Design and Selection of Sequential Programming Languages

SFWR ENG 3E03, Fall 2003

WOLFRAM KAHL

Department of Computing and Software
Faculty of Engineering
McMaster University

Lecture 1

Semantics

Syntax and Semantics

Syntax — *Shape* of PL constructs

- What are the **tokens** of the language? — **Lexical** syntax, “word level”
- How are programs built from tokens? — Mostly use **Context-Free Grammars** (CFG) or **Backus-Naur-Form** (BNF) to describe **syntax** at the “sentence level”

“Static semantics”: aspects of program structure that are checked at compile time, but cannot be captured by CFGs ($\rightarrow$ context-sensitive syntax):

– Scopes of names

– Typing

Semantics — *Meaning* of PL constructs

Three major approaches:

– **Axiomatic semantics:** $\{p\}$ `Prog` $\{q\}$

– **Denotational semantics:** `Prog` *denotes* a *mathematical* function $[\![$`Prog`$]\!]$

– **Operational semantics:** state transitions of an abstract machine

Simple Semantic Domains

From the textbook:

> A semantic domain is any set whose properties and operations are independently well-understood and upon which the functions that define the semantics of a language are ultimately based.

Primitive domains: $\mathrm{B} = \{\mathsf{True}, \mathsf{False}\}$, $\mathbb{N}$, $\mathbb{Z}$, Char, $\mathsf{seq}\,\mathit{Char}$, Ident

Domains for Program States:

- **Locations** are usually natural numbers: $\mathit{Loc} = \mathbb{N}$
- **Values** are, in a simple context, integers: $\mathit{Val}_0 = \mathbb{Z}$
- **Memory states** can be considered as partial functions: $\mathit{Mem}_0 = \mathbb{N} \nrightarrow \mathit{Val}_0$
- **Simple environments** are partial functions, too: $\mathit{Env}_0 = \mathit{Ident} \nrightarrow \mathit{Loc}$
- A simple **state** is pair: $\mathit{State}_0 = \mathit{Env}_0 \times \mathit{Mem}_0$
- A **simple store** directly maps identifiers to values: $\mathit{Store}_0 = \mathit{Ident} \nrightarrow \mathit{Val}_0$

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

- $\oplus$ is **not commutative**

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

- $\oplus$ is **not commutative**
- ***Textbook:*** "overriding union" operator "$\overline{\cup}$"

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

- $\oplus$ is **not commutative**

- ***Textbook:*** "overriding union" operator "$\overline{\cup}$"
- ***Haskell:***
  addListToFM :: *Ord key* ⇒ *FiniteMap k v* → [(*k* , *v*)] → *FiniteMap k v*

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

- $\oplus$ is **not commutative**

- ***Textbook:*** "overriding union" operator "$\overline{\cup}$"
- ***Haskell:***
  $\mathit{addListToFM} :: \mathit{Ord\ key} \Rightarrow \mathit{FiniteMap\ k\ v} \rightarrow [(k, v)] \rightarrow \mathit{FiniteMap\ k\ v}$
- If Q and R are both partial functions, then $Q \oplus R$ is a partial function, too.

Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does.

$$Q \oplus R = \{(x,y) : Q \mid x \notin \mathsf{dom}\ R\} \cup R$$

- $\oplus$ is **not commutative**

- ***Textbook:*** "overriding union" operator "$\overline{\cup}$"
- ***Haskell:***
  $\mathit{addListToFM} :: \mathit{Ord\ key} \Rightarrow \mathit{FiniteMap\ k\ v} \rightarrow [(k, v)] \rightarrow \mathit{FiniteMap\ k\ v}$
- If Q and R are both partial functions, then $Q \oplus R$ is a partial function, too.
- $\oplus$ is used to model
  - writing into memory or store locations
  - insertion into environments (*shadowing* previous bindings)

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\qquad \sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\qquad \sigma_1(s) \Rightarrow \sigma_2$

Execution axioms: $\sigma(c) \Rightarrow c \qquad \sigma(v) \Rightarrow \sigma\ v$ if $v \in \mathsf{dom}\ \sigma$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

Execution axioms: $\sigma(c) \Rightarrow c$ $\sigma(v) \Rightarrow \sigma\ v$ if $v \in \mathsf{dom}\ \sigma$

Execution rules:

$$\frac{premise}{conclusion} \quad \text{or} \quad \frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\qquad \sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\qquad \sigma_1(s) \Rightarrow \sigma_2$

Execution axioms: $\sigma(c) \Rightarrow c$ $\qquad \sigma(v) \Rightarrow \sigma\ v$ if $v \in \mathrm{dom}\ \sigma$

Execution rules:

$$\frac{premise}{conclusion} \qquad \text{or} \qquad \frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

Example rule — **addition:**

$$\frac{\sigma(e_1) \Rightarrow v_1 \qquad \sigma(e_2) \Rightarrow v_2}{\sigma(e_1 + e_2) \Rightarrow v_1 + v_2}$$

Operational Semantics

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\qquad \sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\qquad \sigma_1(s) \Rightarrow \sigma_2$

Execution axioms: $\quad \sigma(c) \Rightarrow c \qquad\qquad \sigma(v) \Rightarrow \sigma\, v$ if $v \in \mathsf{dom}\ \sigma$

Execution rules:

$$\frac{\textit{premise}}{\textit{conclusion}} \qquad \text{or} \qquad \frac{\textit{premise}_1 \quad \ldots \quad \textit{premise}_n}{\textit{conclusion}}$$

Example rule — **addition:**

$$\frac{\sigma(e_1) \Rightarrow v_1 \qquad\qquad \sigma(e_2) \Rightarrow v_2}{\sigma(e_1 + e_2) \Rightarrow v_1 + v_2}$$

(The left “ + ” is **syntax,** the right “ + ” is a mathematical operation on numbers.)

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:

— expression evaluation: $eval : State_1 \times Expr \nrightarrow Val_1$

— statement execution: $exec : State_1 \times Stmt \nrightarrow State_1$

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:

— expression evaluation: $eval : State_1 \times Expr \nrightarrow Val_1$

— statement execution: $exec : State_1 \times Stmt \nrightarrow State_1$

Note: one **syntactic** and one **semantic** argument.

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:

— expression evaluation: $eval : State_1 \times Expr \nrightarrow Val_1$

— statement execution: $exec : State_1 \times Stmt \nrightarrow State_1$

Note: one **syntactic** and one **semantic** argument.

Two **interpreter functions** (assuming **deterministic** semantics):

$$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe\ Value1$$
$$interpStmt :: Statement \rightarrow State1 \rightarrow Maybe\ State1$$

Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression e starting in store σ produces value v $\sigma(e) \Rightarrow v$

— Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:

— expression evaluation: $eval : State_1 \times Expr \nrightarrow Val_1$

— statement execution: $exec : State_1 \times Stmt \nrightarrow State_1$

Note: one **syntactic** and one **semantic** argument.

Two **interpreter functions** (assuming **deterministic** semantics):

```
evalExpr   :: Expression -> State1 -> Maybe Value1
interpStmt :: Statement  -> State1 -> Maybe State1
```

```
data Value1 = ValInt Int | ValBool Bool
type State1 = FiniteMap Variable Value1 -- even simpler than State_0
```

Interpreter: Expression Evaluation

$$\mathit{evalExpr} :: \mathit{Expression} \rightarrow \mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{Value1}$$

$$\begin{aligned}\textbf{data}\ \mathit{Value1} &= \mathit{ValInt}\ \mathit{Int}\\ &\mid \mathit{ValBool}\ \mathit{Bool}\end{aligned}$$

$\textbf{type}\ \mathit{State1} = \mathit{FiniteMap}\ \mathit{Variable}\ \mathit{Value1}$ -- even simpler than $State_0$

Interpreter: Expression Evaluation

$$evalExpr :: Expression \to State1 \to Maybe\ Value1$$

data $Value1 = ValInt\ Int$
$\quad\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$$evalExpr\ (Var\ v)\ s =$$

Interpreter: Expression Evaluation

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$

Interpreter: Expression Evaluation

$evalExpr :: Expression \to State1 \to Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$

Interpreter: Expression Evaluation

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ -- better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$

Interpreter: Expression Evaluation

$evalExpr :: Expression \to State1 \to Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ -- better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$

Interpreter: Expression Evaluation

$evalExpr :: Expression \to State1 \to Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ −− better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$
case $(evalExpr\ e1\ s, evalExpr\ e2\ s)$ **of**

Interpreter: Expression Evaluation

$evalExpr :: Expression \to State1 \to Maybe\ Value1$

$\textbf{data}\ Value1 = ValInt\ Int$
$\quad\quad\quad\quad | \ ValBool\ Bool$

$\textbf{type}\ State1 = FiniteMap\ Variable\ Value1$ –– even simpler than $State_0$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ –– better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$
$\quad \textbf{case}\ (evalExpr\ e1\ s,\ evalExpr\ e2\ s)\ \textbf{of}$
$\quad\quad (Just\ (ValInt\ v1),\ Just\ (ValInt\ v2)) \to$

Interpreter: Expression Evaluation

$evalExpr :: Expression \to State1 \to Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ -- better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$
case $(evalExpr\ e1\ s, evalExpr\ e2\ s)$ **of**
$(Just\ (ValInt\ v1), Just\ (ValInt\ v2)) \to Just\ (ValInt\ (v1 + v2))$

Interpreter: Expression Evaluation

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | \ ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ –– better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$
case $(evalExpr\ e1\ s, evalExpr\ e2\ s)$ **of**
$(Just\ (ValInt\ v1), Just\ (ValInt\ v2)) \rightarrow Just\ (ValInt\ (v1 + v2))$
$\_ \rightarrow Nothing$

Interpreter: Expression Evaluation (*Maybe* Monad)

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe\ Value1$

data $Value1 = ValInt\ Int$
$\quad | ValBool\ Bool$

type $State1 = FiniteMap\ Variable\ Value1$

$evalExpr\ (Var\ v)\ s = lookupFM\ s\ v$
$evalExpr\ (Value\ (LitInt\ i))\ s = Just\ (ValInt\ i)$ -- better: function litToVal
$evalExpr\ (Value\ (LitBool\ b))\ s = Just\ (ValBool\ b)$
$evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2)\ s =$ **do**
$\quad ValInt\ v1 \leftarrow evalExpr\ e1\ s$
$\quad ValInt\ v2 \leftarrow evalExpr\ e2\ s$
$\quad Just\ (ValInt\ (v1 + v2))$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\sigma_1(x := x + 3) \Rightarrow$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\sigma_1(x + 3) \Rightarrow \qquad\qquad\qquad}{\sigma_1(x := x + 3) \Rightarrow \qquad\qquad\qquad\qquad\qquad\qquad}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow \qquad\qquad \sigma_1(3) \Rightarrow}{\sigma_1(x + 3) \Rightarrow}}{\sigma_1(x := x + 3) \Rightarrow}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow 39 \qquad\qquad \sigma_1(3) \Rightarrow}{\sigma_1(x + 3) \Rightarrow}}{\sigma_1(x := x + 3) \Rightarrow}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\cfrac{\cfrac{\sigma_1(x) \Rightarrow 39 \qquad\qquad \sigma_1(3) \Rightarrow 3}{\sigma_1(x + 3) \Rightarrow}}{\sigma_1(x := x + 3) \Rightarrow}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow 39 \qquad \sigma_1(3) \Rightarrow 3}{\sigma_1(x + 3) \Rightarrow 42}}{\sigma_1(x := x + 3) \Rightarrow}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow 39 \qquad \sigma_1(3) \Rightarrow 3}{\sigma_1(x + 3) \Rightarrow 42}}{\sigma_1(x := x + 3) \Rightarrow \{x \mapsto 42, y \mapsto 7\}}$$

Assignment

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

For example:

- Assume $\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$
- Then:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow 39 \qquad \sigma_1(3) \Rightarrow 3}{\sigma_1(x + 3) \Rightarrow 42}}{\sigma_1(x := x + 3) \Rightarrow \{x \mapsto 42, y \mapsto 7\}}$$

since $\sigma_1 \oplus \{x \mapsto 42\} = \{x \mapsto 42, y \mapsto 7\}$

Interpreter: Assignment

$$\mathit{evalExpr} :: \mathit{Expression} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{Value1}$$
$$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$$

data *Value1* = *ValInt Int*
　　| *ValBool Bool*
type *State1* = *FiniteMap Variable Value1*

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

interpStmt (*Assignment var e*) *s* =

Interpreter: Assignment

$$\mathit{evalExpr} :: \mathit{Expression} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{Value1}$$
$$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$$

$$\textbf{data}\ \mathit{Value1} = \mathit{ValInt}\ \mathit{Int}$$
$$\quad\quad \mid \mathit{ValBool}\ \mathit{Bool}$$
$$\textbf{type}\ \mathit{State1} = \mathit{FiniteMap}\ \mathit{Variable}\ \mathit{Value1}$$

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

$$\mathit{interpStmt}\ (\mathit{Assignment}\ \mathit{var}\ e)\ s = \textbf{case}\ \mathit{evalExpr}\ e\ s\ \textbf{of}$$
$$\quad \mathit{Just}\ \mathit{val} \to \mathit{Just}\ (\mathit{addToFM}\ s\ \mathit{var}\ \mathit{val})$$
$$\quad \mathit{Nothing} \to \mathit{Nothing}$$

Interpreter: Assignment

$$\mathit{evalExpr} :: \mathit{Expression} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{Value1}$$
$$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$$

data *Value1* = *ValInt Int*
  | *ValBool Bool*
type *State1* = *FiniteMap Variable Value1*

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

interpStmt (*Assignment var e*) *s* = **case** *evalExpr e s* **of**
  Just val → *Just* (*addToFM s var val*)
  Nothing → *Nothing*

(Using the *Maybe* monad:)

interpStmt (*Assignment var e*) *s* = **do**
  val ← *evalExpr e s* **of**
  Just (*addToFM s var val*)

Sequencing, Conditionals, Loops

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1;\ s_2) \Rightarrow \sigma_3}$$

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False} \qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\{x \mapsto 7\}(P) \Rightarrow$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while}\ x < 50\ \textbf{do}\ x := 2 * x\ \textbf{od}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \{x \mapsto 14\}(P) \Rightarrow}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\,(x := 2*x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \text{True}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\cfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \cfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \{x \mapsto 28\}(P) \Rightarrow}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \;\equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\,(x < 50) \Rightarrow}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \;\equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\ (x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \mathsf{True}}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad\qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\cfrac{\{x \mapsto 7\}\,(x<50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\,(x := 2*x) \Rightarrow \{x \mapsto 14\} \qquad \cfrac{\{x \mapsto 14\}\,(x<50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\,(x := 2*x) \Rightarrow \{x \mapsto 28\} \qquad \cfrac{\{x \mapsto 28\}\,(x<50) \Rightarrow \text{True} \qquad \{x \mapsto 28\}\,(x := 2*x) \Rightarrow}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 28\}\,(x := 2 * x) \Rightarrow \{x \mapsto 56\}}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\cfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \cfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \cfrac{\{x \mapsto 28\}\,(x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 28\}\,(x := 2 * x) \Rightarrow \{x \mapsto 56\} \qquad \{x \mapsto 56\}(P) \Rightarrow}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\ (x := 2*x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\ (x := 2*x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 28\}\ (x := 2*x) \Rightarrow \{x \mapsto 56\} \qquad \dfrac{\{x \mapsto 56\}\ (x < 50) \Rightarrow \text{False}}{\{x \mapsto 56\}(P) \Rightarrow}}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 14\}\ (x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \text{True} \qquad \{x \mapsto 28\}\ (x := 2 * x) \Rightarrow \{x \mapsto 56\} \qquad \dfrac{\{x \mapsto 56\}\ (x < 50) \Rightarrow \text{False}}{\{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 28\}(P) \Rightarrow}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\ (x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 28\}\ (x := 2 * x) \Rightarrow \{x \mapsto 56\} \qquad \dfrac{\{x \mapsto 56\}\ (x < 50) \Rightarrow \mathsf{False}}{\{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 14\}(P) \Rightarrow}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\dfrac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \quad \dfrac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 14\}\ (x := 2 * x) \Rightarrow \{x \mapsto 28\} \quad \dfrac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 28\}\ (x := 2 * x) \Rightarrow \{x \mapsto 56\} \quad \dfrac{\{x \mapsto 56\}\ (x < 50) \Rightarrow \text{False}}{\{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 14\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 7\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Loop Example

$$P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 * x \textbf{ od}$$

$$\frac{\{x \mapsto 56\}\ (x < 50) \Rightarrow \mathsf{False}}{\{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}$$

$$\frac{\{x \mapsto 28\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 28\}\ (x := 2 * x) \Rightarrow \{x \mapsto 56\} \qquad \{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}{\{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}}$$

$$\frac{\{x \mapsto 14\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\ (x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}}{\{x \mapsto 14\}(P) \Rightarrow \{x \mapsto 56\}}$$

$$\frac{\{x \mapsto 7\}\ (x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\ (x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \{x \mapsto 14\}(P) \Rightarrow \{x \mapsto 56\}}{\{x \mapsto 7\}(P) \Rightarrow \{x \mapsto 56\}}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Sequencing, Conditionals, Loops

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1;\ s_2) \Rightarrow \sigma_3}$$

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False} \qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Additional Control Structures

- do { ... } while (...)
- *repeat* { ... } *until* (...)
- for (... , ... , ...) { ... }
- for $i = beg$ *to end* do { ... }

Additional Control Structures

- do { ... } while (...)
- *repeat* { ... } *until* (...)
- for (... , ... , ...) { ... }
- for *i* = *beg to end* do { ... }

Options:

– **Direct definition** using new operational semantics rules

Additional Control Structures

- do { ... } while (...)
- *repeat* { ... } *until* (...)
- for (... , ... , ...) { ... }
- for *i* = *beg to end* do { ... }

Options:

– **Direct definition** using new operational semantics rules

$$\frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \text{False}}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_2} \qquad \frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \text{True} \qquad \sigma_2(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}$$

Additional Control Structures

- do { ... } while (...)
- *repeat* { ... } *until* (...)
- for (... , ... , ...) { ... }
- for *i* = *beg to end* do { ... }

Options:

– **Direct definition** using new operational semantics rules

$$\frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \text{False}}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_2} \qquad \frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \text{True} \qquad \sigma_2(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}$$

– **Translation into core language** — ***derived*** features

Additional Control Structures

- do { ... } while (...)
- *repeat* { ... } *until* (...)
- for (... , ... , ...) { ... }
- for *i* = *beg to end* do { ... }

Options:

– **Direct definition** using new operational semantics rules

$$\frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \textsf{False}}{\sigma_1(\ \textbf{do}\ s\ \textbf{while}\ (b)) \Rightarrow \sigma_2} \qquad \frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \textsf{True} \qquad \sigma_2(\ \textbf{do}\ s\ \textbf{while}\ (b)) \Rightarrow \sigma_3}{\sigma_1(\ \textbf{do}\ s\ \textbf{while}\ (b)) \Rightarrow \sigma_3}$$

– **Translation into core language** — ***derived*** features

$$\frac{\sigma_1(s\ \textbf{;}\ \textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma_1(\ \textbf{do}\ s\ \textbf{while}\ (c)) \Rightarrow \sigma_2}$$

Additional Language Features

- Output: **print** (e)
- Input: **read** (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Additional Language Features

– Output: **print** (e)

– Input: **read** (e)

– Nested Scopes (declarations in inner blocks)

– Function and procedure calls

– Side-effecting expressions

Main tasks:

Additional Language Features

- Output: **print** (e)
- Input: **read** (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space

Additional Language Features

– Output: **print** (e)

– Input: **read** (e)

– Nested Scopes (declarations in inner blocks)

– Function and procedure calls

– Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

Additional Language Features

- Output: **print** (e)
- Input: **read** (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

  e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', v)$

Additional Language Features

– Output: **print** (e)

– Input: **read** (e)

– Nested Scopes (declarations in inner blocks)

– Function and procedure calls

– Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

  e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', v)$
- "Port" all existing feature definitions to the new states

Additional Language Features

- Output: **print** (e)
- Input: **read** (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

  e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', v)$
- "Port" all existing feature definitions to the new states
- Appropriately define the new features

Additional Language Features

- Output: **print** (e)
- Input: **read** (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

  e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', v)$
- "Port" all existing feature definitions to the new states
- Appropriately define the new features
- Prove "conservative extension"

Additional Language Features

– Output: **print** (e)

– Input: **read** (e)

– Nested Scopes (declarations in inner blocks)

– Function and procedure calls

– Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary

  e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', v)$
- "Port" all existing feature definitions to the new states
- Appropriately define the new features
- Prove "conservative extension": mapping from old states to new is injective and preserves transitions.

New Language Feature Example: Output

Assume a new statement “ **print** (e)”

New Language Feature Example: Output

Assume a new statement “ **print** (*e*)” that prints the **integer** expression *e* to the screen.

New Language Feature Example: Output

Assume a new statement " **print** (*e*)" that prints the **integer** expression *e* to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.

New Language Feature Example: Output

Assume a new statement “ **print** (e)” that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$

New Language Feature Example: Output

Assume a new statement " **print** (*e*)" that prints the **integer** expression *e* to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$
- New state space: $\mathit{State}_2 = \mathit{State}_1 \times [\mathbb{Z}]$

New Language Feature Example: Output

Assume a new statement "**print** (e)" that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: *Print* :: *Expression* $\rightarrow$ *Statement*
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$

New Language Feature Example: Output

Assume a new statement " **print** (*e*)" that prints the **integer** expression *e* to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$
- New state space: $\mathit{State}_2 = \mathit{State}_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules

New Language Feature Example: Output

Assume a new statement " **print** (e)" that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules, e.g.:

$$\frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)}$$

New Language Feature Example: Output

Assume a new statement “ **print** (e)” that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules, e.g.: $$\frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)}$$
- **Rules for new feature:**

New Language Feature Example: Output

Assume a new statement " **print** (e)" that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: *Print* :: *Expression* → *Statement*
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules, e.g.: $$\frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)}$$
- **Rules for new feature:** $$\frac{\sigma(e) \Rightarrow i}{(\sigma, out)(\textbf{print}\ (e)) \Rightarrow (\sigma, out +\!\!+ [i])}$$

New Language Feature Example: Output

Assume a new statement "**print** (e)" that prints the **integer** expression e to the screen.

- New typing rule: the argument of *PRINT* has to be of type integer.
- New abstract syntax constructor: $\mathit{Print} :: \mathit{Expression} \rightarrow \mathit{Statement}$
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules, e.g.: $$\frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)}$$
- **Rules for new feature:** $$\frac{\sigma(e) \Rightarrow i}{(\sigma, out)(\textbf{print}\ (e)) \Rightarrow (\sigma, out +\!\!+ [i])}$$
- Check determinism, add to interpreter.

Exceptions

Exceptions

- An **exception** is an event that needs to be handled in a special way

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is **thrown** to a higher part of the code

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is **thrown** to a higher part of the code
- A thrown exception is **caught** by an appropriate **exception handler** that processes the exception

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is **thrown** to a higher part of the code
- A thrown exception is **caught** by an appropriate **exception handler** that processes the exception
- Benefits of an **exception handling** mechanism:
  - Exception-handling code can be **separated** from regular code

Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is **thrown** to a higher part of the code
- A thrown exception is **caught** by an appropriate **exception handler** that processes the exception
- Benefits of an **exception handling** mechanism:
  - Exception-handling code can be **separated** from regular code
  - Exceptions can be handled **at the most appropriate place** in the code, not necessarily where they are generated

Exceptions in Java

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
- `Error`: **Nonrecoverable** errors thrown by the system

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
- `Error`: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
- `Error`: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions** which must be declared with a `throws` clause in a method declaration

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
- `Error`: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions** which must be declared with a `throws` clause in a method declaration:

  All user-defined exceptions, *IOExc.*, *ClassNotFoundExc.*, …

Exceptions in Java

In Java, exceptions are represented by objects in the *java.lang.Throwable* class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
- `Error`: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions** which must be declared with a `throws` clause in a method declaration:

  All user-defined exceptions, *IOExc*., *ClassNotFoundExc*., …
- `RuntimeException`: Abnormal runtime events which need not be declared with a `throws` clause:

  ArithmeticExc., *ClassCastExc*., *IllegalArgumentExc*., *IndexOutOfBoundsExc*., *NullPointerExc*., …

Exception Handling in Java

Exception Handling in Java

- Any Java code can construct an exception and then throw it

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement
- Every **checked exception** that can be thrown in a method must be either caught in the method or declared in the method with a throws clause

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement
- Every **checked exception** that can be thrown in a method must be either caught in the method or declared in the method with a throws clause
- Benefits of Java's exception handling mechanism:

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement
- Every **checked exception** that can be thrown in a method must be either caught in the method or declared in the method with a throws clause
- Benefits of Java's exception handling mechanism:
  - A class of exceptions can be subclassed

Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement
- Every **checked exception** that can be thrown in a method must be either caught in the method or declared in the method with a throws clause
- Benefits of Java's exception handling mechanism:
  - A class of exceptions can be subclassed
  - There is an **enforced discipline** for checked exceptions

Try-Catch-Finally Statement

try {
  try body
}
catch (*Exception*$_1$ *var*$_1$) {
  catch$_1$ *body*
}
catch (*Exception*$_2$ *var*$_2$) {
  catch$_2$ *body*
}
…
catch (*Exception*$_n$ *var*$_n$) {
  catch$_n$ *body*
}
finally {
  finally body
}

Try-Catch-Finally Example

```
import java.io.*;
class Read1 {
  public static void main(String[] args) {
    BufferedReader in =
     new BufferedReader(new InputStreamReader(System.in));
    try { System.out.println("How old are you?");
            String inputLine = in.readLine();
            int age = Integer.parseInt(inputLine);
            age++;
            System.out.println("Next year, you'll be " + age);
    }
    catch (IOException exception)
    { System.out.println("Input/output error " + exception); }
    catch (NumberFormatException exception)
    { System.out.println("Input was not a number"); }
    finally { if (in ≠ null) { try { in.close(); }
                                    catch (IOException exception) {}  } }
}}
```

Ways of Handling Exceptions

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**
- Capture it and execute no code (ignore the exception)

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**
- Capture it and execute no code (ignore the exception) — ***bad idea!***

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**
- Capture it and execute no code (ignore the exception) — ***bad idea!***
- Do **not capture** it (let it propagate up)

Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**
- Capture it and execute no code (ignore the exception) — ***bad idea!***
- Do **not capture** it (let it propagate up) — *may need to declare!*

Exceptions — Example

```
class Simulate4 {
  private static void println(String s)
   {System.out.println(s);}
  public static int _q = 0;
  public static void main(String[] a)
  { int s = g (2);
    println("* "+s+"  "+_q);
  }
  public static int f(int k, int m) {
    println("f("+k+","+m+")");
    _q += m;
    int r = g(k) + _q;
    println("f("+k+","+m+")="+r);
    return r;
  }
```

```
public static int g(int n) {
    println("g(" + n + ")");
    int t = 3 * n;
    if ( t < 10 ) {
      try { t = (f (n + 1, _q));
          }
      catch (Exception e) {
        println("g: caught exception!");
        _q += n;
      }
    }
    t = t / _q;
    println("g( " + n + " )= " + t);
    return t;
  }
}
```

Operational Semantics of Exceptions

Operational Semantics of Exceptions

Originally: Two kinds of assertions:

$\sigma(e) \Rightarrow v$ — evaluating expression e starting in state σ can produce value v

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Operational Semantics of Exceptions

Originally: Two kinds of assertions:

$\sigma(e) \Rightarrow v$ — evaluating expression e starting in state σ can produce value v

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Now an additional possibility:

$\sigma_1(s) \stackrel{!}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **rasing exception** x

Operational Semantics of Exceptions

Originally: Two kinds of assertions:

$\sigma(e) \Rightarrow v$ — evaluating expression e starting in state σ can produce value v

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Now an additional possibility:

$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **rasing exception** x

Two additional sequencing rules:

$$\frac{\sigma_1(s_1) \overset{!}{\Rightarrow} (\sigma_2, z)}{\sigma_1(s_1;\ s_2) \overset{!}{\Rightarrow} (\sigma_2, z)}$$

Operational Semantics of Exceptions

Originally: Two kinds of assertions:

$\sigma(e) \Rightarrow v$ — evaluating expression e starting in state σ can produce value v

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Now an additional possibility:

$\sigma_1(s) \stackrel{!}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **rasing exception** x

Two additional sequencing rules:

$$\frac{\sigma_1(s_1) \stackrel{!}{\Rightarrow} (\sigma_2, z)}{\sigma_1(s_1;\ s_2) \stackrel{!}{\Rightarrow} (\sigma_2, z)} \qquad \frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \stackrel{!}{\Rightarrow} (\sigma_3, z)}{\sigma_1(s_1;\ s_2) \stackrel{!}{\Rightarrow} (\sigma_3, z)}$$

Operational Semantics of Exceptions

Originally: Two kinds of assertions:

$\sigma(e) \Rightarrow v$ — evaluating expression e starting in state σ can produce value v

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Now an additional possibility:

$\sigma_1(s) \stackrel{!}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **rasing exception** x

Two additional sequencing rules:

$$\frac{\sigma_1(s_1) \stackrel{!}{\Rightarrow} (\sigma_2, z)}{\sigma_1(s_1;\ s_2) \stackrel{!}{\Rightarrow} (\sigma_2, z)} \qquad \frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \stackrel{!}{\Rightarrow} (\sigma_3, z)}{\sigma_1(s_1;\ s_2) \stackrel{!}{\Rightarrow} (\sigma_3, z)}$$

Two additional if **rules (no exceptions in expression evaluation yet):**

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s_1) \stackrel{!}{\Rightarrow} (\sigma_{1,} x)}{\sigma(\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}) \stackrel{!}{\Rightarrow} (\sigma_1, x)} \qquad \frac{\sigma(b) \Rightarrow \textsf{False} \qquad \sigma(s_2) \stackrel{!}{\Rightarrow} (\sigma_{2,} x)}{\sigma(\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}) \stackrel{!}{\Rightarrow} (\sigma_2, x)}$$

Exceptions — Interpreter

Original statement interpretation:

$$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$$

Exceptions — Interpreter

Original statement interpretation:

$$\mathit{interpStmt} :: \mathit{Statement} \rightarrow \mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1}$$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \qquad \textbf{iff} \qquad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ \sigma_2$$

$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \qquad \textbf{iff} \qquad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot,\ \mathit{Nothing}\}$$

Exceptions — Interpreter

Original statement interpretation:

$\mathit{interpStmt} :: \mathit{Statement} \rightarrow \mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1}$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ \sigma_2$$
$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot, \mathit{Nothing}\}$$

Statement interpretation **with exceptions**:

$\mathit{interpStmtExc} :: \mathit{Statement} \rightarrow \mathit{State1} \rightarrow \mathit{Maybe}\ (\mathit{Either}\ \mathit{State1}\ (\mathit{State1}, \mathit{Exc}))$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff}$$
$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff}$$

Exceptions — Interpreter

Original statement interpretation:

$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ \sigma_2$$
$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot, \mathit{Nothing}\}$$

Statement interpretation **with exceptions**:

$\mathit{interpStmtExc} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ (\mathit{Either}\ \mathit{State1}\ (\mathit{State1}, \mathit{Exc}))$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Left}\ \sigma_2)$$
$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff}$$

Exceptions — Interpreter

Original statement interpretation:

$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ \sigma_2$$
$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot, \mathit{Nothing}\}$$

Statement interpretation **with exceptions**:

$\mathit{interpStmtExc} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ (\mathit{Either}\ \mathit{State1}\ (\mathit{State1}, \mathit{Exc}))$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Left}\ \sigma_2)$$
$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Right}\ (\sigma_2, x))$$

Exceptions — Interpreter

Original statement interpretation:

$interpStmt :: Statement \to State1 \to Maybe\ State1$

meaning:

$\sigma_1(s) \Rightarrow \sigma_2$	**iff**	$interpStmt\ s\ \sigma_1 = Just\ \sigma_2$
$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2$	**iff**	$interpStmt\ s\ \sigma_1 \in \{\bot, Nothing\}$

Statement interpretation **with exceptions**:

$interpStmtExc :: Statement \to State1 \to Maybe\ (Either\ State1\ (State1, Exc))$

meaning:

$\sigma_1(s) \Rightarrow \sigma_2$	**iff**	$interpStmt\ s\ \sigma_1 = Just\ (Left\ \sigma_2)$
$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x)$	**iff**	$interpStmt\ s\ \sigma_1 = Just\ (Right\ (\sigma_2, x))$
	iff	$interpStmt\ s\ \sigma_1 \in \{\bot, Nothing\}$

Exceptions — Interpreter

Original statement interpretation:

$$\mathit{interpStmt} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ \mathit{State1}$$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ \sigma_2$$

$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot,\ \mathit{Nothing}\}$$

Statement interpretation **with exceptions**:

$$\mathit{interpStmtExc} :: \mathit{Statement} \to \mathit{State1} \to \mathit{Maybe}\ (\mathit{Either}\ \mathit{State1}\ (\mathit{State1}, \mathit{Exc}))$$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Left}\ \sigma_2)$$

$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 = \mathit{Just}\ (\mathit{Right}\ (\sigma_2, x))$$

$$\neg\exists\sigma_2, x \bullet\ \sigma_1(s) \Rightarrow \sigma_2 \ \vee \ \sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff} \quad \mathit{interpStmt}\ s\ \sigma_1 \in \{\bot,\ \mathit{Nothing}\}$$

Exceptions in Expression Evaluation

Exercise

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\frac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix} (x \neq 0) \Rightarrow}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\frac{\begin{matrix}\{s \mapsto 0, \\ \ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \text{True}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\frac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\frac{\begin{matrix}\{s \mapsto 0,\\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{matrix}\{s \mapsto 0,\\ x \mapsto -4\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4,\\ x \mapsto -5\}\end{matrix}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\frac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\{s \mapsto -4, x \mapsto -5\}(x \neq 0) \Rightarrow}{\{s \mapsto -4, x \mapsto -5\}(P) \Rightarrow} \qquad \{s \mapsto 0, x \mapsto -4\}(x \neq 0) \Rightarrow \mathsf{True} \qquad \{s \mapsto 0, x \mapsto -4\}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} (x \neq 0) \Rightarrow \text{True}}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow} \\ \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix} (x \neq 0) \Rightarrow \text{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix} (s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\dfrac{\dfrac{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow}{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ ;\ x := x - 1\ \textbf{od}$$

$$\dfrac{\dfrac{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x\ ;\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}}{}}{}$$

$$\dfrac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x\ ;\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\dfrac{\begin{array}{c}\{s \mapsto 0, \\ x \mapsto -4\}\end{array}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{array}{c}\{s \mapsto 0, \\ x \mapsto -4\}\end{array}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{array}{c}\{s \mapsto -4, \\ x \mapsto -5\}\end{array} \qquad \dfrac{\begin{array}{c}\{s \mapsto -4, \\ x \mapsto -5\}\end{array}(x \neq 0) \Rightarrow \mathsf{True} \qquad \begin{array}{c}\{s \mapsto -4, \\ x \mapsto -5\}\end{array}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{array}{c}\{s \mapsto -9, \\ x \mapsto -6\}\end{array} \qquad \begin{array}{c}\{s \mapsto -9, \\ x \mapsto -6\}\end{array}(P) \Rightarrow}{\begin{array}{c}\{s \mapsto -4, \\ x \mapsto -5\}\end{array}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(x \neq 0) \Rightarrow}{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix} \qquad \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(P) \Rightarrow}}{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \qquad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \qquad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}$$

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True}}{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(P) \Rightarrow}}{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \quad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix} \quad \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(P) \Rightarrow}}{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \mathsf{True} \quad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x\ \textbf{;}\ x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \quad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}$$

$$\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \quad \dfrac{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix} \quad \dfrac{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow}{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(P) \Rightarrow}}{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\dfrac{\{s \mapsto -9, x \mapsto -6\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -9, x \mapsto -6\}(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -15, x \mapsto -7\}}{\{s \mapsto -4, x \mapsto -5\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -4, x \mapsto -5\}(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\} \quad \{s \mapsto -9, x \mapsto -6\}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto 0, x \mapsto -4\}(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\} \quad \{s \mapsto -4, x \mapsto -5\}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\cfrac{\begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto 0, \\ x \mapsto -4\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix} \quad \cfrac{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix} \quad \cfrac{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(x \neq 0) \Rightarrow \text{True} \quad \begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \begin{matrix}\{s \mapsto -15, \\ x \mapsto -7\}\end{matrix} \quad \begin{matrix}\{s \mapsto -15, \\ x \mapsto -7\}\end{matrix}(P) \Rightarrow}{\begin{matrix}\{s \mapsto -9, \\ x \mapsto -6\}\end{matrix}(P) \Rightarrow}}{\begin{matrix}\{s \mapsto -4, \\ x \mapsto -5\}\end{matrix}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\{s \mapsto 0, x \mapsto -4\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto 0, x \mapsto -4\}\,(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\} \qquad \dfrac{\{s \mapsto -4, x \mapsto -5\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto -4, x \mapsto -5\}\,(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\} \qquad \dfrac{\{s \mapsto -9, x \mapsto -6\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto -9, x \mapsto -6\}\,(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -15, x \mapsto -7\} \qquad \dfrac{\{s \mapsto -15, x \mapsto -7\}\,(x \neq 0) \Rightarrow}{\{s \mapsto -15, x \mapsto -7\}\,(P) \Rightarrow}}{\{s \mapsto -9, x \mapsto -6\}\,(P) \Rightarrow}}{\{s \mapsto -4, x \mapsto -5\}\,(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\{s \mapsto 0, x \mapsto -4\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto 0, x \mapsto -4\}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\} \quad \dfrac{\{s \mapsto -4, x \mapsto -5\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -4, x \mapsto -5\}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\} \quad \dfrac{\{s \mapsto -9, x \mapsto -6\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -9, x \mapsto -6\}(s := s + x \textbf{ ; } x := x - 1) \Rightarrow \{s \mapsto -15, x \mapsto -7\} \quad \dfrac{\{s \mapsto -15, x \mapsto -7\}(x \neq 0) \Rightarrow \text{True} \quad \ldots}{\{s \mapsto -15, x \mapsto -7\}(P) \Rightarrow}}{\{s \mapsto -9, x \mapsto -6\}(P) \Rightarrow}}{\{s \mapsto -4, x \mapsto -5\}(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$$\dfrac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \dfrac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Another Loop Example

$$P \quad \equiv \quad \textbf{while } x \neq 0 \textbf{ do } s := s + x \,\textbf{;}\, x := x - 1 \textbf{ od}$$

$$\dfrac{\dfrac{\dfrac{\{s \mapsto 0, x \mapsto -4\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto 0, x \mapsto -4\}\,(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\} \qquad \dfrac{\{s \mapsto -4, x \mapsto -5\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto -4, x \mapsto -5\}\,(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\} \qquad \dfrac{\{s \mapsto -9, x \mapsto -6\}\,(x \neq 0) \Rightarrow \text{True} \qquad \{s \mapsto -9, x \mapsto -6\}\,(s := s + x \,;\, x := x - 1) \Rightarrow \{s \mapsto -15, x \mapsto -7\} \qquad \dfrac{\{s \mapsto -15, x \mapsto -7\}\,(x \neq 0) \Rightarrow \text{True} \qquad \ldots}{\{s \mapsto -15, x \mapsto -7\}\,(P) \Rightarrow}}{\{s \mapsto -9, x \mapsto -6\}\,(P) \Rightarrow}}{\{s \mapsto -4, x \mapsto -5\}\,(P) \Rightarrow}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}}{}}{}$$

This is not a direct proof of non-termination!

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$"

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression *e* starting in state σ can produce value *v*"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : Expr \to \mathbb{P}\,(State_1 \times Value)$$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : Expr \to \mathbb{P}\,(State_1 \times Value)$$

$$M_{\mathsf{Expr}} : Expr \to (State_1 \leftrightarrow Value)$$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : Expr \to \mathbb{P}\,(State_1 \times Value)$$
$$M_{\mathsf{Expr}} : Expr \to (State_1 \leftrightarrow Value)$$

If expression evaluation is deterministic, the result relations are all partial functions

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(\mathit{State}_1 \times \mathit{Expr} \times \mathit{Value})$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : \mathit{Expr} \rightarrow \mathbb{P}\,(\mathit{State}_1 \times \mathit{Value})$$

$$M_{\mathsf{Expr}} : \mathit{Expr} \rightarrow (\mathit{State}_1 \leftrightarrow \mathit{Value})$$

If expression evaluation is deterministic, the result relations are all partial functions:

$$M_{\mathsf{Expr}} : \mathit{Expr} \rightarrow (\mathit{State}_1 \nrightarrow \mathit{Value})$$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression e starting in state σ can produce value v"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : Expr \to \mathbb{P}\,(State_1 \times Value)$$

$$M_{\mathsf{Expr}} : Expr \to (State_1 \leftrightarrow Value)$$

If expression evaluation is deterministic, the result relations are all partial functions:

$$M_{\mathsf{Expr}} : Expr \to (State_1 \nrightarrow Value)$$

This corresponds to the Haskell type we have chosen:

$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$

From Operational to Relational Denotational Semantics

The derivable assertions of shape "$\sigma(e) \Rightarrow v$", meaning:

"Evaluating expression *e* starting in state σ can produce value *v*"

give rise to a ternary relation $\mathsf{evalExpr}_0 : \mathbb{P}\,(State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Expr}} : Expr \to \mathbb{P}\,(State_1 \times Value)$$

$$M_{\mathsf{Expr}} : Expr \to (State_1 \leftrightarrow Value)$$

If expression evaluation is deterministic, the result relations are all partial functions:

$$M_{\mathsf{Expr}} : Expr \to (State_1 \nrightarrow Value)$$

This corresponds to the Haskell type we have chosen:

$$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$$

Note: For an expression *e*, we write "$[\![e]\!]_E$" instead of "$M_{\mathsf{Expr}}(e)$".

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \rightarrow (State_1 \leftrightarrow State_1)$$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \nrightarrow State_1)$$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \rightarrow (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \rightarrow (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

This can be used for proving **undefinedness**

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \rightarrow (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \rightarrow (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

This can be used for proving **undefinedness**:

$$[\![\,\textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}\,]\!]_{\mathsf{S}}$$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

This can be used for proving **undefinedness**:

$$\mathsf{dom}\ [\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ s := s + x\ ;\ x := x - 1\ \mathbf{od}]\!]_{\mathsf{S}}$$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

This can be used for proving **undefinedness**:

$$\{s \mapsto 0, x \mapsto -4\} \notin \mathsf{dom}\ [\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ s := s + x\ \mathbf{;}\ x := x - 1\ \mathbf{od}]\!]_{\mathsf{S}}$$

Relational Denotational Statement Semantics

The derivable assertions of shape "$\sigma(s) \Rightarrow \sigma'$", meaning:

"Executing statement s starting in state σ can terminate in state σ'"

give rise to a ternary relation $\mathsf{execStmt}_0 : \mathbb{P}\,(State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\mathsf{Stmt}} : Stmt \to (State_1 \nrightarrow State_1)$$

Note: For a statement s, we write "$[\![s]\!]_{\mathsf{S}}$" instead of "$M_{\mathsf{Stmt}}(s)$".

This can be used for proving **undefinedness**:

$$\{s \mapsto 0, x \mapsto -4\} \notin \mathsf{dom}\,[\![\textbf{while}\ x \neq 0\ \textbf{do}\ s := s + x\ \textbf{;}\ x := x - 1\ \textbf{od}]\!]_{\mathsf{S}}$$

This uses properties of <u>mathematical object</u> found as denotational semantics of a statement.

Denotational Semantics is Compositional

$$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$$

Denotational Semantics is Compositional

$$\mathit{evalExpr} :: \mathit{Expression} \rightarrow (\mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{Value1})$$

We can reflect these parentheses in the definition:

$$\mathit{evalExpr}\ (\mathit{Var}\ v) = \lambda\ s \rightarrow \mathit{lookupFM}\ s\ v$$
$$\mathit{evalExpr}\ (\mathit{Value}\ \mathit{lit}) = \mathit{const}\ (\mathit{Just}\ (\mathit{litToVal}\ \mathit{lit}))$$

Denotational Semantics is Compositional

$$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$$

We can reflect these parentheses in the definition:

$$
\begin{array}{l}
\mathit{evalExpr}\ (\mathit{Var}\ v) \quad = \lambda\ s \to \mathit{lookupFM}\ s\ v \\
\mathit{evalExpr}\ (\mathit{Value}\ \mathit{lit}) = \mathit{const}\ (\mathit{Just}\ (\mathit{litToVal}\ \mathit{lit})) \\
\mathit{evalExpr}\ (\mathit{Binary}\ (\mathit{MkArithOp}\ \mathit{Plus})\ e1\ e2) = \lambda\ s \to \\
\quad \mathbf{case}\ ((\mathit{evalExpr}\ e1)\ s, (\mathit{evalExpr}\ e2)\ s)\ \mathbf{of} \\
\qquad (\mathit{Just}\ (\mathit{ValInt}\ v1), \mathit{Just}\ (\mathit{ValInt}\ v2)) \to \mathit{Just}\ (\mathit{ValInt}\ (v1 + v2)) \\
\qquad \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Nothing}
\end{array}
$$

Denotational Semantics is Compositional

$$evalExpr :: Expression \to (State1 \to Maybe\ Value1)$$

We can reflect these parentheses in the definition:

$$
\begin{array}{l}
evalExpr\ (Var\ v) \quad = \lambda\ s \to lookupFM\ s\ v \\
evalExpr\ (Value\ lit) = const\ (Just\ (litToVal\ lit)) \\
evalExpr\ (Binary\ (MkArithOp\ Plus)\ e1\ e2) = \lambda\ s \to \\
\quad \textbf{case}\ ((evalExpr\ e1)\ s, (evalExpr\ e2)\ s)\ \textbf{of} \\
\qquad (Just\ (ValInt\ v1), Just\ (ValInt\ v2)) \to Just\ (ValInt\ (v1 + v2)) \\
\qquad \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to Nothing
\end{array}
$$

Translating the last case back into mathematical notation:

Denotational Semantics is Compositional

$$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$$

We can reflect these parentheses in the definition:

$$
\begin{array}{l}
\mathit{evalExpr}\ (\mathit{Var}\ v) \quad = \lambda\ s \to \mathit{lookupFM}\ s\ v \\
\mathit{evalExpr}\ (\mathit{Value}\ \mathit{lit}) = \mathit{const}\ (\mathit{Just}\ (\mathit{litToVal}\ \mathit{lit})) \\
\mathit{evalExpr}\ (\mathit{Binary}\ (\mathit{MkArithOp}\ \mathit{Plus})\ \mathit{e1}\ \mathit{e2}) = \lambda\ s \to \\
\quad \mathbf{case}\ ((\mathit{evalExpr}\ \mathit{e1})\ s, (\mathit{evalExpr}\ \mathit{e2})\ s)\ \mathbf{of} \\
\qquad (\mathit{Just}\ (\mathit{ValInt}\ \mathit{v1}), \mathit{Just}\ (\mathit{ValInt}\ \mathit{v2})) \to \mathit{Just}\ (\mathit{ValInt}\ (\mathit{v1} + \mathit{v2})) \\
\qquad \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Nothing}
\end{array}
$$

Translating the last case back into mathematical notation:

$$[\![e_1 + e_2]\!]_E \quad := \quad \lambda\, s \;\to\; [\![e_1]\!]_E\ s + [\![e_2]\!]_E\ s$$

Denotational Semantics is Compositional

$$\mathit{evalExpr} :: \mathit{Expression} \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{Value1})$$

We can reflect these parentheses in the definition:

$$
\begin{array}{l}
\mathit{evalExpr}\ (\mathit{Var}\ v) \quad = \lambda\ s \to \mathit{lookupFM}\ s\ v \\
\mathit{evalExpr}\ (\mathit{Value}\ \mathit{lit}) = \mathit{const}\ (\mathit{Just}\ (\mathit{litToVal}\ \mathit{lit})) \\
\mathit{evalExpr}\ (\mathit{Binary}\ (\mathit{MkArithOp}\ \mathit{Plus})\ \mathit{e1}\ \mathit{e2}) = \lambda\ s \to \\
\quad \textbf{case}\ ((\mathit{evalExpr}\ \mathit{e1})\ s, (\mathit{evalExpr}\ \mathit{e2})\ s)\ \textbf{of} \\
\qquad (\mathit{Just}\ (\mathit{ValInt}\ \mathit{v1}), \mathit{Just}\ (\mathit{ValInt}\ \mathit{v2})) \to \mathit{Just}\ (\mathit{ValInt}\ (\mathit{v1} + \mathit{v2})) \\
\qquad \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Nothing}
\end{array}
$$

Translating the last case back into mathematical notation:

$$[\![e_1 + e_2]\!]_{\mathrm{E}} \quad := \quad \lambda\, s \ \to\ [\![e_1]\!]_{\mathrm{E}}\ s + [\![e_2]\!]_{\mathrm{E}}\ s \qquad \text{(using partial operations)}$$

Denotational Semantics is Compositional

evalExpr :: *Expression* → (*State1* → *Maybe Value1*)

We can reflect these parentheses in the definition:

```
evalExpr (Var v)       = λ s → lookupFM s v
evalExpr (Value lit) = const (Just (litToVal lit))
evalExpr (Binary (MkArithOp Plus) e1 e2) = λ s →
  case ((evalExpr e1) s, (evalExpr e2) s) of
    (Just (ValInt v1), Just (ValInt v2)) → Just (ValInt (v1 + v2))
    _                                    → Nothing
```

Translating the last case back into mathematical notation:

$$[\![e_1 + e_2]\!]_E \quad := \quad \lambda\, s \;\rightarrow\; [\![e_1]\!]_E\, s + [\![e_2]\!]_E\, s \qquad \text{(using partial operations)}$$

Compositional Semantics

The semantics of each syntactic construct is defined in terms of the semantics of its constituents.

Sequencing, Conditionals, Loops in Operational Semantics

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1;\ s_2) \Rightarrow \sigma_3}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad\qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1} \qquad \frac{\sigma(b) \Rightarrow \text{False} \qquad\qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad\qquad \sigma(s) \Rightarrow \sigma_1 \qquad\qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

Sequencing, Conditionals, Loops in Operational Semantics

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1;\ s_2) \Rightarrow \sigma_3}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False} \qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

The last operational semantics rule here is **<u>not</u> compositional!**

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \,;\, s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1* , *stmt2*]) =

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \,;\, s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \mathbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \mathbf{of}$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1*, *stmt2*]) = λ *s* → **case** (*interpStmt stmt1*) *s* **of**
  Just s1 →

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \mathbf{;}\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1*, *stmt2*]) = λ *s* → **case** (*interpStmt* *stmt1*) *s* **of**
  Just *s1* → (*interpStmt* *stmt2*) *s1*

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1*, *stmt2*]) = λ *s* → **case** (*interpStmt stmt1*) *s* **of**
  Just s1 → (*interpStmt stmt2*) *s1*
  Nothing → *Nothing*

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \textbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \textbf{of}$$
$$\quad \mathit{Just}\ \mathit{s1} \rightarrow (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1}$$
$$\quad \mathit{Nothing} \rightarrow \mathit{Nothing}$$

General case:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ s$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \textbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \textbf{of}$$
$$\quad \mathit{Just}\ \mathit{s1} \rightarrow (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1}$$
$$\quad \mathit{Nothing} \rightarrow \mathit{Nothing}$$

General case:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ s$$

$$\mathit{interpBlock} :: [\mathit{Statement}] \rightarrow (\mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1})$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ \mathbf{;}\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \mathbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \mathbf{of}$$
$$\mathit{Just}\ s1 \rightarrow (\mathit{interpStmt}\ \mathit{stmt2})\ s1$$
$$\mathit{Nothing} \rightarrow \mathit{Nothing}$$

General case:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ s$$

$$\mathit{interpBlock} :: [\mathit{Statement}] \rightarrow (\mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1})$$
$$\mathit{interpBlock}\ [\,] =$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \to \mathbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \mathbf{of}$$
$$\quad \mathit{Just}\ \mathit{s1} \to (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1}$$
$$\quad \mathit{Nothing} \to \mathit{Nothing}$$

General case:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \to \mathit{interpBlock}\ \mathit{stmts}\ s$$

$$\mathit{interpBlock} :: [\mathit{Statement}] \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{State1})$$
$$\mathit{interpBlock}\ [\,] = \mathit{Just}$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1*, *stmt2*]) = λ *s* → **case** (*interpStmt stmt1*) *s* **of**
  Just s1 → (*interpStmt stmt2*) *s1*
  Nothing → *Nothing*

General case:

interpStmt (*MkBlock stmts*) = λ *s* → *interpBlock stmts s*

interpBlock :: [*Statement*] → (*State1* → *Maybe State1*)
interpBlock [ ] = *Just*
interpBlock (*stmt* : *stmts*) =

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \,;\, s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$$\begin{array}{l} \mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \mathbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \mathbf{of} \\ \quad \mathit{Just}\ \mathit{s1} \rightarrow (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1} \\ \quad \mathit{Nothing} \rightarrow \mathit{Nothing} \end{array}$$

General case:

$$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ s$$

$$\begin{array}{l} \mathit{interpBlock} :: [\mathit{Statement}] \rightarrow (\mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1}) \\ \mathit{interpBlock}\ [\,] = \mathit{Just} \\ \mathit{interpBlock}\ (\mathit{stmt} : \mathit{stmts}) = \lambda\ s \rightarrow \end{array}$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \to \textbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \textbf{of}$

$\quad \mathit{Just}\ \mathit{s1} \to (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1}$

$\quad \mathit{Nothing} \to \mathit{Nothing}$

General case:

$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \to \mathit{interpBlock}\ \mathit{stmts}\ s$

$\mathit{interpBlock} :: [\mathit{Statement}] \to (\mathit{State1} \to \mathit{Maybe}\ \mathit{State1})$

$\mathit{interpBlock}\ [\,] = \mathit{Just}$

$\mathit{interpBlock}\ (\mathit{stmt} : \mathit{stmts}) = \lambda\ s \to \textbf{case}\ \mathit{interpStmt}\ \mathit{stmt}\ s\ \textbf{of}$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\ ;\ s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

```
interpStmt (MkBlock [stmt1, stmt2]) = λ s → case (interpStmt stmt1) s of
  Just s1 → (interpStmt stmt2) s1
  Nothing → Nothing
```

General case:

```
interpStmt (MkBlock stmts) = λ s → interpBlock stmts s


interpBlock :: [Statement] → (State1 → Maybe State1)
interpBlock [ ] = Just
interpBlock (stmt : stmts) = λ s → case interpStmt stmt s of
  Just s1 →
```

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \mathbin{;} s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

```
interpStmt (MkBlock [stmt1, stmt2]) = λ s → case (interpStmt stmt1) s of
  Just s1 → (interpStmt stmt2) s1
  Nothing → Nothing
```

General case:

```
interpStmt (MkBlock stmts) = λ s → interpBlock stmts s


interpBlock :: [Statement] → (State1 → Maybe State1)
interpBlock [ ] = Just
interpBlock (stmt : stmts) = λ s → case interpStmt stmt s of
  Just s1 → interpBlock stmts s1
```

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad\qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1\,;\,s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$\mathit{interpStmt}\ (\mathit{MkBlock}\ [\mathit{stmt1}, \mathit{stmt2}]) = \lambda\ s \rightarrow \mathbf{case}\ (\mathit{interpStmt}\ \mathit{stmt1})\ s\ \mathbf{of}$
$\quad \mathit{Just}\ \mathit{s1} \rightarrow (\mathit{interpStmt}\ \mathit{stmt2})\ \mathit{s1}$
$\quad \mathit{Nothing} \rightarrow \mathit{Nothing}$

General case:

$\mathit{interpStmt}\ (\mathit{MkBlock}\ \mathit{stmts}) = \lambda\ s \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ s$

$\mathit{interpBlock} :: [\mathit{Statement}] \rightarrow (\mathit{State1} \rightarrow \mathit{Maybe}\ \mathit{State1})$
$\mathit{interpBlock}\ [\,] = \mathit{Just}$
$\mathit{interpBlock}\ (\mathit{stmt} : \mathit{stmts}) = \lambda\ s \rightarrow \mathbf{case}\ \mathit{interpStmt}\ \mathit{stmt}\ s\ \mathbf{of}$
$\quad \mathit{Just}\ \mathit{s1} \rightarrow \mathit{interpBlock}\ \mathit{stmts}\ \mathit{s1}$
$\quad \mathit{Nothing} \rightarrow \mathit{Nothing}$

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

interpStmt (*Loop cond body*) =

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

interpStmt (*Loop cond body*) = λ *s* → **case** (*evalExpr cond*) *s* **of**

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

$\mathit{interpStmt}\ (\mathit{Loop}\ \mathit{cond}\ \mathit{body}) = \lambda\ s \rightarrow \textbf{case}\ (\mathit{evalExpr}\ \mathit{cond})\ s\ \textbf{of}$

$\quad \mathit{Just}\ (\mathit{ValBool}\ \textbf{False}) \rightarrow$

$\quad \mathit{Just}\ (\mathit{ValBool}\ \textbf{True}) \rightarrow$

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

interpStmt (*Loop cond body*) = λ *s* → **case** (*evalExpr cond*) *s* **of**
  Just (*ValBool* **False**) → *Just s*
  Just (*ValBool* **True**) →

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 →
```

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
```

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
    Nothing → Nothing
```

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}) \Rightarrow \sigma}$$

$$
\begin{array}{l}
\mathit{interpStmt}\ (\mathit{Loop}\ \mathit{cond}\ \mathit{body}) = \lambda\ s \rightarrow \textbf{case}\ (\mathit{evalExpr}\ \mathit{cond})\ s\ \textbf{of} \\
\quad \mathit{Just}\ (\mathit{ValBool}\ \textbf{False}) \rightarrow \mathit{Just}\ s \\
\quad \mathit{Just}\ (\mathit{ValBool}\ \textbf{True}) \rightarrow \textbf{case}\ (\mathit{interpStmt}\ \mathit{body})\ s\ \textbf{of} \\
\quad\quad \mathit{Just}\ s1 \rightarrow (\mathit{interpStmt}\ (\mathit{Loop}\ \mathit{cond}\ \mathit{body}))\ s1 \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\quad \mathit{Just}\ (\mathit{ValInt}\ i) \rightarrow \mathit{Nothing} \\
\quad \mathit{Nothing} \rightarrow \mathit{Nothing}
\end{array}
$$

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
    Nothing → Nothing
  Just (ValInt i) → Nothing
  Nothing → Nothing
```

This is **not compositional**

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
    Nothing → Nothing
  Just (ValInt i) → Nothing
  Nothing → Nothing
```

This is **not compositional**, but **recursive**

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
    Nothing → Nothing
  Just (ValInt i) → Nothing
  Nothing → Nothing
```

This is **not compositional**, but **recursive:**

“*interpStmt* (*Loop cond body*)”

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \textsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \textsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

```
interpStmt (Loop cond body) = λ s → case (evalExpr cond) s of
  Just (ValBool False) → Just s
  Just (ValBool True) → case (interpStmt body) s of
    Just s1 → (interpStmt (Loop cond body)) s1
    Nothing → Nothing
  Just (ValInt i) → Nothing
  Nothing → Nothing
```

This is **not compositional**, but **recursive:**

"*interpStmt (Loop cond body)*" occurs also on the right-hand side.

Recursive *Definitions*?

Recursive *Definitions*?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S\ )$$

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_{S} = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_{S}\ )$$

In ***mathematics:***

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$$

In ***mathematics:***

- “$x = \langle\langle$term not containing $x\rangle\rangle$”

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$"

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ s\ \textsf{\textbf{od}}]\!]_{\mathrm{S}} = F(\ [\![\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ s\ \textsf{\textbf{od}}]\!]_{\mathrm{S}}\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation**

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}} = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}}\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!
- "$x = F(x)$" can be used as **implicit definition** of x

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!
- "$x = F(x)$" can be used as **implicit definition** of x only if the equation is known to have **exactly one solution!**

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ s\ \textsf{\textbf{od}}]\!]_{\mathrm{S}} = F(\ [\![\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ s\ \textsf{\textbf{od}}]\!]_{\mathrm{S}}\ )$$

In ***mathematics:***

- “$x = \langle\langle$term not containing $x\rangle\rangle$” is an **explicit definition** of x
- “$x = F(x)$” is an **equation** that may have many solutions!
- “$x = F(x)$” can be used as **implicit definition** of x only if the equation is known to have **exactly one solution!** (I.e., writing such a definition produces a **proof obligation** of **well-definedness**)

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!
- "$x = F(x)$" can be used as **implicit definition** of x only if the equation is known to have **exactly one solution!** (I.e., writing such a definition produces a **proof obligation** of **well-definedness**)

In **denotational semantics**:

"$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_S\ )$"

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!
- "$x = F(x)$" can be used as **implicit definition** of x only if the equation is known to have **exactly one solution!** (I.e., writing such a definition produces a **proof obligation** of **well-definedness**)

In **denotational semantics**:

"$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S = F(\ [\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S\ )$" considered as equation in "$[\![\textbf{while } b \textbf{ do } s \textbf{ od}]\!]_S$"

Recursive *Definitions?*

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}} = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}}\ )$$

In ***mathematics:***

- "$x = \langle\langle$term not containing $x\rangle\rangle$" is an **explicit definition** of x
- "$x = F(x)$" is an **equation** that may have many solutions!
- "$x = F(x)$" can be used as **implicit definition** of x only if the equation is known to have **exactly one solution!** (I.e., writing such a definition produces a **proof obligation** of **well-definedness**)

In **denotational semantics**:

"$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}} = F(\ [\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}}\ )$" considered as equation in

"$[\![\textbf{while}\ b\ \textbf{do}\ s\ \textbf{od}]\!]_{\mathrm{S}}$" usually has **many** solutions!

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$\mathit{fact}\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$\mathit{fact}\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Principle of **extensionality:** two functions are equal iff all their resp. applications to the same argument are equal:

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$\mathit{fact}\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Principle of **extensionality:** two functions are equal iff all their resp. applications to the same argument are equal:

$$\mathit{fact} = \lambda\ n \rightarrow \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$\mathit{fact}\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Principle of **extensionality:** two functions are equal iff all their resp. applications to the same argument are equal:

$$\mathit{fact} = \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Reverse β-reduction to isolate the RHS occurrence of *fact*:

$$\mathit{fact} = (\lambda\ f \to \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1))\ \mathit{fact}$$

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$\mathit{fact}\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Principle of **extensionality:** two functions are equal iff all their resp. applications to the same argument are equal:

$$\mathit{fact} = \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1)$$

Reverse β-reduction to isolate the RHS occurrence of *fact*:

$$\mathit{fact} = (\lambda\ f \to \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1))\ \mathit{fact}$$

Defining (via an explicit definition)

$$\tau = \lambda\ f \to \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1)$$

Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$fact\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * fact\ (n-1)$$

Principle of **extensionality:** two functions are equal iff all their resp. applications to the same argument are equal:

$$fact = \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * fact\ (n-1)$$

Reverse β-reduction to isolate the RHS occurrence of *fact*:

$$fact = (\lambda\ f \to \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1))\ fact$$

Defining (via an explicit definition)

$$\tau = \lambda\ f \to \lambda\ n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1)$$

we recognise a **fixedpoint equation** (stating that *fact* is a fix edpointof τ):

$$fact = \tau\ fact$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$\tau^0 \perp \quad = \quad \perp \quad = \quad \{\}$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$
\begin{array}{lllll}
\tau^0 \; \bot & = & \bot & = & \{\} \\
\tau^1 \; \bot & = & \tau \; \bot & = & \{0 \mapsto 1\}
\end{array}
$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$
\begin{aligned}
\tau^0\ \bot &= \bot &&= \{\} \\
\tau^1\ \bot &= \tau\ \bot &&= \{0 \mapsto 1\} \\
\tau^2\ \bot &= \tau\ (\tau\ \bot\ ) &&= \{0 \mapsto 1,\ 1 \mapsto 1\}
\end{aligned}
$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$
\begin{aligned}
\tau^0\ \bot &= \bot &&= \{\} \\
\tau^1\ \bot &= \tau\ \bot &&= \{0 \mapsto 1\} \\
\tau^2\ \bot &= \tau\,(\tau\ \bot\,) &&= \{0 \mapsto 1, 1 \mapsto 1\} \\
\tau^3\ \bot &= \tau\,(\tau\,(\tau\ \bot\,)) &&= \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\}
\end{aligned}
$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$
\begin{array}{lcll}
\tau^0 \perp & = & \perp & = \{\} \\
\tau^1 \perp & = & \tau \perp & = \{0 \mapsto 1\} \\
\tau^2 \perp & = & \tau\,(\tau \perp) & = \{0 \mapsto 1, 1 \mapsto 1\} \\
\tau^3 \perp & = & \tau\,(\tau\,(\tau \perp)) & = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\} \\
\tau^4 \perp & = & \tau\,(\tau\,(\tau\,(\tau \perp))) & = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\}
\end{array}
$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$\begin{array}{lllll}
\tau^0\ \bot & = & \bot & = & \{\} \\
\tau^1\ \bot & = & \tau\ \bot & = & \{0\mapsto 1\} \\
\tau^2\ \bot & = & \tau\,(\tau\ \bot\,) & = & \{0\mapsto 1,\ 1\mapsto 1\} \\
\tau^3\ \bot & = & \tau\,(\tau\,(\tau\ \bot\,)) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2\} \\
\tau^4\ \bot & = & \tau\,(\tau\,(\tau\,(\tau\ \bot\,))) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2,\ 3\mapsto 6\} \\
\tau^5\ \bot & = & \tau\,(\tau\,(\tau\,(\tau\,(\tau\ \bot\,)))) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2,\ 3\mapsto 6,\ 4\mapsto 24\}
\end{array}$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$\begin{array}{lllll}
\tau^0 \perp & = & \perp & = & \{\} \\
\tau^1 \perp & = & \tau \perp & = & \{0 \mapsto 1\} \\
\tau^2 \perp & = & \tau\,(\tau \perp) & = & \{0 \mapsto 1, 1 \mapsto 1\} \\
\tau^3 \perp & = & \tau\,(\tau\,(\tau \perp)) & = & \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\} \\
\tau^4 \perp & = & \tau\,(\tau\,(\tau\,(\tau \perp))) & = & \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\} \\
\tau^5 \perp & = & \tau\,(\tau\,(\tau\,(\tau\,(\tau \perp)))) & = & \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, 4 \mapsto 24\}
\end{array}$$

In addition:

$$\perp \sqsubseteq \tau \perp \sqsubseteq \tau^2 \perp \sqsubseteq \tau^3 \perp \sqsubseteq \tau^4 \perp \sqsubseteq \tau^5 \perp \sqsubseteq \ldots \sqsubseteq \tau^{1,000,000} \perp \sqsubseteq \ldots$$

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$\begin{array}{llll}
\tau^0\ \bot & = & \bot & = \{\} \\
\tau^1\ \bot & = & \tau\ \bot & = \{0 \mapsto 1\} \\
\tau^2\ \bot & = & \tau\ (\tau\ \bot) & = \{0 \mapsto 1, 1 \mapsto 1\} \\
\tau^3\ \bot & = & \tau\ (\tau\ (\tau\ \bot)) & = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\} \\
\tau^4\ \bot & = & \tau\ (\tau\ (\tau\ (\tau\ \bot))) & = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\} \\
\tau^5\ \bot & = & \tau\ (\tau\ (\tau\ (\tau\ (\tau\ \bot)))) & = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, 4 \mapsto 24\}
\end{array}$$

In addition:

$$\bot \sqsubseteq \tau\ \bot \sqsubseteq \tau^2\ \bot \sqsubseteq \tau^3\ \bot \sqsubseteq \tau^4\ \bot \sqsubseteq \tau^5\ \bot \sqsubseteq \ldots \sqsubseteq \tau^{1,000,000}\ \bot \sqsubseteq \ldots$$

Iterated application of τ yields better and better **finite approximations**!

Fixedpoint Approximation

For the functional τ associated with the definition of the factorial function *fact*, we observe:

$$
\begin{array}{lllll}
\tau^0\ \bot & = & \bot & = & \{\} \\
\tau^1\ \bot & = & \tau\ \bot & = & \{0\mapsto 1\} \\
\tau^2\ \bot & = & \tau\,(\tau\ \bot\,) & = & \{0\mapsto 1,\ 1\mapsto 1\} \\
\tau^3\ \bot & = & \tau\,(\tau\,(\tau\ \bot\,)) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2\} \\
\tau^4\ \bot & = & \tau\,(\tau\,(\tau\,(\tau\ \bot\,))) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2,\ 3\mapsto 6\} \\
\tau^5\ \bot & = & \tau\,(\tau\,(\tau\,(\tau\,(\tau\ \bot\,)))) & = & \{0\mapsto 1,\ 1\mapsto 1,\ 2\mapsto 2,\ 3\mapsto 6,\ 4\mapsto 24\}
\end{array}
$$

In addition:

$$\bot\ \sqsubseteq \tau\ \bot\ \sqsubseteq \tau^2\ \bot\ \sqsubseteq \tau^3\ \bot\ \sqsubseteq \tau^4\ \bot\ \sqsubseteq \tau^5\ \bot\ \sqsubseteq \ldots \sqsubseteq \tau^{1{,}000{,}000}\ \bot\ \sqsubseteq \ldots$$

Iterated application of τ yields better and better **finite approximations**!

The union of all these approximations **is** the factorial function.

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\, f$.

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\, f$.
- The “functional” τ is extracted from that equation.

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\ f$.
- The "functional" τ is extracted from that equation.
- Use τ for **fixedpoint iteration**

  $\bot \sqsubseteq \tau\ \bot \sqsubseteq \tau^2\ \bot \sqsubseteq \tau^3\ \bot \sqsubseteq \ldots$

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\ f$.
- The "functional" τ is extracted from that equation.
- Use τ for **fixedpoint iteration**
  $\bot \ \sqsubseteq \tau\ \bot \ \sqsubseteq \tau^2\ \bot \ \sqsubseteq \tau^3\ \bot \ \sqsubseteq \dots$
- The semantics of f is the least upper bound of this chain:

$$[\![f]\!] = \bigcup\{k : \mathbb{N} \bullet \tau^k\ \bot\}$$

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\, f$.
- The "functional" τ is extracted from that equation.
- Use τ for **fixedpoint iteration**

  $\perp \;\sqsubseteq \tau\ \perp\; \sqsubseteq \tau^2\ \perp\; \sqsubseteq \tau^3\ \perp\; \sqsubseteq \ldots$
- The semantics of f is the least upper bound of this chain:

$$[\![f]\!] = \bigcup\{k : \mathbb{N} \;\bullet\; \tau^k\ \perp\}$$

- This least upper bound is the **least fixedpoint** of τ

Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function f is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau\ f$.
- The "functional" τ is extracted from that equation.
- Use τ for **fixedpoint iteration**

  $\bot \sqsubseteq \tau\ \bot \sqsubseteq \tau^2\ \bot \sqsubseteq \tau^3\ \bot \sqsubseteq \ldots$
- The semantics of f is the least upper bound of this chain:

$$[\![f]\!] = \bigcup \{ k : \mathbb{N} \bullet \tau^k\ \bot \}$$

- This least upper bound is the **least fixedpoint** of τ
- We write " $Y\ \tau$ " for the **least fixedpoint** of τ.

while-Loop Semantics

$$[\![\_]\!]_S \quad : \; Stmt \to (State \nrightarrow State)$$

For $p : Stmt$ and $e : Expr$:

$$[\![\textbf{while } e \textbf{ do } p]\!]_S$$

$$= \mathbf{Y}\left(\lambda f : State \nrightarrow State \bullet \lambda s : State \bullet \begin{cases} f([\![p]\!]_S(s)) & \text{if } [\![e]\!]_E(s) = \mathsf{True} \\ s & \text{if } [\![e]\!]_E(s) = \mathsf{False} \\ \bot & \text{otherwise} \end{cases} \right)$$

Example Statement Semantics

$[\![\textbf{while}\ \text{True}\ \textbf{do}\ \textbf{skip}]\!]_S =$

Example Statement Semantics

$$[\![\textbf{while}\ \text{True}\ \textbf{do}\ \textbf{skip}]\!]_{\mathrm{S}} = \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet \lambda s : \mathit{State} \bullet f([\![\textbf{skip}]\!]_{\mathrm{S}}(s)))$$

Example Statement Semantics

$$
\begin{aligned}
[\![\textbf{while}\ \mathrm{True}\ \textbf{do}\ \textbf{skip}]\!]_{\mathrm{S}} &= \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet \lambda s : \mathit{State} \bullet f([\![\textbf{skip}]\!]_{\mathrm{S}}(s))) \\
&= \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet \lambda s : \mathit{State} \bullet f(s))
\end{aligned}
$$

Example Statement Semantics

$$
\begin{aligned}
[\![\textbf{while}\ \mathrm{True}\ \textbf{do}\ \textbf{skip}]\!]_S &= \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet \lambda s : \mathit{State} \bullet f([\![\textbf{skip}]\!]_S(s))) \\
&= \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet \lambda s : \mathit{State} \bullet f(s)) \\
&= \mathbf{Y}(\lambda f : \mathit{State} \nrightarrow \mathit{State} \bullet f)
\end{aligned}
$$

Example Statement Semantics

$$
\begin{aligned}
[\![\textbf{while}\ \text{True}\ \textbf{do}\ \textbf{skip}]\!]_S &= \mathbf{Y}(\lambda f : State \nrightarrow State \bullet \lambda s : State \bullet f([\![\textbf{skip}]\!]_S(s))) \\
&= \mathbf{Y}(\lambda f : State \nrightarrow State \bullet \lambda s : State \bullet f(s)) \\
&= \mathbf{Y}(\lambda f : State \nrightarrow State \bullet f) \\
&= \bot
\end{aligned}
$$

Example Statement Semantics

$$
\begin{aligned}
[\![\textbf{while}\ \text{True}\ \textbf{do}\ \textbf{skip}]\!]_S &= \mathbf{Y}(\lambda f : State \mathrel{+\!\!\!\to} State \bullet \lambda s : State \bullet f([\![\textbf{skip}]\!]_S(s))) \\
&= \mathbf{Y}(\lambda f : State \mathrel{+\!\!\!\to} State \bullet \lambda s : State \bullet f(s)) \\
&= \mathbf{Y}(\lambda f : State \mathrel{+\!\!\!\to} State \bullet f) \\
&= \bot
\end{aligned}
$$

For $k : \mathbb{N}$, we have:

$$[\![\textbf{while}\ n > 0\ \textbf{do}\ (r \mathbin{\textbf{:=}} n * r\ \textbf{;}\ n \mathbin{\textbf{:=}} n - 1)]\!]_S(\{n \mapsto k, r \mapsto 1\}) = \{n \mapsto 0, r \mapsto k!\}$$

Summary

Summary

- The **syntax** of programming languages is best described mathematically.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.
  - Much of the complexity can be hidden in simple calculi like Hoare logic.

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.
  - Much of the complexity can be hidden in simple calculi like Hoare logic.
- Some programming languages have a "**more mathematical semantics**"

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.
  - Much of the complexity can be hidden in simple calculi like Hoare logic.
- Some programming languages have a "**more mathematical semantics**": mathematical rules of reasoning can be applied directly to program constructs

Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.
  - Much of the complexity can be hidden in simple calculi like Hoare logic.
- Some programming languages have a "**more mathematical semantics**": mathematical rules of reasoning can be applied directly to program constructs
- **Programming is a mathematical activity**

Semantics with Exceptions — Simple Statements

$$[\![\_]\!]_S : \; \mathit{Stmt} \to (\mathit{Store} \nrightarrow (\mathit{Store} + (\mathit{Store} \times \mathit{Num})))$$

Semantics with Exceptions — Simple Statements

$$[\![\_]\!]_S : \ \mathit{Stmt} \to (\mathit{Store} \nrightarrow (\mathit{Store} + (\mathit{Store} \times \mathit{Num})))$$

$$[\![\mathbf{skip}]\!]_S = \mathit{Left}$$

Semantics with Exceptions — Simple Statements

$$[\![\_]\!]_{\mathrm{S}} : \ \mathit{Stmt} \to (\mathit{Store} \nrightarrow (\mathit{Store} + (\mathit{Store} \times \mathit{Num})))$$

$$[\![\mathbf{skip}]\!]_{\mathrm{S}} = \mathit{Left}$$

$$[\![s_1 \ \mathbf{;} \ s_2]\!]_{\mathrm{S}} \ (s) \ = \begin{cases} [\![s_2]\!]_{\mathrm{S}} \ (t) & \text{if } [\![s_1]\!]_{\mathrm{S}}(s) = \mathit{Left} \ t \\ \mathit{Right}(t, e) & \text{if } [\![s_1]\!]_{\mathrm{S}}(s) = \mathit{Right} \ (t, e) \end{cases}$$

Semantics with Exceptions — Simple Statements

$$[\![\_]\!]_S : \ Stmt \to (Store \nrightarrow (Store + (Store \times Num)))$$

$$[\![\textbf{skip}]\!]_S = Left$$

$$[\![s_1 \ \textbf{;} \ s_2]\!]_S \ (s) \ = \begin{cases} [\![s_2]\!]_S \ (t) & \text{if } [\![s_1]\!]_S(s) = Left \ t \\ Right(t, e) & \text{if } [\![s_1]\!]_S(s) = Right \ (t, e) \end{cases}$$

$$[\![\textbf{try} \ s_1 \ \textbf{catch(} \ i \ \textbf{)} \ s_2]\!]_S \ (s) \ = \begin{cases} t & \text{if } [\![s_1]\!]_S(s) = Left \ t \\ [\![s_2]\!]_S(t \oplus \{i \mapsto e\}) & \text{if } [\![s_1]\!]_S(s) = Right \ (t, e) \\ \bot & \text{if } s \notin \mathsf{dom} \ [\![s_1]\!]_S \end{cases}$$

Semantics with Exceptions — Expressions

$$Expr \to (Store \to (Val + Num))$$

Semantics with Exceptions — Expressions

$$Expr \rightarrow (Store \rightarrow (Val + Num))$$

$$[\![\textbf{throw}\ e]\!]_{\mathrm{S}}(s) \quad = \quad \begin{cases} Right\ (s, val) & \text{if } [\![e]\!]_{\mathrm{E}}(s) = Left\ val \\ Right\ (s, exc) & \text{if } [\![e]\!]_{\mathrm{E}}(s) = Right\ exc \end{cases}$$

Semantics with Exceptions — Expressions

$$Expr \to (Store \to (Val + Num))$$

$$[\![\textbf{throw } e]\!]_S(s) = \begin{cases} Right\ (s, val) & \text{if } [\![e]\!]_E(s) = Left\ val \\ Right\ (s, exc) & \text{if } [\![e]\!]_E(s) = Right\ exc \end{cases}$$

$$[\![v \textbf{ := } e]\!]_S(s) = \begin{cases} Left\ (s \oplus \{v \mapsto val\}) & \text{if } [\![e]\!]_E(s) = Left\ val \\ Right\ (s, exc) & \text{if } [\![e]\!]_E(s) = Right\ exc \end{cases}$$

Semantics with Exceptions — Expressions

$$Expr \rightarrow (Store \rightarrow (Val + Num))$$

$$[\![\textbf{throw } e]\!]_S(s) = \begin{cases} Right\ (s, val) & \text{if } [\![e]\!]_E(s) = Left\ val \\ Right\ (s, exc) & \text{if } [\![e]\!]_E(s) = Right\ exc \end{cases}$$

$$[\![v \textbf{ := } e]\!]_S(s) = \begin{cases} Left\ (s \oplus \{v \mapsto val\}) & \text{if } [\![e]\!]_E(s) = Left\ val \\ Right\ (s, exc) & \text{if } [\![e]\!]_E(s) = Right\ exc \end{cases}$$

$$[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2]\!]_S(s) = \begin{cases} [\![s_1]\!]_S(s) & \text{if } [\![b]\!]_E(s) = Left\ \mathsf{True} \\ [\![s_2]\!]_S(s) & \text{if } [\![b]\!]_E(s) = Left\ \mathsf{False} \\ Right\ (s, exc_{\mathbb{B}}) & \text{if } [\![b]\!]_E(s) = Left\ n \wedge n \in Num \\ Right\ (s, exc) & \text{if } [\![b]\!]_E(s) = Right\ exc \end{cases}$$

while-Semantics with Exceptions

while-Semantics with Exceptions

Exercise!

Output

Semantic Domains for simple imperative programs with **print** statements:

Output

Semantic Domains for simple imperative programs with **print** statements:

SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Store	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores

Output

Semantic Domains for simple imperative programs with **print** statements:

SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Store	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores
State	$= \mathit{Store}^{\perp} \times [\mathit{Num}]$	**states including output**

Output

Semantic Domains for simple imperative programs with **print** statements:

SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Store	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores
State	$= \mathit{Store}^{\perp} \times [\mathit{Num}]$	**states including output**
Val	$= \mathit{SVal}$	values
$\mathit{Store} \nrightarrow \mathit{Val}$		(expression semantics)
$\mathit{State} \rightarrow \mathit{State}$		(**new** statement semantics)

Output

Semantic Domains for simple imperative programs with **print** statements:

$SVal$	$= Bool + Num$	storable values
$Store$	$= Id \nrightarrow SVal$	(simple) stores
$State$	$= Store^{\perp} \times [Num]$	**states including output**
Val	$= SVal$	values
$Store \nrightarrow Val$		(expression semantics)
$State \rightarrow State$		(**new** statement semantics)

In case of program errors or nontermination, **previous output is not lost**!

$$[\![\mathbf{print}\ e]\!]_S = \lambda\,(s, ns) : State \bullet \begin{cases} (s, n{:}ns) & \text{if } n = [\![e]\!]_E(s) \in Num \\ (\perp, ns) & \text{otherwise} \end{cases}$$

Output

Semantic Domains for simple imperative programs with **print** statements:

$SVal$	$= Bool + Num$	storable values
$Store$	$= Id \nrightarrow SVal$	(simple) stores
$State$	$= Store^{\perp} \times [Num]$	**states including output**
Val	$= SVal$	values
$Store \nrightarrow Val$		(expression semantics)
$State \rightarrow State$		(**new** statement semantics)

In case of program errors or nontermination, **previous output is not lost**!

$$[\![\textbf{print}\ e]\!]_S = \lambda\,(s, ns) : State \bullet \begin{cases} (s, n{:}ns) & \text{if } n = [\![e]\!]_E(s) \in Num \\ (\perp, ns) & \text{otherwise} \end{cases}$$

Note: statement semantics here is ***oversimplified*** — fixpoint construction in $State \rightarrow State$ does not work, except with Haskell-like list domains.

Input

Input

- **Output** is reflected by the introduction of a state component representing **past output**:

$$State = Store^{\perp} \times [Num]$$

Input

- **Output** is reflected by the introduction of a state component representing **past output**:

$$State = Store^{\perp} \times [Num]$$

- (Additional) **Input** is reflected by the introduction of a state component representing **future input**:

$$State = Store^{\perp} \times [Num] \times [Num]$$

Input

- **Output** is reflected by the introduction of a state component representing **past output**:

$$State = Store^{\perp} \times [Num]$$

- (Additional) **Input** is reflected by the introduction of a state component representing **future input**:

$$State = Store^{\perp} \times [Num] \times [Num]$$

$$[\![\textbf{read } v]\!]_S =$$

$$\lambda\,(s, outs, ins) : State \bullet \begin{cases} (s \oplus \{v \mapsto in\}, outs, ins') & \text{if } ins = in{:}ins' \\ (\perp, outs, ins) & \text{if } ins = [] \end{cases}$$

Scope

Nested scopes with shadowing of identifiers are modelled as **stacks** (lists) **of environments**:

Env	$= Id \nrightarrow SVal^{\perp}$	**environments** (with $\perp$ for uninit. var.)
$Store$	$= [Env]$	**stores**
$State$	$= Store^{\perp} \times [Num] \times [Num]$	**states including I/O**

Records

Semantic Domains: Only **storable values** change:

$SVal$	$= Bool + Num + (Id \mathrel{+\!\!\!\!\twoheadrightarrow} SVal)$	storable values
$State$	$= Id \mathrel{+\!\!\!\!\rightarrow} SVal$	(simple) stores
$State \mathrel{+\!\!\!\!\rightarrow} SVal$		(expression semantics)
$State \mathrel{+\!\!\!\!\rightarrow} State$		(statement semantics)

New record field expressions:

$$[\![e.\, f]\!]_E = \lambda\, s : State \bullet ([\![e]\!]_E\,(s))\; f$$

New record construction expressions (not in C or Oberon, but e.g. in Ada):

$$[\![\mathbf{record}(f_1 = e_1, \ldots, f_n = e_n)]\!]_E = \lambda\, s : State \bullet \{ f_1 \mapsto [\![e_1]\!]_E(s), \ldots, f_n \mapsto [\![e_n]\!]_E(s) \}$$

New record field assignment statements:

$$[\![r.\, f \mathbin{:=} e]\!]_S = \lambda\, s : State \bullet s \oplus \{ r \mapsto ((s\ r) \oplus \{ f \mapsto [\![e]\!]_E(s) \}) \}$$

Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics Semantics

Lecture 2

Axiomatic Semantics

Axiomatic Semantics

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings

Axiomatic Semantics

Derivation of judgements written as "Hoare triples"

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q

Axiomatic Semantics

Derivation of judgements written as "Hoare triples"

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***,
then its terminating state satisfies Q

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: ***If*** S starts in a state satisfying P, ***then it terminates*** and its terminating state satisfies Q

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics

Derivation of judgements written as "Hoare triples"

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***,
then its terminating state satisfies Q

("terminates" means "terminates without run-time error")

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q
— “S **is totally correct with respect to** P and Q”

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***,
then its terminating state satisfies Q
— “S **is partially correct with respect to** P and Q”

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements

- Axiomatic semantics relates **conditions on states** via statements

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Therefore:

- Operational semantics facilitates investigation of examples

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements

- Axiomatic semantics relates **conditions on states** via statements

Therefore:

- Operational semantics facilitates investigation of examples (“***testing***”)

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Therefore:

- Operational semantics facilitates investigation of examples (“***testing***”)
- Axiomatic semantics facilitates relating a program with its specification

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Therefore:

- Operational semantics facilitates investigation of examples (“***testing***”)
- Axiomatic semantics facilitates relating a program with its specification — **verification**

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition P **holds**, or **is valid**, in state s"

For example: • $\{x \mapsto 5, y \mapsto 7\} \models x > 0$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition P **holds**, or **is valid**, in state s"

For example:

- $\{x \mapsto 5, y \mapsto 7\} \models x > 0$
- $\{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition P **holds**, or **is valid**, in state s"

For example:

- $\{x \mapsto 5, y \mapsto 7\} \models x > 0$
- $\{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55$
- $\{x \mapsto 5, y \mapsto 7\} \not\models x > y$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition *P* **holds**, or **is valid**, in state *s*"

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: ***For all*** states σ_1 and σ_2

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: ***For all*** states σ_1 and σ_2,
if $\sigma_1 \models P$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- " $s \models P$ " means "condition P **holds**, or **is valid**, in state s"

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2,
if $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- " $s \models P$ " means "condition P **holds**, or **is valid**, in state s"

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2,
if $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition P **holds**, or **is valid**, in state s"

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2,
if $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P, ***then it terminates*** and its terminating state satisfies Q

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2, ***if*** $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P, ***then it terminates*** and its terminating state satisfies Q

I.e.: For all states σ_1

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2,
if $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q

I.e.: For all states σ_1, ***if*** $\sigma_1 \models P$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- "$s \models P$" means "condition P **holds**, or **is valid**, in state s"

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2, ***if*** $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P, ***then it terminates*** and its terminating state satisfies Q

I.e.: For all states σ_1, ***if*** $\sigma_1 \models P$, ***then there is*** a state σ_2

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- " $s \models P$ " means "condition P **holds**, or **is valid**, in state s"

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2,
if $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P,
then it terminates and its terminating state satisfies Q

I.e.: For all states σ_1, ***if*** $\sigma_1 \models P$,
then there is a state σ_2
such that $\sigma_1(S) \Rightarrow \sigma_2$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: ***If*** S starts in a state satisfying P and ***terminates***, ***then*** its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2, ***if*** $\sigma_1 \models P$ ***and*** $\sigma_1(S) \Rightarrow \sigma_2$, ***then*** $\sigma_2 \models Q$

Total correctness: ***If*** S starts in a state satisfying P, ***then it terminates*** and its terminating state satisfies Q

I.e.: For all states σ_1, ***if*** $\sigma_1 \models P$, ***then there is*** a state σ_2 ***such that*** $\sigma_1(S) \Rightarrow \sigma_2$, ***and*** $\sigma_2 \models Q$

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**
- Proving partial correctness therefore does not need operational semantics

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — ***well-ordered*** sets)

Proving Partial and Total Correctness

Total correctness of $\{P\}\ S\ \{Q\}$

is equivalent to

partial correctness of $\{P\}\ S\ \{Q\}$ ***together with*** the fact that S terminates when started in a state satisfying P

$\Rightarrow$ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — ***well-ordered*** sets)

Unless explicitly mentioned, we read " $\{P\}\ S\ \{Q\}$ " as meaning **partial correctness**.

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

$$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

$$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

$$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Sequence:

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1;\ S_2\{Q\}}$$

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

$$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Sequence:

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1;\ S_2\{Q\}}$$

Conditional:

$$\frac{\{P \land b\}S_1\{Q\} \qquad \{P \land \neg b\}S_2\{Q\}}{\{P\}\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{Q\}}$$

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

$$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Sequence:

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1;\ S_2\{Q\}}$$

Conditional:

$$\frac{\{P \wedge b\}S_1\{Q\} \qquad \{P \wedge \neg b\}S_2\{Q\}}{\{P\}\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{Q\}}$$

while-Loop:

$$\frac{\{INV \wedge b\}S\{INV\}}{\{INV\}\textbf{while } b \textbf{ do } S \textbf{ od}\{INV \wedge \neg b\}}$$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$
- $\{n + 1 = 2\}x := n + 1\{x = 2\}$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$
- $\{n + 1 = 2\}x := n + 1\{x = 2\}$

Typically, Hoare triples are derived starting from the ***postcondition***

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$
- $\{n + 1 = 2\}x := n + 1\{x = 2\}$

Typically, Hoare triples are derived starting from the ***postcondition***

— **backward reasoning**.

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$
- $\{n + 1 = 2\}x := n + 1\{x = 2\}$

Typically, Hoare triples are derived starting from the ***postcondition***

— **backward reasoning**.

Considering this axiom schema as a way to ***calculate*** a precondition from assignment and postcondition, it calculates the **weakest precondition** that completes a valid Hoare triple.

Example Verification

$$\{\text{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

Example Annotated Program

$$\{\textsf{True}\} \;\Rightarrow\; \{0 = \sum_{i=1}^{0} i\}$$

$$k := 0\mathbf{;} \qquad \{0 = \sum_{i=1}^{k} i\}$$

$$s := 0\mathbf{;} \qquad \{s = \sum_{i=1}^{k} i\}$$

while $k \neq n$

do $\{s = \sum_{i=1}^{k} i \wedge k \neq n\} \Rightarrow \{s + k + 1 = \sum_{i=1}^{k+1} i\}$

$$k := k + 1\mathbf{;} \qquad \{s + k = \sum_{i=1}^{k} i\}$$

$$s := s + k \qquad \{s = \sum_{i=1}^{k} i\}$$

od

$$\{s = \sum_{i=1}^{k} i \wedge k = n\} \;\Rightarrow\; \{s = \sum_{i=1}^{n} i\}$$

Example Verification

$$\{\textsf{True}\}k := 0\textbf{;}\ s := 0\textbf{;}\ \textbf{while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

Example Verification

$$\{\mathsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\mathsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

Example Verification

$$\{\textsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\textsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\textsf{True}\}k := 0\textbf{;}\ s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge$$

Example Verification

$$\{\textsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k+1\textbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\textsf{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k+1\textbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\textsf{True}\}k := 0\textbf{;}\ s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge \{s = \sum_{i=1}^{k} i\}\textbf{while}\ k \neq n\ \textbf{do}\ k := k+1\textbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

Example Verification

$$\{\text{True}\}k := 0; s := 0; \textbf{while } k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\text{True}\}k := 0; s := 0; \textbf{while } k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0; s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge \{s = \sum_{i=1}^{k} i\}\textbf{while } k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge$$

$$\wedge$$

Example Verification

$$\{\text{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\text{True}\}k := 0\textbf{;}\ s := 0\textbf{; while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0\textbf{;}\ s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge \{s = \sum_{i=1}^{k} i\}\textbf{while}\ k \neq n\ \textbf{do}\ k := k + 1\textbf{;}\ s := s + k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge \{0 = \sum_{i=1}^{k} i\}s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge$$

Example Verification

$$\{\mathsf{True}\}k := 0\mathbf{;}\ s := 0\mathbf{;}\ \textbf{while}\ k \neq n\ \textbf{do}\ k := k+1\mathbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{n} i\}$$

$$\Leftarrow \{\mathsf{True}\}k := 0\mathbf{;}\ s := 0\mathbf{;}\ \textbf{while}\ k \neq n\ \textbf{do}\ k := k+1\mathbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\mathsf{True}\}k := 0\mathbf{;}\ s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge \{s = \sum_{i=1}^{k} i\}\textbf{while}\ k \neq n\ \textbf{do}\ k := k+1\mathbf{;}\ s := s+k\ \textbf{od}\{s = \sum_{i=1}^{k} i \wedge k = n\}$$

$$\Leftarrow \{\mathsf{True}\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge \{0 = \sum_{i=1}^{k} i\}s := 0\{s = \sum_{i=1}^{k} i\}$$

$$\wedge \{s = \sum_{i=1}^{k} i \wedge k \neq n\}k := k+1\mathbf{;}\ s := s+k\{s = \sum_{i=1}^{k} i\}$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\}$$

$$\wedge \ \text{True}$$

$$\wedge \ \{s = \sum_{i=1}^{k} i \wedge k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge \ \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\}$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\}$$

$$\wedge \ \text{True}$$

$$\wedge \ \{s = \sum_{i=1}^{k} i \wedge k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge \ \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\}$$

$$\Leftarrow \ \text{True}$$

$$\wedge$$

$$\wedge$$

$$\wedge$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

$$\wedge\ \{s = \sum_{i=1}^{k} i \wedge k \neq n\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \{s + k = \sum_{i=1}^{k} i\}s := s + k\{s = \sum_{i=1}^{k} i\}$$

$$\Leftarrow \text{True}$$

$$\wedge\ s = \sum_{i=1}^{k} i \wedge k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i$$

$$\wedge$$

$$\wedge$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

$$\wedge\ \{s = \sum_{i=1}^{k} i \wedge k \neq n\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \{s + k = \sum_{i=1}^{k} i\}s := s + k\{s = \sum_{i=1}^{k} i\}$$

$$\Leftarrow \text{True}$$

$$\wedge\ s = \sum_{i=1}^{k} i \wedge k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i$$

$$\wedge\ \{s + k + 1 = \sum_{i=1}^{k+1} i\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

$$\wedge\ \{s = \sum_{i=1}^{k} i \wedge k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\}$$

$$\Leftarrow\ \text{True}$$

$$\wedge\ s = \sum_{i=1}^{k} i \wedge k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i$$

$$\wedge\ \{s + k + 1 = \sum_{i=1}^{k+1} i\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^{0} i\ ) \quad \wedge \quad \{0 = \sum_{i=1}^{0} i\}k := 0\{0 = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

$$\wedge\ \{s = \sum_{i=1}^{k} i \wedge k \neq n\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \{s + k = \sum_{i=1}^{k} i\}s := s + k\{s = \sum_{i=1}^{k} i\}$$

$$\Leftarrow \text{True}$$

$$\wedge\ s = \sum_{i=1}^{k} i \wedge k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i$$

$$\wedge\ \{s + k + 1 = \sum_{i=1}^{k+1} i\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}$$

$$\wedge\ \text{True}$$

$$\Leftarrow \text{True}$$

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Given a loop "**while** b **do** S **od**" and a postcondition Q

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

  Auxiliary variables used in a loop are usually involved in the invariant!

  Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that

  - $Q' \Rightarrow Q$

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

  Auxiliary variables used in a loop are usually involved in the invariant!

  Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that

  - $Q' \Rightarrow Q$ (strengthening)

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that

- $Q' \Rightarrow Q$ (strengthening)
- Q' involves all auxiliary variables

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it
- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration
- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

  Auxiliary variables used in a loop are usually involved in the invariant!

  Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that
  - $Q' \Rightarrow Q$ (strengthening)
  - Q' involves all auxiliary variables — **generalisation!**

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:

  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Given a loop "**while** b **do** S **od**" and a postcondition Q, use the consequence rule to strengthen Q to Q', such that

- $Q' \Rightarrow Q$ (strengthening)
- Q' involves all auxiliary variables — **generalisation!**
- Q' is of shape $INV \wedge \neg b$

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$
- $\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}$

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$
- $\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}$

Simultaneous assignments

– shorten code

– save auxiliary variables (for example for swapping)

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$
- $\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}$

Simultaneous assignments

– shorten code

– save auxiliary variables (for example for swapping)

– make proofs easier

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$
- $\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}$

Simultaneous assignments

– shorten code

– save auxiliary variables (for example for swapping)

– make proofs easier

– **require simultaneous substitution**

Example Problems (with Simultaneous Assignments)

$$\{n \geq 0\} \quad (y, a, b) := (0, 1, 1) \,\textbf{;}$$
$$\textbf{while } y \neq n \textbf{ do } (y, a, b) := (y + 1, b, a + b) \textbf{ od} \quad \{a = fib_n\}$$

Given an n-element C-like array s, prove partial correctness:

$$\{\mathsf{True}\}$$
$$(i, a) := (0, 0) \,\textbf{;}$$
$$\textbf{while } i \neq n$$
$$\textbf{do} \quad \textbf{if } x = s[i]$$
$$\textbf{then } (i, a) := (i + 1, a + 1)$$
$$\textbf{fi od}$$
$$\{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < n\}\}$$

What does this program do?

Semantics and Language Design

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**
- Such rule sets are **specifications** of **language implementations**

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**
- Such rule sets are **specifications** of **language implementations**
- We **define the rules** for language features and extensions

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**
- Such rule sets are **specifications** of **language implementations**
- We **define the rules** for language features and extensions
- We **justify the rules** against different presentations of the defined features

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**
- Such rule sets are **specifications** of **language implementations**
- We **define the rules** for language features and extensions
- We **justify the rules** against different presentations of the defined features
- We **derive the rules** e.g. from source-to-source translations

Fibonacci

$\{n \geq 0\}$ $(y, a, b) := (0, 1, 1)$ **;**

while $y \neq n$ **do** $(y, a, b) := (y + 1, b, a + b)$ **od** $\{a = fib_n\}$

Fibonacci

$\{n \geq 0\}$ $(y, a, b) := (0, 1, 1)$ **;**

while $y \neq n$ **do** $(y, a, b) := (y + 1, b, a + b)$ **od** $\{a = fib_n\}$

$\Leftarrow \langle$ (right consequence) $\rangle$

$\{n \geq 0\}\ P\ \{a = fib_y \wedge b = fib_{y+1} \wedge y = n\}$

$\wedge\ (a = fib_y \wedge b = fib_{y+1} \wedge y = n \Rightarrow a = fib_n)$

Fibonacci

$$\{n \geq 0\} \quad (y, a, b) := (0, 1, 1)\ \mathbf{;}$$
$$\quad \textbf{while } y \neq n \textbf{ do } (y, a, b) := (y + 1, b, a + b) \textbf{ od} \quad \{a = fib_n\}$$

$\Leftarrow \langle$ (right consequence) $\rangle$

$$\{n \geq 0\}\ P\ \{a = fib_y \wedge b = fib_{y+1} \wedge y = n\}$$
$$\wedge (a = fib_y \wedge b = fib_{y+1} \wedge y = n \Rightarrow a = fib_n)$$

$\Leftarrow \langle$ (sequence , logic) $\rangle$

$$\{n \geq 0\}\ (y, a, b) := (0, 1, 1)\ \{a = fib_y \wedge b = fib_{y+1}\} \wedge$$
$$\{a = fib_y \wedge b = fib_{y+1}\}\textbf{while } y \neq n \textbf{ do } A \textbf{ od}\{a = fib_y \wedge b = fib_{y+1} \wedge y = n\}$$
$$\wedge \text{ True}$$

Fibonacci (ctd.)

$\Leftarrow \langle$ (left consequence , **while**) $\rangle$

$$(n \geq 0 \Rightarrow 1 = fib_0 \wedge 1 = fib_{0+1})$$
$$\wedge \{1 = fib_0 \wedge 1 = fib_{0+1}\}\ (y, a, b) := (0, 1, 1)\ \{a = fib_y \wedge b = fib_{y+1}\}$$
$$\wedge \{a = fib_y \wedge b = fib_{y+1} \wedge y \neq n\}\ (y, a, b) := (y + 1, b, a + b)$$
$$\{a = fib_y \wedge b = fib_{y+1}\}$$

Fibonacci (ctd.)

$\Leftarrow \langle$ (left consequence , **while**) $\rangle$

$(n \geq 0 \Rightarrow 1 = fib_0 \wedge 1 = fib_{0+1})$

$\wedge \{1 = fib_0 \wedge 1 = fib_{0+1}\}\ (y, a, b) := (0, 1, 1)\ \{a = fib_y \wedge b = fib_{y+1}\}$

$\wedge \{a = fib_y \wedge b = fib_{y+1} \wedge y \neq n\}\ (y, a, b) := (y + 1, b, a + b)$

$\{a = fib_y \wedge b = fib_{y+1}\}$

$\Leftarrow \langle$ (arithmetic , assignment , left consequence) $\rangle$

True $\wedge$ True

$\wedge (a = fib_y \wedge b = fib_{y+1} \wedge y \neq n \Rightarrow b = fib_{y+1} \wedge a + b = fib_{(y+1)+1})$

$\wedge \{b = fib_{y+1} \wedge a + b = fib_{(y+1)+1}\}\ (y, a, b) := (y + 1, b, a + b)$

$\{a = fib_y \wedge b = fib_{y+1}\}$

Fibonacci (ctd.)

$\Leftarrow \langle$ (left consequence , **while**) $\rangle$

$(n \geq 0 \Rightarrow 1 = fib_0 \wedge 1 = fib_{0+1})$

$\wedge \{1 = fib_0 \wedge 1 = fib_{0+1}\}\ (y, a, b) := (0, 1, 1)\ \{a = fib_y \wedge b = fib_{y+1}\}$

$\wedge \{a = fib_y \wedge b = fib_{y+1} \wedge y \neq n\}\ (y, a, b) := (y + 1, b, a + b)$

$\{a = fib_y \wedge b = fib_{y+1}\}$

$\Leftarrow \langle$ (arithmetic , assignment , left consequence) $\rangle$

True $\wedge$ True

$\wedge (a = fib_y \wedge b = fib_{y+1} \wedge y \neq n \Rightarrow b = fib_{y+1} \wedge a + b = fib_{(y+1)+1})$

$\wedge \{b = fib_{y+1} \wedge a + b = fib_{(y+1)+1}\}\ (y, a, b) := (y + 1, b, a + b)$

$\{a = fib_y \wedge b = fib_{y+1}\}$

$\Leftarrow \langle$ (arithmetic , assignment) $\rangle$

True $\wedge$ True

Array Traversal

Given an n-element C-like array s, prove partial correctness:

$\{\textsf{True}\}$

$(i, a) := (0, 0)$ **;**

while $i \neq n$

do **if** $x = s[i]$

then $(i, a) := (i + 1, a + 1)$

fi od

$\{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < n\}\}$

$\{\textsf{True}\}\ P\ \{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < n\}\}$

$\Leftarrow \langle$ (right consequence) $\rangle$

$\{\textsf{True}\}\ Init\ \mathbf{;}\ W\ \{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i\} \wedge i = n\}$

$\wedge ((a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i\} \wedge i = n) \Rightarrow Post)$

$\Leftarrow \langle$ (sequence , logic) $\rangle$

$$\{\mathsf{True}\}\ (i,a) := (0,0)\ \{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\}\ \}$$

$$\wedge\ \{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\}\ \}\ W$$

$$\{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\} \wedge i = n\}$$

$$\wedge\ \mathsf{True}$$

$$\Leftarrow \langle\ (\text{ left consequence },\ \ \textbf{while}\ )\ \rangle$$

$$(\mathsf{True} \Rightarrow 0 = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < 0\})$$

$$\wedge\ \{0 = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < 0\}\ \}\ (i,a) := (0,0)$$

$$\{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\}\ \}$$

$$\wedge\ \{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\} \wedge i \ne n\}\ B$$

$$\{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\}\ \}$$

$$\Leftarrow \langle\ (\text{ logic and arithmetic },\ \ \text{assignment },\ \ \text{conditional }) \ \rangle$$

$$\mathsf{True} \wedge \mathsf{True}$$

$$\wedge\ \{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\} \wedge i \ne n \wedge x = s[i]\}$$

$$(i,a) := (i+1, a+1)\ \{a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\}\ \}$$

$$\wedge\ ((a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\} \wedge i \ne n \wedge x \ne s[i]) \Rightarrow a = \#\{j : \mathbb{N} \ \mid s[j] = x \wedge 0 \le j < i\})$$

$\Leftarrow \langle$ (left consequence , logic) $\rangle$

$$((a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i\} \wedge i \neq n \wedge x = s[i])$$
$$\Rightarrow a + 1 = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i + 1)$$
$$\wedge \{a + 1 = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i + 1\}\} \; (i, a) := (i + 1, a + 1)$$
$$\{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < i\} \}$$

$\wedge$ True

$\Leftarrow \langle$ (logic , assignment) $\rangle$

True $\wedge$ True

Array Traversal

Given an n-element C-like array s, prove partial correctness:

$\{\mathsf{True}\}$
$(i, a) := (0, 0)$ **;**
while $i \neq n$
do **if** $x = s[i]$
then $(i, a) := (i + 1, a + 1)$
fi od
$\{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < n\}\}$

What does this program do?

Integer Square Root

$\{n \geq 0\}$

$(a, b) := (0, n + 1)$ **;**

while $a + 1 \neq b$

do $d := (a + b)/2$ **;**

if $d * d \leq n$

then $a := d$

else $b := d$

fi

od

$\{a^2 \leq n < (a + 1)^2\}$

All variables and expressions are of type **integer**.

Exercise 11.1(a, b)

$$\dfrac{\dfrac{\dfrac{\text{True}}{x \geq -5 \;\Rightarrow\; x \geq -6}\,(\text{arith.})}{x \geq -5 \;\Rightarrow\; 5 - x \leq 11}\,(\text{arith.}) \qquad \dfrac{\text{True}}{\{5 - x \leq 11\}\; z := 5 - x\; \{z \leq 11\}}\,(\text{assign.})}{\{x \geq -5\}\; z := 5 - x\; \{z \leq 11\}}\,(\text{conseq.})$$

Exercise 11.1(a, b)

$$\dfrac{\dfrac{\dfrac{\textsf{True}}{x \geq -5 \Rightarrow x \geq -6}(\text{arith.})}{x \geq -5 \Rightarrow 5 - x \leq 11}(\text{arith.}) \qquad \dfrac{\textsf{True}}{\{5 - x \leq 11\}\ z := 5 - x\ \{z \leq 11\}}(\text{assign.})}{\{x \geq -5\}\ z := 5 - x\ \{z \leq 11\}}(\text{conseq.})$$

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -7\}$$

$\Leftarrow$ ⟨ (left consequence) ⟩

$$(x \geq -5 \Rightarrow 5 - x \leq 11 \wedge x \geq -7)$$
$$\wedge \{5 - x \leq 11 \wedge x \geq -7\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -7\}$$

$\Leftarrow$ ⟨ (arithmetic , assignment) ⟩

$$(x \geq -5 \Rightarrow x \geq -6 \wedge x \geq -7) \wedge \textsf{True}$$

$\Leftarrow$ ⟨ (arithmetic) ⟩

True

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\{x \mapsto -5\}(z := 5 - x) \Rightarrow$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\frac{\{x \mapsto -5\}(5 - x) \Rightarrow}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\frac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow \qquad\qquad \{x \mapsto -5\}(x) \Rightarrow}{\{x \mapsto -5\}(5 - x) \Rightarrow}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\dfrac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow 5 \qquad \{x \mapsto -5\}(x) \Rightarrow}{\{x \mapsto -5\}(5 - x) \Rightarrow}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\dfrac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow 5 \qquad \{x \mapsto -5\}(x) \Rightarrow -5}{\{x \mapsto -5\}(5 - x) \Rightarrow}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\dfrac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow 5 \qquad \{x \mapsto -5\}(x) \Rightarrow -5}{\{x \mapsto -5\}(5 - x) \Rightarrow 10}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\dfrac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow 5 \qquad \{x \mapsto -5\}(x) \Rightarrow -5}{\{x \mapsto -5\}(5 - x) \Rightarrow 10}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow \{x \mapsto -5, z \mapsto 10\}}$$

Exercise 11.1(c)

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample:**

$$\dfrac{\dfrac{\{x \mapsto -5\}(5) \Rightarrow 5 \qquad \{x \mapsto -5\}(x) \Rightarrow -5}{\{x \mapsto -5\}(5 - x) \Rightarrow 10}}{\{x \mapsto -5\}(z := 5 - x) \Rightarrow \{x \mapsto -5, z \mapsto 10\}}$$

This last state clearly does not satisfy $\{z \leq 11 \wedge x \geq -3\}$

Exercise 11.1(d)

$$\{x \geq -5\}\ z := 5 - x\ \mathbf{;}\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$$

Exercise 11.1(d)

$$\{x \geq -5\}\ z := 5 - x\ \mathbf{;}\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$$

$\Leftarrow$ $\langle$ sequence rule $\rangle$

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11\ \wedge\ x + 2 \geq -3\}$$

$$\wedge\ \{z \leq 11\ \wedge\ x + 2 \geq -3\}\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$$

Exercise 11.1(d)

$$\{x \geq -5\}\ z := 5 - x\ ;\ x := x + 2\ \{z \leq 11 \land x \geq -3\}$$

$\Leftarrow$ $\langle$ sequence rule $\rangle$

$$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \ \land\ x + 2 \geq -3\}$$

$$\land\ \{z \leq 11 \ \land\ x + 2 \geq -3\}\ x := x + 2\ \{z \leq 11 \land x \geq -3\}$$

$\Leftarrow$ $\langle$ left consequence , assignment $\rangle$

$$(x \geq -5 \Rightarrow (5 - x \leq 11 \ \land\ x + 2 \geq -3))$$

$$\land\ \{5 - x \leq 11 \ \land\ x + 2 \geq -3\}\ z := 5 - x\ \{z \leq 11 \ \land\ x + 2 \geq -3\}$$

$$\land\ \mathsf{True}$$

Exercise 11.1(d)

$\{x \geq -5\}\ z := 5 - x\ ;\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$

$\Leftarrow$ ⟨ sequence rule ⟩

$\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x + 2 \geq -3\}$

$\wedge\ \{z \leq 11 \wedge x + 2 \geq -3\}\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$

$\Leftarrow$ ⟨ left consequence , assignment ⟩

$(x \geq -5 \Rightarrow (5 - x \leq 11 \wedge x + 2 \geq -3))$

$\wedge\ \{5 - x \leq 11 \wedge x + 2 \geq -3\}\ z := 5 - x\ \{z \leq 11 \wedge x + 2 \geq -3\}$

$\wedge$ True

$\Leftarrow$ ⟨ logic and arithmetic , assignment ⟩

$(x \geq -5 \Rightarrow x \geq -6) \wedge (x \geq -5 \Rightarrow x \geq -5) \wedge$ True

Exercise 11.1(d)

$\{x \geq -5\}\ z := 5 - x\ ;\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$

$\Leftarrow \quad \langle$ sequence rule $\rangle$

$\quad \{x \geq -5\}\ z := 5 - x\ \{z \leq 11\ \wedge\ x + 2 \geq -3\}$

$\quad \wedge \{z \leq 11\ \wedge\ x + 2 \geq -3\}\ x := x + 2\ \{z \leq 11 \wedge x \geq -3\}$

$\Leftarrow \quad \langle$ left consequence , assignment $\rangle$

$\quad (x \geq -5 \Rightarrow (5 - x \leq 11\ \wedge\ x + 2 \geq -3))$

$\quad \wedge \{5 - x \leq 11\ \wedge\ x + 2 \geq -3\}\ z := 5 - x\ \{z \leq 11\ \wedge\ x + 2 \geq -3\}$

$\quad \wedge$ True

$\Leftarrow \quad \langle$ logic and arithmetic , assignment $\rangle$

$\quad (x \geq -5 \Rightarrow x \geq -6) \wedge (x \geq -5 \Rightarrow x \geq -5) \wedge$ True

$\Leftarrow \quad \langle$ arithmetic $\rangle$

$\quad$ True

Exercise 11.1(e)

$$\{x \geq -5\}\ z := 5 - x\ ;\ x := x + z\ \{z \leq 11 \wedge x = 2\}$$

Exercise 11.1(e)

$$\{x \geq -5\}\ z := 5 - x\ ;\ x := x + z\ \{z \leq 11 \wedge x = 2\}$$

We again use operational semantics (expression evaluation not shown) to prove a **counterexample:**

$$\sigma_1 = \{x \mapsto 0\}$$
$$\sigma_2 = \{x \mapsto 0, z \mapsto 5\}$$
$$\sigma_3 = \{x \mapsto 5, z \mapsto 5\}$$

$$\dfrac{\dfrac{\sigma_1(5 - x) \Rightarrow 5}{\sigma_1(z := 5 - x) \Rightarrow \sigma_2} \qquad \dfrac{\sigma_2(x + z) \Rightarrow 5}{\sigma_2(x := x + z) \Rightarrow \sigma_3}}{\sigma_1(z := 5 - x\ ;\ x := x + z) \Rightarrow \sigma_3}$$

Although $\sigma_1 = \{x \mapsto 0\}$ satisfies the precondition $\{x \geq -5\}$, the final state $\sigma_3 = \{x \mapsto 5, z \mapsto 5\}$ does not satisfy the postcondition $\{z \leq 11 \wedge x = 2\}$.

Exercise 11.1(f)

$$\{z = abs(x)\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ z := -z\ \textbf{fi}\ \{xz = -x^2\}$$

Exercise 11.1(f)

one-sided conditional:

$\{z = abs(x)\}$ **if** $x \geq 0$ **then** $z := -z$ **fi** $\{xz = -x^2\}$

Exercise 11.1(f)

Rule for one-sided conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad P \wedge \neg b \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{fi}\ \{Q\}}$$

$\{z = abs(x)\}$ **if** $x \geq 0$ **then** $z := -z$ **fi** $\{xz = -x^2\}$

Exercise 11.1(f)

Rule for one-sided conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad P \wedge \neg b \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{fi}\ \{Q\}}$$

$$\{z = abs(x)\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ z := -z\ \textbf{fi}\ \{xz = -x^2\}$$

$$\Leftarrow \quad \langle \text{ one-sided conditional } \rangle$$

$$\{z = abs(x) \wedge x \geq 0\}\ z := -z\ \{xz = -x^2\}$$

$$\wedge\ (z = abs(x) \wedge x < 0 \Rightarrow xz = -x^2)$$

Exercise 11.1(f)

Rule for one-sided conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad P \wedge \neg b \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{fi}\ \{Q\}}$$

$$\{z = abs(x)\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ z := -z\ \textbf{fi}\ \{xz = -x^2\}$$

$$\begin{aligned}
\Leftarrow \quad & \langle \text{ one-sided conditional } \rangle \\
& \{z = abs(x) \wedge x \geq 0\}\ z := -z\ \{xz = -x^2\} \\
& \wedge (z = abs(x) \wedge x < 0 \Rightarrow xz = -x^2) \\
\Leftarrow \quad & \langle \text{ left consequence , arithmetic } \rangle \\
& (z = abs(x) \wedge x \geq 0 \Rightarrow x \cdot (-z) = -x^2) \\
& \wedge \{x \cdot (-z) = -x^2\}\ z := -z\ \{xz = -x^2\} \\
& \wedge (z = -x \Rightarrow xz = -x^2)
\end{aligned}$$

Exercise 11.1(f)

Rule for one-sided conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad P \wedge \neg b \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{fi}\ \{Q\}}$$

$$\{z = abs(x)\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ z := -z\ \textbf{fi}\ \{xz = -x^2\}$$

$$\Leftarrow \quad \langle \text{ one-sided conditional } \rangle$$

$$\{z = abs(x) \wedge x \geq 0\}\ z := -z\ \{xz = -x^2\}$$

$$\wedge\ (z = abs(x) \wedge x < 0 \Rightarrow xz = -x^2)$$

$$\Leftarrow \quad \langle \text{ left consequence , arithmetic } \rangle$$

$$(z = abs(x) \wedge x \geq 0 \Rightarrow x \cdot (-z) = -x^2)$$

$$\wedge\ \{x \cdot (-z) = -x^2\}\ z := -z\ \{xz = -x^2\}$$

$$\wedge\ (z = -x \Rightarrow xz = -x^2)$$

$$\Leftarrow \quad \langle \text{ arithmetic , assignment , arithmetic } \rangle$$

$$(z = x \Rightarrow x \cdot (-z) = -x^2) \wedge \mathsf{True} \wedge \mathsf{True}$$

Exercise 11.1(f)

Rule for one-sided conditional:

$$\frac{\{P \wedge b\}\ S_1\ \{Q\} \qquad P \wedge \neg b \Rightarrow Q}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{fi}\ \{Q\}}$$

$\{z = abs(x)\}$ **if** $x \geq 0$ **then** $z := -z$ **fi** $\{xz = -x^2\}$

$\Leftarrow \quad \langle$ one-sided conditional $\rangle$

$\quad \{z = abs(x) \wedge x \geq 0\}\ z := -z\ \{xz = -x^2\}$

$\quad \wedge (z = abs(x) \wedge x < 0 \Rightarrow xz = -x^2)$

$\Leftarrow \quad \langle$ left consequence , arithmetic $\rangle$

$\quad (z = abs(x) \wedge x \geq 0 \Rightarrow x \cdot (-z) = -x^2)$

$\quad \wedge \{x \cdot (-z) = -x^2\}\ z := -z\ \{xz = -x^2\}$

$\quad \wedge (z = -x \Rightarrow xz = -x^2)$

$\Leftarrow \quad \langle$ arithmetic , assignment , arithmetic $\rangle$

$\quad (z = x \Rightarrow x \cdot (-z) = -x^2) \wedge \mathsf{True} \wedge \mathsf{True}$

$\Leftarrow \quad \langle$ arithmetic $\rangle$

$\quad$ True

Exercise 11.1(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w := \mathsf{True}$ **else** $z := 1/x$ **fi** $\{\neg w \rightarrow xz = 1\}$

Exercise 11.1(g)

$$\{z = 0\}\ \textbf{if}\ x = 0\ \textbf{then}\ w := \mathsf{True}\ \textbf{else}\ z := 1/x\ \textbf{fi}\ \{\neg w \rightarrow xz = 1\}$$

$$\Leftarrow\ \langle\ \text{conditional}\ \rangle$$

$$\{z = 0 \wedge x = 0\}\ w := \mathsf{True}\ \{\neg w \rightarrow xz = 1\}$$

$$\wedge\ \{z = 0 \wedge x \neq 0\}\ z := 1/x\ \{\neg w \rightarrow xz = 1\}$$

Exercise 11.1(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w :=$ True **else** $z := 1/x$ **fi** $\{\neg w \to xz = 1\}$

$\Leftarrow$ ⟨ conditional ⟩

$\{z = 0 \wedge x = 0\}\ w :=$ True $\{\neg w \to xz = 1\}$

$\wedge \{z = 0 \wedge x \neq 0\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow$ ⟨ left consequence , left consequence ⟩

$(z = 0 \wedge x = 0 \Rightarrow (\neg\text{True} \to xz = 1))$

$\wedge \{\neg\text{True} \to xz = 1\}\ w := \text{True}\ \{\neg w \to xz = 1\}$

$\wedge (z = 0 \wedge x \neq 0 \Rightarrow (\neg w \to x \cdot (1/x) = 1))$

$\wedge \{\neg w \to x \cdot (1/x) = 1\}\ z := 1/x\ \{\neg w \to xz = 1\}$

Exercise 11.1(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w := \mathsf{True}$ **else** $z := 1/x$ **fi** $\{\neg w \to xz = 1\}$

$\Leftarrow$ ⟨ conditional ⟩

$\{z = 0 \wedge x = 0\}\ w := \mathsf{True}\ \{\neg w \to xz = 1\}$

$\wedge\ \{z = 0 \wedge x \neq 0\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow$ ⟨ left consequence , left consequence ⟩

$(z = 0 \wedge x = 0 \Rightarrow (\neg\mathsf{True} \to xz = 1))$

$\wedge\ \{\neg\mathsf{True} \to xz = 1\}\ w := \mathsf{True}\ \{\neg w \to xz = 1\}$

$\wedge\ (z = 0 \wedge x \neq 0 \Rightarrow (\neg w \to x \cdot (1/x) = 1))$

$\wedge\ \{\neg w \to x \cdot (1/x) = 1\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow$ ⟨ logic , assignment , logic , assignment ⟩

$(z = 0 \wedge x = 0 \Rightarrow (\mathsf{False} \to xz = 1)) \wedge \mathsf{True} \wedge (x \neq 0 \Rightarrow x \cdot (1/x) = 1)$

Exercise 11.1(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w := \mathsf{True}$ **else** $z := 1/x$ **fi** $\{\neg w \to xz = 1\}$

$\Leftarrow \langle$ conditional $\rangle$

$\{z = 0 \wedge x = 0\}\ w := \mathsf{True}\ \{\neg w \to xz = 1\}$

$\wedge\ \{z = 0 \wedge x \neq 0\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow \langle$ left consequence , left consequence $\rangle$

$(z = 0 \wedge x = 0 \Rightarrow (\neg \mathsf{True} \to xz = 1))$

$\wedge\ \{\neg \mathsf{True} \to xz = 1\}\ w := \mathsf{True}\ \{\neg w \to xz = 1\}$

$\wedge\ (z = 0 \wedge x \neq 0 \Rightarrow (\neg w \to x \cdot (1/x) = 1))$

$\wedge\ \{\neg w \to x \cdot (1/x) = 1\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow \langle$ logic , assignment , logic , assignment $\rangle$

$(z = 0 \wedge x = 0 \Rightarrow (\mathsf{False} \to xz = 1)) \wedge \mathsf{True} \wedge (x \neq 0 \Rightarrow x \cdot (1/x) = 1)$

$\Leftarrow \langle$ *ex falso quodlibet* , arithmetic $\rangle$

$(z = 0 \wedge x = 0 \Rightarrow \mathsf{True}) \wedge \mathsf{True}$

Exercise 11.1(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w :=$ True **else** $z := 1/x$ **fi** $\{\neg w \to xz = 1\}$

$\Leftarrow \langle$ conditional $\rangle$

$\{z = 0 \land x = 0\}\ w :=$ True $\{\neg w \to xz = 1\}$

$\land\ \{z = 0 \land x \neq 0\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow \langle$ left consequence , left consequence $\rangle$

$(z = 0 \land x = 0 \Rightarrow (\neg \text{True} \to xz = 1))$

$\land\ \{\neg \text{True} \to xz = 1\}\ w :=$ True $\{\neg w \to xz = 1\}$

$\land\ (z = 0 \land x \neq 0 \Rightarrow (\neg w \to x \cdot (1/x) = 1))$

$\land\ \{\neg w \to x \cdot (1/x) = 1\}\ z := 1/x\ \{\neg w \to xz = 1\}$

$\Leftarrow \langle$ logic , assignment , logic , assignment $\rangle$

$(z = 0 \land x = 0 \Rightarrow (\text{False} \to xz = 1)) \land \text{True} \land (x \neq 0 \Rightarrow x \cdot (1/x) = 1)$

$\Leftarrow \langle$ *ex falso quodlibet* , arithmetic $\rangle$

$(z = 0 \land x = 0 \Rightarrow \text{True}) \land \text{True}$

$\Leftarrow \langle$ logic $\rangle$

True

repeat … until …

Operational Semantics:

repeat … **until** …

Operational Semantics:

$$\frac{\sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(b) \Rightarrow \mathsf{True}}{\sigma(\textbf{repeat } s \textbf{ until } b) \Rightarrow \sigma_1}$$

repeat ... **until** ...

Operational Semantics:

$$\frac{\sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(b) \Rightarrow \textsf{True}}{\sigma(\textbf{repeat } s \textbf{ until } b) \Rightarrow \sigma_1}$$

$$\frac{\sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(b) \Rightarrow \textsf{False} \qquad \sigma_1(\textbf{repeat } s \textbf{ until } b) \Rightarrow \sigma_2}{\sigma(\textbf{repeat } s \textbf{ until } b) \Rightarrow \sigma_2}$$

repeat … **until** …

Operational Semantics:

$$\frac{\sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(b) \Rightarrow \mathsf{True}}{\sigma(\textbf{repeat}\ s\ \textbf{until}\ b) \Rightarrow \sigma_1}$$

$$\frac{\sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(b) \Rightarrow \mathsf{False} \qquad \sigma_1(\textbf{repeat}\ s\ \textbf{until}\ b) \Rightarrow \sigma_2}{\sigma(\textbf{repeat}\ s\ \textbf{until}\ b) \Rightarrow \sigma_2}$$

Axiomatic Semantics:

$$\frac{\{INV\}\ S\ \{INV\}}{\{INV\}\ \textbf{repeat}\ s\ \textbf{until}\ b\ \{INV \wedge b\}}$$

Proper for-Loops — Operational Semantics

Proper **for**-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset

Proper for-Loops — Operational Semantics

***Proper* for**-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset

Operational Semantics:

$$\frac{\sigma(beg) \Rightarrow b \qquad \sigma(end) \Rightarrow e \qquad \sigma_b = \sigma \qquad \begin{array}{l} \forall i : \mathbb{N} \mid b \le i \le e \bullet \\ (\sigma_i \oplus \{v \mapsto i\})(s) \Rightarrow \sigma_{i+1} \end{array}}{\sigma(\textbf{for } v := beg \textbf{ to } end \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_{\max(b,e+1)}}$$

Proper for-Loops — Operational Semantics

***Proper* for**-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset

Operational Semantics:

$$\frac{\sigma(beg) \Rightarrow b \qquad \sigma(end) \Rightarrow e \qquad \sigma_b = \sigma \qquad \begin{array}{c}\forall i : \mathbb{N} \mid b \leq i \leq e \bullet \\ (\sigma_i \oplus \{v \mapsto i\})(s) \Rightarrow \sigma_{i+1}\end{array}}{\sigma(\textbf{for } v := beg \textbf{ to } end \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_{\max(b,e+1)}}$$

- This resets the loop variable at the beginning of each iteration
- A static test can prevent assignments to the loop variable

Exercise 11.2

$\{\text{True}\}$

$(i, j, s) := (0, 0, 0)$ **;**

while $i \neq n$ **do**

  if $i = j$

  then $(i, j, s) := (i + 1, 0, s + 1)$

  else $(j, s) := (j + 1, s + 2)$

  fi

 od

$\{s = n^2 + 2j\}$

Exercise 11.2 — Proof

$\{\mathsf{True}\}\ (i, j, s) := (0, 0, 0)$ **; while** $i \neq n$ **do** B **od** $\{s = n^2 + 2j\}$

Exercise 11.2 — Proof

$\{\mathsf{True}\}\ (i, j, s) := (0, 0, 0)$ **; while** $i \neq n$ **do** B **od** $\{s = n^2 + 2j\}$

$\Leftarrow$ ⟨ right consequence ⟩

$\{\mathsf{True}\}\ (i, j, s) := (0, 0, 0)$ **; while** $i \neq n$ **do** B **od** $\{s = i^2 + 2j \wedge i = n\}$

$\wedge\ (s = i^2 + 2j \wedge i = n \Rightarrow s = n^2 + 2j)$

Exercise 11.2 — Proof

$\{\textsf{True}\}\ (i, j, s) := (0, 0, 0)\ \textbf{; while}\ i \neq n\ \textbf{do}\ B\ \textbf{od}\ \{s = n^2 + 2j\}$

$\Leftarrow\quad \langle$ right consequence $\rangle$

$\{\textsf{True}\}\ (i, j, s) := (0, 0, 0)\ \textbf{; while}\ i \neq n\ \textbf{do}\ B\ \textbf{od}\ \{s = i^2 + 2j \wedge i = n\}$

$\wedge\ (s = i^2 + 2j \wedge i = n \Rightarrow s = n^2 + 2j)$

$\Leftarrow\quad \langle$ sequence , logic $\rangle$

$\{\textsf{True}\}\ (i, j, s) := (0, 0, 0)\ \{s = i^2 + 2j\}$

$\wedge\ \{s = i^2 + 2j\}\ \textbf{while}\ i \neq n\ \textbf{do}\ B\ \textbf{od}\ \{s = i^2 + 2j \wedge i = n\}$

$\wedge\ \textsf{True}$

Exercise 11.2 — Proof

$\{\mathsf{True}\}\ (i,j,s) := (0,0,0)\ \mathbf{;\ while}\ i \neq n\ \mathbf{do}\ B\ \mathbf{od}\ \{s = n^2 + 2j\}$

$\Leftarrow\ \langle$ right consequence $\rangle$

$\{\mathsf{True}\}\ (i,j,s) := (0,0,0)\ \mathbf{;\ while}\ i \neq n\ \mathbf{do}\ B\ \mathbf{od}\ \{s = i^2 + 2j \wedge i = n\}$

$\wedge\ (s = i^2 + 2j \wedge i = n \Rightarrow s = n^2 + 2j)$

$\Leftarrow\ \langle$ sequence , logic $\rangle$

$\{\mathsf{True}\}\ (i,j,s) := (0,0,0)\ \{s = i^2 + 2j\}$

$\wedge\ \{s = i^2 + 2j\}\ \mathbf{while}\ i \neq n\ \mathbf{do}\ B\ \mathbf{od}\ \{s = i^2 + 2j \wedge i = n\}$

$\wedge\ \mathsf{True}$

$\Leftarrow\ \langle$ left consequence , **while** -rule $\rangle$

$(\mathsf{True} \Rightarrow 0 = 0^2 + 2 \cdot 0)$

$\wedge\ \{0 = 0^2 + 2 \cdot 0\}\ (i,j,s) := (0,0,0)\ \{s = i^2 + 2j\}$

$\wedge\ \{s = i^2 + 2j \wedge i \neq n\}\ \mathbf{if}\ i = j\ \mathbf{then}\ (i,j,s) := (i+1, 0, s+1)$

$\mathbf{else}\ (j,s) := (j+1, s+2)\ \mathbf{fi}\ \{s = i^2 + 2j\}$

Exercise 11.2 — Proof (ctd.)

$\Leftarrow$ $\langle$ arithmetic , assignment , conditional $\rangle$

$\mathsf{True} \wedge \mathsf{True}$

$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i = j\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\}$

$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i \neq j\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}$

Exercise 11.2 — Proof (ctd.)

$\Leftarrow \quad \langle$ arithmetic , assignment , conditional $\rangle$

$\mathsf{True} \wedge \mathsf{True}$

$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i = j\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\}$

$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i \neq j\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}$

Exercise 11.2 — Proof (ctd.)

$\Leftarrow$ $\langle$ arithmetic , assignment , conditional $\rangle$

$$\begin{aligned}
&\mathsf{True} \wedge \mathsf{True} \\
&\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i = j\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\} \\
&\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i \neq j\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}
\end{aligned}$$

$\Leftarrow$ $\langle$ left consequence , left consequence $\rangle$

$$\begin{aligned}
&(s = i^2 + 2j \wedge i \neq n \wedge i = j \Rightarrow s + 1 = (i + 1)^2 + 2 \cdot 0) \\
&\wedge \{s + 1 = (i + 1)^2 + 2 \cdot 0\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\} \\
&\wedge (s = i^2 + 2j \wedge i \neq n \wedge i \neq j \Rightarrow s + 2 = i^2 + 2 \cdot (j + 1)) \\
&\wedge \{s + 2 = i^2 + 2 \cdot (j + 1)\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}
\end{aligned}$$

Exercise 11.2 — Proof (ctd.)

$\Leftarrow$ $\langle$ arithmetic , assignment , conditional $\rangle$

$$\mathsf{True} \wedge \mathsf{True}$$
$$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i = j\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\}$$
$$\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i \neq j\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}$$

$\Leftarrow$ $\langle$ left consequence , left consequence $\rangle$

$$(s = i^2 + 2j \wedge i \neq n \wedge i = j \Rightarrow s + 1 = (i + 1)^2 + 2 \cdot 0)$$
$$\wedge \{s + 1 = (i + 1)^2 + 2 \cdot 0\}\ (i, j, s) := (i + 1, 0, s + 1)\ \{s = i^2 + 2j\}$$
$$\wedge (s = i^2 + 2j \wedge i \neq n \wedge i \neq j \Rightarrow s + 2 = i^2 + 2 \cdot (j + 1))$$
$$\wedge \{s + 2 = i^2 + 2 \cdot (j + 1)\}\ (j, s) := (j + 1, s + 2)\ \{s = i^2 + 2j\}$$

$\Leftarrow$ $\langle$ arithmetic , assignment , arithmetic , assignment $\rangle$

$$\mathsf{True} \wedge \mathsf{True} \wedge \mathsf{True} \wedge \mathsf{True}$$

Exercise 11.2 — Stronger Postcondition

$\{\mathsf{True}\}$

$(i, j, s) := (0, 0, 0)$ **;**

while $i \neq n$ **do**

  if $i = j$

  then $(i, j, s) := (i + 1, 0, s + 1)$

  else $(j, s) := (j + 1, s + 2)$

  fi

od

$\{s = n^2\}$

Exercise 11.2 — Stronger Postcondition

$\{\mathsf{True}\}$

$(i, j, s) := (0, 0, 0)$ **;**

while $i \neq n$ **do**

  if $i = j$

  then $(i, j, s) := (i + 1, 0, s + 1)$

  else $(j, s) := (j + 1, s + 2)$

  fi

od

$\{s = n^2\}$

Challenge: How can you prove this?

Exercise 11.2 — Stronger Postcondition

$\{\mathsf{True}\}$

$(i, j, s) := (0, 0, 0)$ **;**

while $i \neq n$ **do**

  if $i = j$

  then $(i, j, s) := (i + 1, 0, s + 1)$

  else $(j, s) := (j + 1, s + 2)$

  fi

od

$\{s = n^2\}$

Challenge: How can you prove this?
Can you reformulate the program to make this easier?

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

$$\mathit{State}_1 \; (\; \mathit{Statement} \;) \;\Rightarrow\; \mathit{State}_2$$

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  $$\mathit{State}_1 \ (\ \mathit{Statement}\ ) \ \Rightarrow\ \mathit{State}_2$$

  For some combinations of *Statement* and State_1, no such State_2 exists …

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  $$State_1\ (\ Statement\ )\ \Rightarrow\ State_2$$

  For some combinations of *Statement* and $State_1$, no such $State_2$ exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  $$\mathit{State}_1 \; (\mathit{Statement}) \;\Rightarrow\; \mathit{State}_2$$

  For some combinations of *Statement* and State_1, no such State_2 exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:

  $$\{ \mathit{Precondition} \} \; \mathit{Statement} \; \{ \mathit{Postcondition} \}$$

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  $$State_1 \; (\; Statement \;) \;\; \Rightarrow \;\; State_2$$

  For some combinations of *Statement* and $State_1$, no such $State_2$ exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:

  $$\{ \; Precondition \; \} \; Statement \; \{ \; Postcondition \; \}$$

  Proofs operate on **conditions on states** instead of states: **abstracting** away from states

Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  $$State_1 \ (\ Statement\ ) \ \Rightarrow\ State_2$$

  For some combinations of *Statement* and $State_1$, no such $State_2$ exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:

  $$\{\ Precondition\ \}\ Statement\ \{\ Postcondition\ \}$$

  Proofs operate on **conditions on states** instead of states: **abstracting** away from states

  For different properties, different proofs ***along the same program structure.***

Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

$$[\![\mathit{Statement}]\!] = F$$

  for some function or relation F.

Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

  $$[\![\mathit{Statement}]\!] = F$$

  for some function or relation F.

  F typically is a restricted kind of function between **semantic domains**.

Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

$$[\![\mathit{Statement}]\!] = F$$

  for some function or relation F.

  F typically is a restricted kind of function between **semantic domains**.

  Since F is a **single mathematical object**, it may be used as starting point for showing ***any kind of (functional) program properties***.

Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

  $$[\![Statement]\!] = F$$

  for some function or relation F.

  F typically is a restricted kind of function between **semantic domains**.

  Since F is a **single mathematical object**, it may be used as starting point for showing ***any kind of (functional) program properties***.

  In the textbook, denotational semantics appears mostly as a reorganisation of operational semantics.

  In general, the denotational semantics is **far more abstract** than operational semantics, and employs advanced concepts from discrete mathematics.

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a **definedness ordering** $\sqsubseteq$

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a **definedness ordering** $\sqsubseteq$, and

– a **least element** $\perp$ (read: "**bottom**") wrt. $\sqsubseteq$

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a **definedness ordering** $\sqsubseteq$, and

– a **least element** $\bot$ (read: "**bottom**") wrt. $\sqsubseteq$:

$$\forall x : D \bullet \bot \sqsubseteq x$$

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a **definedness ordering** $\sqsubseteq$, and

– a **least element** $\bot$ (read: "**bottom**") wrt. $\sqsubseteq$:

$$\forall x : D \bullet \bot \sqsubseteq x$$

– (least upper bounds of chains $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$)

Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a **definedness ordering** $\sqsubseteq$, and

– a **least element** $\bot$ (read: “**bottom**”) wrt. $\sqsubseteq$:

$$\forall x : D \bullet \bot \sqsubseteq x$$

– (least upper bounds of chains $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$)

For **simple** semantics of imperative programs, **sets of partial functions** $A \nrightarrow B$ can be used as domains:

– the subset ordering $\subseteq$ serves as definedness ordering:

$$\forall f, g : A \nrightarrow B \bullet f \sqsubseteq g :\Leftrightarrow f \subseteq g$$

– the empty function $\emptyset : A \nrightarrow B$ is the a least element of $A \nrightarrow B$.

Semantic Domains for Simple Imperative Programs

$Bool$	$= \{\text{True}, \text{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers

Semantic Domains for Simple Imperative Programs

$Bool$	$= \{\text{True}, \text{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
$SVal$	$= Bool + Num$	storable values

Semantic Domains for Simple Imperative Programs

$Bool$	$= \{\text{True}, \text{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
$SVal$	$= Bool + Num$	storable values
Id		identifiers

Semantic Domains for Simple Imperative Programs

Bool	$= \{\mathrm{True}, \mathrm{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Id		identifiers
State	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores

Semantic Domains for Simple Imperative Programs

$Bool$	$= \{\text{True}, \text{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
$SVal$	$= Bool + Num$	storable values
Id		identifiers
$State$	$= Id \nrightarrow SVal$	(simple) stores
Val	$= SVal$	values

Semantic Domains for Simple Imperative Programs

Bool	$= \{\text{True}, \text{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Id		identifiers
State	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores
Val	$= \mathit{SVal}$	values
$\mathit{State} \nrightarrow \mathit{Val}$		(expression semantics)

Semantic Domains for Simple Imperative Programs

Bool	$= \{\mathrm{True}, \mathrm{False}\}$	booleans
Num	$= \mathbb{Z}$	numbers
SVal	$= \mathit{Bool} + \mathit{Num}$	storable values
Id		identifiers
State	$= \mathit{Id} \nrightarrow \mathit{SVal}$	(simple) stores
Val	$= \mathit{SVal}$	values
$\mathit{State} \nrightarrow \mathit{Val}$		(expression semantics)
$\mathit{State} \nrightarrow \mathit{State}$		(statement semantics)

Direct Sums, or Disjoint Unions

"*A* + *B*" is the **direct sum** of the two sets *A* and *B*.

Direct Sums, or Disjoint Unions

"$A + B$" is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

Direct Sums, or Disjoint Unions

"$A + B$" is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

In ***Haskell,*** there is the following prelude type constructor:

data *Either a b* = *Left a* | *Right b*

Direct Sums, or Disjoint Unions

"$A + B$" is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

In ***Haskell,*** there is the following prelude type constructor:

data *Either a b = Left a | Right b*

This produces the two **constructors** for *Either* (which are **injections):**

$Left \;::\; a \rightarrow Either\ a\ b$
$Right \;::\; b \rightarrow Either\ a\ b$

Direct Sums, or Disjoint Unions

"$A + B$" is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

In ***Haskell,*** there is the following prelude type constructor:

data *Either a b* = *Left a* | *Right b*

This produces the two **constructors** for *Either* (which are **injections):**

$$\begin{aligned}&\mathit{Left} :: a \to \mathit{Either}\ a\ b\\ &\mathit{Right} :: b \to \mathit{Either}\ a\ b\end{aligned}$$

and allows pattern matching:

```
valShow :: Either Integer Bool → String
valShow (Left  i) = "int: "  ++ show i
valShow (Right b) = "bool: " ++ show b
```

Direct Sums, or Disjoint Unions

"$A + B$" is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

In ***Haskell,*** there is the following prelude type constructor:

data *Either a b* = *Left a* | *Right b*

This produces the two **constructors** for *Either* (which are **injections):**

Left :: *a* → *Either a b*
Right :: *b* → *Either a b*

and allows pattern matching:

valShow :: *Either Integer Bool* → *String*
valShow (*Left i*) = "int: " ++ *show i*
valShow (*Right b*) = "bool: " ++ *show b*

In mathematical use, *Left* and *Right* are frequently not mentioned.

Semantic Functions

$[\![\_]\!]_E$	$: \; \mathit{Expr} \to (\mathit{State} \nrightarrow \mathit{Val})$	expression semantics
$[\![\_]\!]_S$	$: \; \mathit{Stmt} \to (\mathit{State} \nrightarrow \mathit{State})$	statement semantics

Semantic Functions

$[\![\_]\!]_E$: $Expr \rightarrow (State \nrightarrow Val)$ expression semantics

$[\![\_]\!]_S$: $Stmt \rightarrow (State \nrightarrow State)$ statement semantics

$[\![\_]\!]_O$: $Op \rightarrow ((Val \times Val) \nrightarrow Val)$ operator semantics (given)

Semantic Functions

$[\![\_]\!]_E$	:	$\mathit{Expr} \to (\mathit{State} \nrightarrow \mathit{Val})$	expression semantics
$[\![\_]\!]_S$	:	$\mathit{Stmt} \to (\mathit{State} \nrightarrow \mathit{State})$	statement semantics
$[\![\_]\!]_O$	:	$\mathit{Op} \to ((\mathit{Val} \times \mathit{Val}) \nrightarrow \mathit{Val})$	operator semantics (given)

Textbook:

M	:	$(\mathit{Expr} \times \mathit{State}) \nrightarrow \mathit{Val}$	expression semantics
M	:	$(\mathit{Stmt} \times \mathit{State}) \nrightarrow \mathit{State}$	statement semantics
$\mathit{ApplyBinary}$	:	$(\mathit{Op} \times \mathit{Val} \times \mathit{Val}) \nrightarrow \mathit{Val}$	operator semantics

Semantic Functions

$[\![\_]\!]_E$	:	$Expr \to (State \nrightarrow Val)$	expression semantics
$[\![\_]\!]_S$	:	$Stmt \to (State \nrightarrow State)$	statement semantics
$[\![\_]\!]_O$	:	$Op \to ((Val \times Val) \nrightarrow Val)$	operator semantics (given)

Textbook:

M	:	$(Expr \times State) \nrightarrow Val$	expression semantics
M	:	$(Stmt \times State) \nrightarrow State$	statement semantics
$ApplyBinary$	:	$(Op \times Val \times Val) \nrightarrow Val$	operator semantics

- No clean separation between syntax and semantics
- Undefinedness ordering less obvious

Expression Semantics

$$Expr ::= Id \mid Num \mid Bool \mid Expr\ Op\ Expr$$

$$[\![\_]\!]_E \quad : \ Expr \to (State \nrightarrow Val)$$

Assuming $s : State$, i.e., $s : Id \nrightarrow SVal$, we define:

for $v : Id$: $\quad [\![v]\!]_E\ (s) = s(v)$

— **undefined** if $s(v)$ is undefined!

for $n : Num$: $\quad [\![n]\!]_E\ (s) = n$

for $b : Bool$: $\quad [\![b]\!]_E\ (s) = b$

for $e_1, e_2 : Expr; op : Op$: $\quad [\![e_1\ op\ e_2]\!]_E\ (s) = [\![op]\!]_O([\![e_1]\!]_E\ (s), [\![e_2]\!]_E\ (s))$

— **undefined** if $[\![e_1]\!]_E\ (s)$ or $[\![e_2]\!]_E\ (s)$ undefined, or from $[\![op]\!]_O$!

Expression Semantics

$$Expr ::= Id \mid Num \mid Bool \mid Expr\ Op\ Expr$$

$$[\![\_]\!]_{\mathrm{E}} \quad : \ Expr \to (State \mathrel{+\!\!\!\rightarrow} Val)$$

Assuming $s : State$, i.e., $s : Id \mathrel{+\!\!\!\rightarrow} SVal$, we define:

for $v : Id$: $[\![v]\!]_{\mathrm{E}}\ (s) = s(v)$

— **undefined** if $s(v)$ is undefined!

for $n : Num$: $[\![n]\!]_{\mathrm{E}}\ (s) = n$

for $b : Bool$: $[\![b]\!]_{\mathrm{E}}\ (s) = b$

for $e_1, e_2 : Expr; op : Op$: $[\![e_1\ op\ e_2]\!]_{\mathrm{E}}\ (s) = [\![op]\!]_{\mathrm{O}}([\![e_1]\!]_{\mathrm{E}}\ (s), [\![e_2]\!]_{\mathrm{E}}\ (s))$

— **undefined** if $[\![e_1]\!]_{\mathrm{E}}\ (s)$ or $[\![e_2]\!]_{\mathrm{E}}\ (s)$ undefined, or from $[\![op]\!]_{\mathrm{O}}$!

Where clear from the context, we write $[\![e]\!]$ instead of $[\![e]\!]_{\mathrm{E}}$.

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$$[\![x + y]\!](s_1) \quad = [\![x]\!](s_1) + [\![y]\!](s_1) \quad = s_1(x) + s_1(y) \quad = 5 + 42 \quad = 47$$

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$$[\![x + y]\!](s_1) = [\![x]\!](s_1) + [\![y]\!](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$[\![7 - q]\!](s_1) = [\![7]\!](s_1) - [\![q]\!](s_1) = 7 - s_1(q) = 7 - \bot = \bot$$

uninit. var.!

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$$[\![x + y]\!](s_1) = [\![x]\!](s_1) + [\![y]\!](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$[\![7 - q]\!](s_1) = [\![7]\!](s_1) - [\![q]\!](s_1) = 7 - s_1(q) = 7 - \bot = \bot$$

uninit. var.!

Writing “ $\bot$ ” here is short-hand for indicating **undefined** terms.

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$$[\![x + y]\!](s_1) = [\![x]\!](s_1) + [\![y]\!](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$[\![7 - q]\!](s_1) = [\![7]\!](s_1) - [\![q]\!](s_1) = 7 - s_1(q) = 7 - \perp = \perp$$

uninit. var.!

$$[\![12 / z]\!](s_1) = [\![12]\!](s_1) / [\![z]\!](s_1) = 12 / s_1(z) = 12 / 0 = \perp$$

Writing “ $\perp$ ” here is short-hand for indicating **undefined** terms.

Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$$[\![x + y]\!](s_1) = [\![x]\!](s_1) + [\![y]\!](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$[\![7 - q]\!](s_1) = [\![7]\!](s_1) - [\![q]\!](s_1) = 7 - s_1(q) = 7 - \bot = \bot$$

uninit. var.!

$$[\![12 / z]\!](s_1) = [\![12]\!](s_1) / [\![z]\!](s_1) = 12 / s_1(z) = 12 / 0 = \bot$$

$$[\![x \&\& y]\!](s_1) = [\![x]\!](s_1) \wedge [\![y]\!](s_1) = s_1(x) \wedge s_1(y) = 5 \wedge 42 = \bot$$

wrong type!

Writing “ $\bot$ ” here is short-hand for indicating **undefined** terms.

Statement Semantics

$$[\![\_]\!]_{\mathrm{S}} \quad : \ \mathit{Stmt} \to (\mathit{State} \nrightarrow \mathit{State})$$

For $s : \mathit{State}$, i.e., $s : \mathit{Id} \nrightarrow \mathit{SVal}$, and $p, p_1, p_2 : \mathit{Stmt}$ and $e : \mathit{Expr}$ and $v : \mathit{Id}$:

$$[\![\mathbf{skip}]\!]_{\mathrm{S}}\ (s) = s$$

$$[\![v \mathbf{:=} e]\!]_{\mathrm{S}}\ (s) = s \oplus \{v \mapsto [\![e]\!]_{\mathrm{E}}\ (s)\}$$

— **undefined** if $[\![e]\!]_{\mathrm{E}}\ (s)$ is undefined!

$$[\![p_1\ \mathbf{;}\ p_2]\!]_{\mathrm{S}} = [\![p_2]\!]_{\mathrm{S}} \circ [\![p_1]\!]_{\mathrm{S}}$$

$$[\![\mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2]\!]_{\mathrm{S}}\ (s) = \begin{cases} [\![p_1]\!]_{\mathrm{S}}\ (s) & \text{if } [\![e]\!]_{\mathrm{E}}\ (s) = \text{True} \\ [\![p_2]\!]_{\mathrm{S}}\ (s) & \text{if } [\![e]\!]_{\mathrm{E}}\ (s) = \text{False} \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Relating Simple Denotational and Operational Semantics

Simple Denotational		Simple Operational
$[\![e]\!]_{\mathrm{E}}\,(\sigma_1) = v$	$\Leftrightarrow$	$\sigma_1(e) \Rightarrow v$
$[\![s]\!]_{\mathrm{S}}\,(\sigma_1) = \sigma_2$	$\Leftrightarrow$	$\sigma_1(s) \Rightarrow \sigma_2$

Relating Simple Denotational and Operational Semantics

Simple Denotational		**Simple Operational**
$[\![e]\!]_E\,(\sigma_1) = v$	$\Leftrightarrow$	$\sigma_1(e) \Rightarrow v$
$[\![s]\!]_S\,(\sigma_1) = \sigma_2$	$\Leftrightarrow$	$\sigma_1(s) \Rightarrow \sigma_2$

- $[\![e]\!]_E$ and $[\![s]\!]_S$ are **explicit functions**

Relating Simple Denotational and Operational Semantics

Simple Denotational		**Simple Operational**
$[\![e]\!]_{\mathrm{E}}\,(\sigma_1) = v$	$\Leftrightarrow$	$\sigma_1(e) \Rightarrow v$
$[\![s]\!]_{\mathrm{S}}\,(\sigma_1) = \sigma_2$	$\Leftrightarrow$	$\sigma_1(s) \Rightarrow \sigma_2$

- $[\![e]\!]_{\mathrm{E}}$ and $[\![s]\!]_{\mathrm{S}}$ are **explicit functions**
- These can be considered as results of **function abstraction** from the operational semantics of *e* and *s*.

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “ $=$ ”.

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “ $=$ ”. Therefore, we need a way to denote a **function** on the right-hand side.

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of " $=$ ". Therefore, we need a way to denote a **function** on the right-hand side. λ-abstraction is such a notation:

$$f = \lambda x \bullet 2 * x - 3$$

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of " = ". Therefore, we need a way to denote a **function** on the right-hand side. λ-abstraction is such a notation:

$$f = \lambda\, x \bullet 2 * x - 3$$

This is equivalent to the above.

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of "$=$". Therefore, we need a way to denote a **function** on the right-hand side. λ-abstraction is such a notation:

$$f = \lambda\, x \bullet 2 * x - 3$$

This is equivalent to the above. Therefore:

$$f\ 5 \;=\; (\lambda\, x \bullet 2 * x - 3)\, 5 \;=\; 2 * 5 - 3$$

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of "$=$". Therefore, we need a way to denote a **function** on the right-hand side. λ-abstraction is such a notation:

$$f = \lambda x \bullet 2 * x - 3$$

This is equivalent to the above. Therefore:

$$f\ 5 \ = \ (\lambda x \bullet 2 * x - 3)\ 5 \ = \ 2 * 5 - 3$$

λ-abstraction **binds** a variable (here: x).

λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

$$f(x) = 2 * x - 3$$

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of " $=$ ". Therefore, we need a way to denote a **function** on the right-hand side. λ-abstraction is such a notation:

$$f = \lambda\, x \bullet 2 * x - 3$$

This is equivalent to the above. Therefore:

$$f\ 5 \ = \ (\lambda\, x \bullet 2 * x - 3)\, 5 \ = \ 2 * 5 - 3$$

λ-abstraction **binds** a variable (here: x). Application of a λ-abstraction to an argument is **reduced** to the body of the abstraction with the bound variable replaced by the argument.

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\ N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\, x \bullet B$ of an untyped λ-term B (the body) over a variable x.

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\,N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\, x \bullet B$ of an untyped λ-term B (the body) over a variable x.

Note: Every untyped λ-term can be used as function in function applications!

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\ N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\ x \bullet B$ of an untyped λ-term B (the body) over a variable x.

Note: Every untyped λ-term can be used as function in function applications!

Note: We add and omit parentheses using the rules that are used in Haskell:

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\ N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\, x \bullet B$ of an untyped λ-term B (the body) over a variable x.

Note: Every untyped λ-term can be used as function in function applications!

Note: We add and omit parentheses using the rules that are used in Haskell:

- λ-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\,N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\, x \bullet B$ of an untyped λ-term B (the body) over a variable x.

Note: Every untyped λ-term can be used as function in function applications!

Note: We add and omit parentheses using the rules that are used in Haskell:

- λ-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.
- Application associates to the left, i.e., $f\ x\ y$ is understood to mean $(f\ x)\ y$. According to the definition above, this would actually have to be $((f)\ x)\ y$.

λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots,$ or
- a **function application** $(M)\,N$ of one untyped λ-term F (the function) to another A (the argument), or
- a **function abstraction** $\lambda\, x \bullet B$ of an untyped λ-term B (the body) over a variable x.

Note: Every untyped λ-term can be used as function in function applications!

Note: We add and omit parentheses using the rules that are used in Haskell:

- λ-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.
- Application associates to the left, i.e., $f\ x\ y$ is understood to mean $(f\ x)\ y$. According to the definition above, this would actually have to be $((f)\ x)\ y$.

The λ-calculus was intended by its inventor, **Alonzo Church** (1903–1995), as a foundation of mathematics based on functions instead of on sets.

Free Variables

The set $FV(M)$ of the variables occurring **free** in the λ-term M is defined inductively over the construction of λ-terms (this is called: *structural induction*):

- $FV(x) = \{x\}$
- $FV(\lambda\, x \bullet M) = FV(M) \setminus \{x\}$
- $FV(M\ N) = FV(M) \cup FV(N)$

Variable Replacement (auxiliary concept)

$M[x \setminus y]$ denotes the term resulting from M by replacing all **free** occurrences of variable x with variable y:

- $v[x \setminus y] = \begin{cases} y & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$
- $(M\ N)[x \setminus y] = M[x \setminus y]\ N[x \setminus y]$
- $(\lambda\ v \bullet M)[x \setminus y] = \begin{cases} \lambda\ v \bullet M & \text{if } v = x \\ \lambda\ v \bullet (M[x \setminus y]) & \text{if } v \neq x \end{cases}$

— Variable replacement is **only** used in the definition of α-conversion.

α-Conversion

If $y \notin FV(M)$, and if there is no λ-binding for y in M, then the following **renaming of a bound variable** is defined:

$$\lambda\, x \bullet M \;\equiv_\alpha\; \lambda\, y \bullet M[x \setminus y]$$

This can also be applied in any context $C[\ ]$ (a context is a term with exactly one occurrence of the "hole" "[ ]"):

$$C[\ \lambda\, x \bullet M\ ] \;\equiv_\alpha\; C[\ \lambda\, y \bullet M[x \setminus y]\ ]$$

α-Conversion	**=**	**renaming of bound variables**

Substitution

Substitution is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M\ N)[x \setminus t] = M[x \setminus t]\ N[x \setminus t]$

- $(\lambda\, v \bullet M)[x \setminus t] = \begin{cases} \lambda\, v \bullet M & \text{if } v = x \lor x \notin FV(M) \\ \lambda\, v \bullet (M[x \setminus t]) & \text{if } v \neq x \land x \in FV(M) \land v \notin FV(t) \\ \textbf{\textit{not permitted!}} & \text{if } v \neq x \land x \in FV(M) \land v \in FV(t) \end{cases}$

Substitution

Substitution is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M\ N)[x \setminus t] = M[x \setminus t]\ N[x \setminus t]$

- $(\lambda\ v \bullet M)[x \setminus t] = \begin{cases} \lambda\ v \bullet M & \text{if } v = x \lor x \notin FV(M) \\ \lambda\ v \bullet (M[x \setminus t]) & \text{if } v \neq x \land x \in FV(M) \land v \notin FV(t) \\ \textbf{\textit{not permitted!}} & \text{if } v \neq x \land x \in FV(M) \land v \in FV(t) \end{cases}$

Where a substitution $[x \setminus t]$ is not permitted for a term M, an α-conversion $M \equiv_\alpha M'$ is always possible such that the substitution is permitted for M'.

Substitution

Substitution is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M\ N)[x \setminus t] = M[x \setminus t]\ N[x \setminus t]$

- $(\lambda\, v \bullet M)[x \setminus t] = \begin{cases} \lambda\, v \bullet M & \text{if } v = x \vee x \notin FV(M) \\ \lambda\, v \bullet (M[x \setminus t]) & \text{if } v \neq x \wedge x \in FV(M) \wedge v \notin FV(t) \\ \textbf{\textit{not permitted!}} & \text{if } v \neq x \wedge x \in FV(M) \wedge v \in FV(t) \end{cases}$

Where a substitution $[x \setminus t]$ is not permitted for a term M, an α-conversion $M \equiv_\alpha M'$ is always possible such that the substitution is permitted for M'.

Example:

$$(\lambda\, z \bullet f\ (z\ x))[x \setminus f\ z]$$

Substitution

Substitution is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M\ N)[x \setminus t] = M[x \setminus t]\ N[x \setminus t]$

- $(\lambda\, v \bullet M)[x \setminus t] = \begin{cases} \lambda\, v \bullet M & \text{if } v = x \vee x \notin FV(M) \\ \lambda\, v \bullet (M[x \setminus t]) & \text{if } v \neq x \wedge x \in FV(M) \wedge v \notin FV(t) \\ \textbf{\textit{not permitted!}} & \text{if } v \neq x \wedge x \in FV(M) \wedge v \in FV(t) \end{cases}$

Where a substitution $[x \setminus t]$ is not permitted for a term M, an α-conversion $M \equiv_\alpha M'$ is always possible such that the substitution is permitted for M'.

Example:

$$(\lambda\, z \bullet f\ (z\ x))[x \setminus f\ z] \equiv_\alpha (\lambda\, y \bullet f\ (y\ x))[x \setminus f\ z]$$

Substitution

Substitution is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M\ N)[x \setminus t] = M[x \setminus t]\ N[x \setminus t]$

- $(\lambda\, v \bullet M)[x \setminus t] = \begin{cases} \lambda\, v \bullet M & \text{if } v = x \vee x \notin FV(M) \\ \lambda\, v \bullet (M[x \setminus t]) & \text{if } v \neq x \wedge x \in FV(M) \wedge v \notin FV(t) \\ \textbf{\textit{not permitted!}} & \text{if } v \neq x \wedge x \in FV(M) \wedge v \in FV(t) \end{cases}$

Where a substitution $[x \setminus t]$ is not permitted for a term M, an α-conversion $M \equiv_\alpha M'$ is always possible such that the substitution is permitted for M'.

Example:

$$(\lambda\, z \bullet f\ (z\ x))[x \setminus f\ z] \ \equiv_\alpha\ (\lambda\, y \bullet f\ (y\ x))[x \setminus f\ z] \ = \ \lambda\, y \bullet f\ (y\ (f\ z))$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \to_{\beta} \quad B[x \setminus A]$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \rightarrow_{\beta} \quad B[x \setminus A]$$

This can also be applied in any context $C[\ ]$:

$$C[\ (\lambda\, x \bullet B)\, A\ ] \quad \rightarrow_{\beta} \quad C[\ B[x \setminus A\ ]]$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \rightarrow_{\beta} \quad B[x \setminus A]$$

This can also be applied in any context $C[\ ]$:

$$C[\ (\lambda\, x \bullet B)\, A\ ] \quad \rightarrow_{\beta} \quad C[\ B[x \setminus A\ ]]$$

Example:

$$(\lambda\, x \bullet \lambda\, z \bullet x\ (z\ x))\ (\lambda\, y \bullet z\ y) \qquad \rightarrow_{\beta}$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \to_{\beta} \quad B[x \setminus A]$$

This can also be applied in any context $C[\ ]$:

$$C[\ (\lambda\, x \bullet B)\, A\ ] \quad \to_{\beta} \quad C[\ B[x \setminus A\ ]]$$

Example:

$$(\lambda\, x \bullet \lambda\, z \bullet x\ (z\ x))\ (\lambda\, y \bullet z\ y) \qquad \to_{\beta} \qquad (\lambda\, z \bullet x\ (z\ x))[x \setminus (\lambda\, y \bullet z\ y)]$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \to_\beta \quad B[x \setminus A]$$

This can also be applied in any context $C[\ ]$:

$$C[\ (\lambda\, x \bullet B)\, A\ ] \quad \to_\beta \quad C[\ B[x \setminus A\ ]]$$

Example:

$$(\lambda\, x \bullet \lambda\, z \bullet x\ (z\ x))\ (\lambda\, y \bullet z\ y) \quad \to_\beta \quad (\lambda\, z \bullet x\ (z\ x))[x \setminus (\lambda\, y \bullet z\ y)]$$

$$\equiv_\alpha \quad (\lambda\, u \bullet x\ (u\ x))[x \setminus (\lambda\, y \bullet z\ y)]$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \rightarrow_{\beta} \quad B[x \setminus A]$$

This can also be applied in any context $C[\ ]$:

$$C[\ (\lambda\, x \bullet B)\, A\ ] \quad \rightarrow_{\beta} \quad C[\ B[x \setminus A\ ]]$$

Example:

$$\begin{array}{lll} (\lambda\, x \bullet \lambda\, z \bullet x\ (z\ x))\ (\lambda\, y \bullet z\ y) & \rightarrow_{\beta} & (\lambda\, z \bullet x\ (z\ x))[x \setminus (\lambda\, y \bullet z\ y)] \\ & \equiv_{\alpha} & (\lambda\, u \bullet x\ (u\ x))[x \setminus (\lambda\, y \bullet z\ y)] \\ & = & \lambda\, u \bullet ((\lambda\, y \bullet z\ y)\ (u\ (\lambda\, y \bullet z\ y))) \end{array}$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \quad \to_\beta \quad B[x \setminus A]$$

This can also be applied in any context $C[\;]$:

$$C[\; (\lambda\, x \bullet B)\, A \;] \quad \to_\beta \quad C[\; B[x \setminus A\;]]$$

Example:

$$\begin{aligned}
(\lambda\, x \bullet \lambda\, z \bullet x\, (z\, x))\, (\lambda\, y \bullet z\, y) \quad & \to_\beta & (\lambda\, z \bullet x\, (z\, x))[x \setminus (\lambda\, y \bullet z\, y)] \\
& \equiv_\alpha & (\lambda\, u \bullet x\, (u\, x))[x \setminus (\lambda\, y \bullet z\, y)] \\
& = & \lambda\, u \bullet ((\lambda\, y \bullet z\, y)\, (u\, (\lambda\, y \bullet z\, y))) \\
& \to_\beta & \lambda\, u \bullet ((z\, y)[y \setminus (u\, (\lambda\, y \bullet z\, y))])
\end{aligned}$$

β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda\, x \bullet B)\, A \;\; \to_{\beta} \;\; B[x \setminus A]$$

This can also be applied in any context $C[\;]$:

$$C[\, (\lambda\, x \bullet B)\, A \,] \;\; \to_{\beta} \;\; C[\, B[x \setminus A\,]]$$

Example:

$$\begin{array}{lll}
(\lambda\, x \bullet \lambda\, z \bullet x\, (z\, x))\, (\lambda\, y \bullet z\, y) & \to_{\beta} & (\lambda\, z \bullet x\, (z\, x))[x \setminus (\lambda\, y \bullet z\, y)] \\
 & \equiv_{\alpha} & (\lambda\, u \bullet x\, (u\, x))[x \setminus (\lambda\, y \bullet z\, y)] \\
 & = & \lambda\, u \bullet ((\lambda\, y \bullet z\, y)\, (u\, (\lambda\, y \bullet z\, y))) \\
 & \to_{\beta} & \lambda\, u \bullet ((z\, y)[y \setminus (u\, (\lambda\, y \bullet z\, y))]) \\
 & = & \lambda\, u \bullet (z\, (u\, (\lambda\, y \bullet z\, y)))
\end{array}$$

Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.
- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.
- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

"inner" and "outer" are determined by the abstract syntax tree.

Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

“inner” and “outer” are determined by the abstract syntax tree.

Important properties:

- Leftmost-outermost strategy (**Haskell**, Miranda, Clean):
  - **call by name, lazy evaluation**
  - terminates if possible
  - non-strict

Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.
- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

"inner" and "outer" are determined by the abstract syntax tree.

Important properties:

- Leftmost-outermost strategy (**Haskell**, Miranda, Clean):
  - **call by name, lazy evaluation**
  - terminates if possible
  - non-strict
- Leftmost-innermost strategy (**OCaml**, SML, LISP, Scheme):
  - **call by value, eager evaluation**
  - easier to implement
  - **strict:** for all f we have $f(\perp) = \perp$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \;=\; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y = \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$Y\ f \quad =$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \;=\; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$Y\ f \;=\; (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \;=\; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned} Y\ f \quad &= \quad (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\ &\to_{\beta} \quad (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \end{aligned}$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y = \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned} Y\ f &= (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\ &\to_\beta (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\ &= (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \end{aligned}$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \;=\; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned}
Y\ f \quad &= \quad (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\
&\to_\beta \quad (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\
&= \quad (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \\
&\to_\beta \quad f\ ((\lambda x \bullet f\ (x\ x))\ (\lambda x \bullet f\ (x\ x)))
\end{aligned}$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \; = \; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned}
Y\ f \quad & = && (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\
& \to_{\beta} && (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\
& = && (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \\
& \to_{\beta} && f\ ((\lambda x \bullet f\ (x\ x))\ (\lambda x \bullet f\ (x\ x))) \\
& = && f\ (Y\ f)
\end{aligned}$$

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \ = \ \lambda f \bullet (\lambda x \bullet f \ (x\ x))(\lambda x \bullet f \ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{array}{lll} Y\ f & = & (\lambda f \bullet (\lambda x \bullet f \ (x\ x))(\lambda x \bullet f \ (x\ x)))\ f \\ & \to_\beta & (\lambda x \bullet f \ (x\ x))(\lambda x \bullet f \ (x\ x)) \\ & = & (\lambda z \bullet f \ (z\ z))(\lambda x \bullet f \ (x\ x)) \\ & \to_\beta & f \ ((\lambda x \bullet f \ (x\ x))\ (\lambda x \bullet f \ (x\ x))) \\ & = & f \ (Y\ f) \end{array}$$

In the **theory of λ-calculus** this yields the fix edpointequation $Y\ f \ = \ f \ (Y\ f)$.

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y \;=\; \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned} Y\ f &= (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\ &\to_\beta (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\ &= (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \\ &\to_\beta f\ ((\lambda x \bullet f\ (x\ x))\ (\lambda x \bullet f\ (x\ x))) \\ &= f\ (Y\ f) \end{aligned}$$

In the **theory of λ-calculus** this yields the fix edpointequation $Y\ f \;=\; f\ (Y\ f)$. Therefore, for every f, a fix edpoint$Y\ f$ can be obtained via application of the **fixedpoint combinator** Y.

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y = \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned} Y\ f &= (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\ &\rightarrow_\beta (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\ &= (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \\ &\rightarrow_\beta f\ ((\lambda x \bullet f\ (x\ x))\ (\lambda x \bullet f\ (x\ x))) \\ &= f\ (Y\ f) \end{aligned}$$

In the **theory of λ-calculus** this yields the fix edpointequation $Y\ f = f\ (Y\ f)$. Therefore, for every f, a fix edpoint $Y\ f$ can be obtained via application of the **fixedpoint combinator** Y.

All general fix edpointcombinators involve **self-application** like "$x\ x$"

The Fixedpoint Combinator Y

Church's fix edpointcombinator "Y":

$$Y = \lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x))$$

Proof of fix edpointcombinator property — for every f, the following holds:

$$\begin{aligned} Y\ f &= (\lambda f \bullet (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)))\ f \\ &\rightarrow_\beta (\lambda x \bullet f\ (x\ x))(\lambda x \bullet f\ (x\ x)) \\ &= (\lambda z \bullet f\ (z\ z))(\lambda x \bullet f\ (x\ x)) \\ &\rightarrow_\beta f\ ((\lambda x \bullet f\ (x\ x))\ (\lambda x \bullet f\ (x\ x))) \\ &= f\ (Y\ f) \end{aligned}$$

In the **theory of λ-calculus** this yields the fix edpointequation $Y\ f = f\ (Y\ f)$. Therefore, for every f, a fix edpoint $Y\ f$ can be obtained via application of the **fixedpoint combinator** Y.

All general fix edpointcombinators involve **self-application** like "$x\ x$" — this possible in the untyped λ-calculus, but not in most typed systems.

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fixedpoint combinator

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fix edpointrules $Y_t\ f \to_Y f\ (Y_t\ f)$ are added to the rules.

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fix edpointrules $Y_t\ f \quad \to_Y \quad f\ (Y_t\ f)$ are added to the rules.
- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{N} \to \mathbb{N}}\ \tau\ 3 \quad \to_Y \quad \tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau)\ 3 \quad \to_Y \quad \tau\ (\tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau))\ 3 \quad \to_Y \quad \ldots$$

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fix edpointrules $Y_t\ f \quad \to_Y \quad f\ (Y_t\ f)$ are added to the rules.
- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{N} \to \mathbb{N}}\ \tau\ 3 \quad \to_Y \quad \tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau)\ 3 \quad \to_Y \quad \tau\ (\tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau))\ 3 \quad \to_Y \quad \ldots$$

- We write "Y" also as fix edpointcombinator in a mathematical context

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fix edpointrules $Y_t\ f \quad \to_Y \quad f\ (Y_t\ f)$ are added to the rules.
- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{N} \to \mathbb{N}}\ \tau\ 3 \quad \to_Y \quad \tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau)\ 3 \quad \to_Y \quad \tau\ (\tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau))\ 3 \quad \to_Y \quad \ldots$$

- We write "Y" also as fix edpointcombinator in a mathematical context
- More precisely, we let "$Y\ F$" denote the **least fixedpoint** of F

General Fixedpoint Combinators

- In typed λ-calculi, no pure λ-term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types t, the fix edpointcombinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fix edpointrules $Y_t\ f \quad \to_Y \quad f\ (Y_t\ f)$ are added to the rules.
- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{N} \to \mathbb{N}}\ \tau\ 3 \quad \to_Y \quad \tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau)\ 3 \quad \to_Y \quad \tau\ (\tau\ (Y_{\mathbb{N} \to \mathbb{N}}\ \tau))\ 3 \quad \to_Y \quad \ldots$$

- We write "Y" also as fix edpointcombinator in a mathematical context
- More precisely, we let "$Y\ F$" denote the **least fixedpoint** of F
- Other notations: "$\mu\ F$", or "fix F"

Parameter Passing

Parameter passing has two sides — assume a parameterized subprogram *P*:

- **Formal Parameters:** The names used to refer to the parameters in the definition of *P*.
- **Actual Parameters:** The expressions supplied to *P* as instances of its parameters for the purpose of creating an *incarnation* of *P*.

If several actual parameters are supplied to a subprogram defined with several formal parameters, how is a **correspondence** established?

- **Positional correspondence:** *n*-th actual parameter instantiates *n*-th formal parameter
- **Explicit parameter labels:** allow arbitrary order as far as labels are supplied (often position al fallback).

Ada: `Sub( Y => B, X => 27 );`

OCaml: `sub ~y:b ~x:27`

Labelled Arguments in OCaml

`#let f ~x ~y = x - y;;` `val f : x:int -> y:int -> int = <fun>` — x and y are **labels**

`#let x = 3 and y = 2 in f ~x ~y;;` — x and y are **labelled arguments** `- : int = 1`

`#let x = 3 and y = 2 in f ~y ~x;;` — labelled arguments may be **commuted** `- : int = 1`

`#let f ~x:x1 ~y:y1 = x1 - y1;;` — x1 and y1 are **formal parameters**

`val f : x:int -> y:int -> int = <fun>` — x and y are **labels**

`#f ~x:3 ~y:2;;` `- : int = 1` `#f ~y:2 ~x:3;;` `- : int = 1`

`#f 3 2;;` — labels can be omitted if **all** arguments are supplied! `- : int = 1`

Typical Application of Labelled Arguments in OCaml

```
ListLabels.fold_right : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b
val list_predsplit : f:('a -> bool) -> 'a list -> 'a list * 'a list
let list_predsplit ~f = ListLabels.fold_right  ~init:([],[])
      ~f:(fun x (xs,ys) -> if f x then (x :: xs,ys) else (xs,x::ys));;
val of_psp : ((f * 'a) list * f) -> 'a t let of_psp (ps,tl) =   let d = singleton tl in
  List.fold_right ~init:d ps    ~f:(fun p d -> let _ = fe_onto_start p d in d);;
    ...  (List.fold_right ns ~init:[] ~f:(fun n res ->
              (try (match snd (Nodemap.find attrmap ~key:n) with
                      None -> res                    | Some (Left h) -> (n,h) :: res
                    | Some (Right q) -> res                    )
               with Not_found -> res          )));;
```

Optional Arguments in OCaml

```
#let bump ?(step = 1) x = x + step;;
val bump : ?step:int -> int -> int = <fun>
```

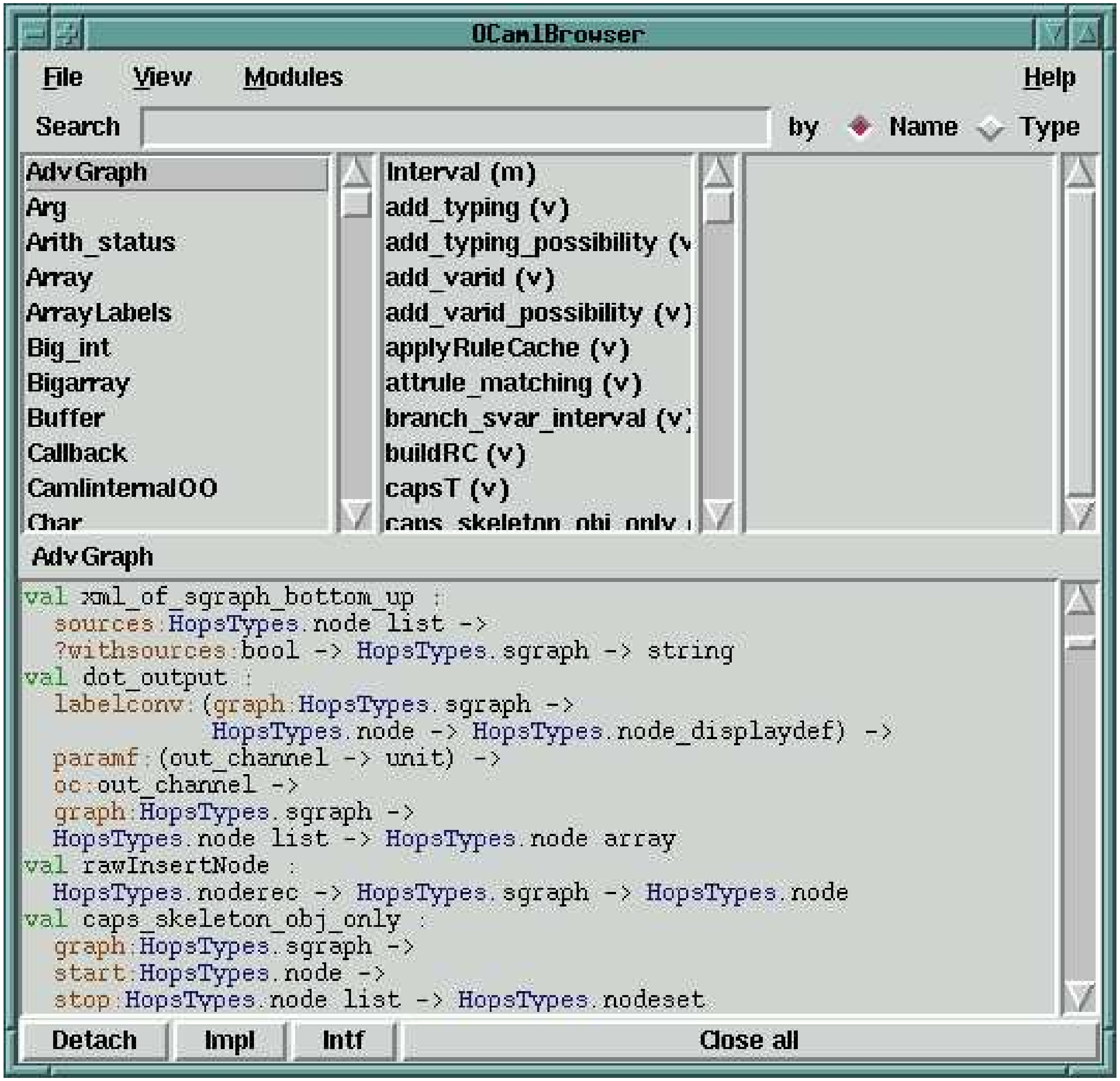
— optional arguments have a label and a default value.

```
#bump 2;;
- : int = 3
```

```
#bump ~step:3 2;;
- : int = 5
```

A function taking some optional arguments must also take at least one non-labeled argument. (Important for partial applications)

OCamlBrowser



Evaluation Aspects of Parameter Passing

- **Call by value:** Actual parameter expression is evaluated to a **value** before instantiation of formal parameter.
  - leftmost-innermost strategy in λ-calculus
  - OCaml, C, Java, Oberon, Ada, …
  - **strict** function call semantics: undefined arguments **always** produce undefined results

```
# const 4 (3 / 0);;
Exception: Division_by_zero.
```

- **Call by name:** Formal parameter f is instantiated with unevaluated actual parameter expression e — several occurrences of f result in several copies of e.
  - leftmost-outermost strategy in λ-calculus

- **Lazy evaluation:** call by name with **sharing** instead of copying: e is never duplicated; all occurrences of f are instantiated with references to single e, which is evaluated at most once.

- graph reduction implementation of leftmost-outermost
- most Haskell implementations
- **non-strict** function call semantics: undefined arguments **need not** produce undefined results

```
Prelude> const 4 (3 / 0)
4
```

Storage Aspects of Parameter Passing

- **Call by constant value:** value available as local constant
- **Call by copy** (call by value): value copied into local variable
- **Call by reference:** actual parameter needs to be a reference, or define a reference; this is used as value of formal parameter.
- **Call by value-result:** actual parameter needs to be a reference, or define a reference *r*; local variable *l* is initialized from r-value of *r*, and on subprogram exit, *r* is overwritten with contents of *l*.
- **Call by result:** Similar, but *l* starts out uninitialized.

Call by value, Call by reference, and Scoping

```
MODULE Scope1;
IMPORT Out;
VAR n : INTEGER;
PROCEDURE B(VAR x : INTEGER;
            z : INTEGER);
  VAR hv : INTEGER;
  BEGIN
    IF z = 0
    THEN x := 0
    ELSE hv := x;
         B(x, z-1);
         x := x+hv  END;
  END B;
```

```
PROCEDURE A(y,x: INTEGER;
            VAR result: INTEGER);
  BEGIN
    IF x = 0
    THEN result:=1;
    ELSE A(y, x-1, result);
         B(result, y)       END;
  END A;
BEGIN
  n := 0;
  A(2,1,n);
  Out.Int(n,0); Out.Ln
END Scope1.
```

Call by value, Call by reference, and Scoping

```
MODULE Scope1;
IMPORT Out;
VAR n : INTEGER;
PROCEDURE B(VAR x : INTEGER;
            z : INTEGER);
  VAR hv : INTEGER;
  BEGIN
    IF z = 0
    THEN x := 0
    ELSE hv := x;
         B(x, z-1);
         x := x+hv  END;
  END B;
```

```
PROCEDURE A(y,x: INTEGER;
            VAR result: INTEGER);
  BEGIN
    IF x = 0
    THEN result:=1;
    ELSE A(y, x-1, result);
         B(result, y)      END;
  END A;
BEGIN
  n := 0;
  A(2,1,n);
  Out.Int(n,0); Out.Ln
END Scope1.
```

$$\{\mathtt{x} = a\} \quad \mathtt{B(x,z)} \quad \{\mathtt{x} = a * \mathtt{z}\}$$

$$\{\mathsf{True}\} \quad \mathtt{A(y,x,result)} \quad \{\mathtt{result} = \mathtt{y}^{\mathtt{x}}\}$$

Parameter Passing

- Formal parameters — actual parameters
- **Correspondence aspects:** by position, by label, optional arguments
- **Evaluation aspects:** call by value, call by name, lazy evaluation
- **Storage aspects:** call by constant value, call by copy, call by reference, call by value-result, call by result

X

X

X

X