Fix}(\Gamma) = \bigcup \varphi_n$, we have: $(\sigma \mapsto \sigma') \in \text{Fix}(\Gamma) \implies (\sigma \mapsto \sigma') \in \varphi_n$ for some n and thus $\langle \text{while } b \text{ do } c_0 \text{ od}, \sigma \rangle \downarrow \sigma'$. \square

Example

hsrcm*

We consider the loop $(\text{while True do skip od})$ and all semantics that we have defined.

Big-step operational semantics cannot derive any σ' such that $\langle \text{while True do skip od}, \sigma \rangle \downarrow \sigma'$ for any σ .

The small-step operational semantics has an infinite sequence of reduction steps:

$$\begin{array}{l} \langle \text{while True do skip od}, \sigma \rangle \\ \xrightarrow{\text{eval, while}} \langle \text{if True then while True do skip od else skip fi}, \sigma \rangle \\ \xrightarrow{\text{eval, ifT}} \langle \text{while True do skip od}, \sigma \rangle \\ \xrightarrow{\text{eval, while}} \dots \end{array}$$

The abstract machine has infinite sequence of transitions:

$$\begin{array}{l} (E, \text{while True do skip od}, []) \\ \rightsquigarrow (E, \text{True}, [T : \text{skip}; \text{while True do skip od}, F : \text{skip}]) \\ \rightsquigarrow (E, \text{skip}; \text{while True do skip od}, []) \\ \rightsquigarrow (E, \text{skip}, \text{while True do skip od}) \\ \rightsquigarrow (E, \text{while True do skip od}, []) \\ \rightsquigarrow \dots \end{array}$$

The denotational semantics is $\mathcal{C}[\![\text{while True do skip od}]\!] := \text{Fix}(\Gamma)$ where $\Gamma := \lambda u \in (\Sigma \rightarrow \Sigma). \text{cond}(\lambda \sigma. \text{True}) (u \circ \text{id}) \text{id}$

For computing the fixpoint, we compute $\varphi_0, \varphi_1, \dots$:

- $\varphi_0 = \emptyset$
- $\varphi_1 = \Gamma(\varphi_0) = \text{cond}(\lambda \sigma. \text{True}) (\emptyset \circ \text{id}) \text{id} = \text{cond}(\lambda \sigma. \text{True}) \emptyset \text{id} = \emptyset$ and thus $\varphi_1 = \varphi_0$

Since $\varphi_1 = \varphi_0$, this shows $\varphi_i = \varphi_0 = \emptyset$ and thus $\text{Fix}(\Gamma) = \emptyset$. Thus the denotation is the partial function that is undefined for every state σ .

Conclusion



- Different concepts and formalisms for semantics
- Operational semantics and denotational semantics
- Different styles of operational semantics
- Equivalence of the denotational and the operational semantics
- Outlook: advanced concepts for non-determinism or parallelism