

## Another Loop Example

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

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

**This is not a direct proof of non-termination!**

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

## From Operational to Relational Denotational Semantics

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

“Evaluating expression  $e$  starting in state  $\sigma$  can produce value  $v$ ”

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

which is equivalent to a total relation-valued function:

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

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

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

$$M_{\text{Expr}} : \text{Expr} \rightarrow (\text{State}_1 \mapsto \text{Value})$$

This corresponds to the Haskell type we have chosen:

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

**Note:** For an expression  $e$ , we write “ $\llbracket e \rrbracket_E$ ” instead of “ $M_{\text{Expr}}(e)$ ”.

## Relational Denotational Statement Semantics

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

“Executing statement  $s$  starting in state  $\sigma$  can terminate in state  $\sigma'$ ”

give rise to a ternary relation  $\text{execStmt}_0 : \mathbb{P}(\text{State}_1 \times \text{Stmt} \times \text{State}_1)$

which is equivalent to a total relation-valued function:

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)$$

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

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \mapsto \text{State}_1)$$

**Note:** For a statement  $s$ , we write “ $\llbracket s \rrbracket_S$ ” instead of “ $M_{\text{Stmt}}(s)$ ”.

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

$$\{s \mapsto 0, x \mapsto -4\} \notin \text{dom } \llbracket \text{while } x \neq 0 \text{ do } s := s + x ; x := x - 1 \text{ od} \rrbracket_S$$

**This uses properties of mathematical object found as denotational semantics of a statement.**

## Denotational Semantics is Compositional

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

We can reflect these parentheses in the definition:

$\text{evalExpr} (\text{Var } v) = \text{Map.lookup } v$

$\text{evalExpr} (\text{Value } lit) = \text{const } (\text{Just } (\text{litToVal } lit))$

$\text{evalExpr} (\text{Binary } (\text{MkArithOp } Plus) e1 e2) = \lambda s \rightarrow$

**case** (( $\text{evalExpr } e1$ )  $s$ , ( $\text{evalExpr } e2$ )  $s$ ) **of**

( $\text{Just } (\text{Vallnt } v1)$ ,  $\text{Just } (\text{Vallnt } v2)$ )  $\rightarrow \text{Just } (\text{Vallnt } (v1 + v2))$

–  $\rightarrow \text{Nothing}$

Translating the last case back into mathematical notation:

$$\llbracket e_1 + e_2 \rrbracket_E := \lambda s \rightarrow \llbracket e_1 \rrbracket_E s + \llbracket e_2 \rrbracket_E s \quad (\text{using partial operations})$$

### Compositional Semantics

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

## Sequencing, Conditionals, Loops in Operational Semantics

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

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

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

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

The last operational semantics rule here is not compositional!

### Interpreter: Loops

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

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

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

$\text{Just} (\text{ValBool False}) \rightarrow \text{Just } s$

$\text{Just} (\text{ValBool True}) \rightarrow \text{case} (\text{interpStmt body}) s \text{ of}$

$\text{Just } s1 \rightarrow (\text{interpStmt} (\text{Loop cond body})) s1$

$\text{Nothing} \rightarrow \text{Nothing}$

$\text{Just} (\text{ValInt } i) \rightarrow \text{Nothing}$

$\text{Nothing} \rightarrow \text{Nothing}$

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

“ $\text{interpStmt} (\text{Loop cond body})$ ” occurs also on the right-hand side.

## Recursive Definitions?

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

$$\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S)$$

In *mathematics*:

- “ $x = \langle\langle \text{term not containing } x \rangle\rangle$ ” 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*:

“ $\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S)$ ” considered as equation in

“ $\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S$ ” usually has **many** solutions!

## Recursive Function Definitions

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

Start (Haskell):

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

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

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

Reverse  $\beta$ -reduction to isolate the RHS occurrence of *fact*:

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

Defining (via an explicit definition)

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

we recognise a **fixedpoint equation** (stating that *fact* is a fixedpoint of  $\tau$ ):

$$\text{fact} = \tau \text{ fact}$$

## Fixedpoint Approximation

For the functional  $\tau$  associated with the definition of the factorial function *fact*, we observe:

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

In addition:

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

Iterated application of  $\tau$  yields better and better **finite approximations!**

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

## 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”  $\tau$  is extracted from that equation.
- Use  $\tau$  for **fixedpoint iteration**

$$\perp \sqsubseteq \tau \perp \sqsubseteq \tau^2 \perp \sqsubseteq \tau^3 \perp \sqsubseteq \dots$$

- The semantics of  $f$  is the least upper bound of this chain:

$$\llbracket f \rrbracket = \bigcup \{k : \mathbf{N} \bullet \tau^k \perp\}$$

- This least upper bound is the **least fixedpoint** of  $\tau$

- We write “ $Y \tau$ ” for the **least fixedpoint** of  $\tau$ .

## while-Loop Semantics

$$\llbracket - \rrbracket_S : Stmt \rightarrow (State \mapsto State)$$

For  $p : Stmt$  and  $e : Expr$ :

$$\begin{aligned} \llbracket \mathbf{while} \ e \ \mathbf{do} \ p \rrbracket_S &= \mathbf{Y} (\lambda f : State \mapsto State \bullet \lambda s : State \bullet \left. \begin{array}{ll} f(\llbracket p \rrbracket_S(s)) & \text{if } \llbracket e \rrbracket_E(s) = \text{True} \\ s & \text{if } \llbracket e \rrbracket_E(s) = \text{False} \\ \perp & \text{otherwise} \end{array} \right\}) \end{aligned}$$

## Example Statement Semantics

$$\begin{aligned} \llbracket \mathbf{while} \ \text{True} \ \mathbf{do} \ \mathbf{skip} \rrbracket_S &= \mathbf{Y} (\lambda f : State \mapsto State \bullet \lambda s : State \bullet f(\llbracket \mathbf{skip} \rrbracket_S(s))) \\ &= \mathbf{Y} (\lambda f : State \mapsto State \bullet \lambda s : State \bullet f(s)) \\ &= \mathbf{Y} (\lambda f : State \mapsto State \bullet f) \\ &= \perp \end{aligned}$$

For  $k : \mathbf{N}$ , we have:

$$\llbracket \mathbf{while} \ n > 0 \ \mathbf{do} \ (r := n * r ; n := n - 1) \rrbracket_S (\{n \mapsto k, r \mapsto 1\}) = \{n \mapsto 0, r \mapsto k!\}$$

## 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**