

## Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

$$\llbracket \text{Statement} \rrbracket = F$$

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

- a **definedness ordering**  $\sqsubseteq$ , and
- a **least element**  $\perp$  (read: “**bottom**”) wrt.  $\sqsubseteq$ :

$$\forall x : D \bullet \perp \sqsubseteq x$$

- (least upper bounds of chains  $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ )

For **simple** semantics of imperative programs, **sets of partial functions**  $A \rightarrow B$  can be used as domains:

- the subset ordering  $\subseteq$  serves as definedness ordering:

$$\forall f, g : A \rightarrow B \bullet f \sqsubseteq g :\Leftrightarrow f \subseteq g$$

- the empty function  $\emptyset : A \rightarrow B$  is the a least element of  $A \rightarrow B$ .

## Semantic Domains for Simple Imperative Programs

$Bool$	$= \{\text{True}, \text{False}\}$	booleans
$Num$	$= \mathbb{Z}$	numbers
$SVal$	$= Bool + Num$	storable values
$Id$		identifiers
$State$	$= Id \rightarrow SVal$	(simple) stores
$Val$	$= SVal$	values
$State \rightarrow Val$		(expression semantics)
$State \rightarrow State$		(statement semantics)

## 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*  $a\ b = \text{Left } a \mid \text{Right } b$

This produces the two **constructors** for *Either* (which are **injections**):

*Left*  $:: a \rightarrow \text{Either } a\ b$

*Right*  $:: b \rightarrow \text{Either } a\ b$

and allows pattern matching:

*valShow*  $:: \text{Either Integer Bool} \rightarrow \text{String}$

*valShow* (*Left*  $i$ ) = “int: ” ++ *show*  $i$

*valShow* (*Right*  $b$ ) = “bool: ” ++ *show*  $b$

In mathematical use, *Left* and *Right* are frequently not mentioned.

## Semantic Functions

$\llbracket \_ \rrbracket_E : Expr \rightarrow (State \mapsto Val)$	expression semantics
$\llbracket \_ \rrbracket_S : Stmt \rightarrow (State \mapsto State)$	statement semantics
$\llbracket \_ \rrbracket_O : Op \rightarrow ((Val \times Val) \mapsto Val)$	operator semantics (given)

**Textbook:**

$M$	$: (Expr \times State) \mapsto Val$	expression semantics
$M$	$: (Stmt \times State) \mapsto State$	statement semantics
$ApplyBinary$	$: (Op \times Val \times Val) \mapsto Val$	operator semantics

- No clean separation between syntax and semantics
- Undefinedness ordering less obvious

## Expression Semantics

$Expr ::= Id \mid Num \mid Bool \mid Expr \ Op \ Expr$

$\llbracket \_ \rrbracket_E : Expr \rightarrow (State \mapsto Val)$

Assuming  $s : State$ , i.e.,  $s : Id \mapsto SVal$ , we define:

for  $v : Id$ :  $\llbracket v \rrbracket_E (s) = s(v)$   
 — **undefined** if  $s(v)$  is undefined!

for  $n : Num$ :  $\llbracket n \rrbracket_E (s) = n$

for  $b : Bool$ :  $\llbracket b \rrbracket_E (s) = b$

for  $e_1, e_2 : Expr$ ;  $op : Op$ :  $\llbracket e_1 \ op \ e_2 \rrbracket_E (s) = \llbracket op \rrbracket_O (\llbracket e_1 \rrbracket_E (s), \llbracket e_2 \rrbracket_E (s))$   
 — **undefined** if  $\llbracket e_1 \rrbracket_E (s)$  or  $\llbracket e_2 \rrbracket_E (s)$  undefined, or from  $\llbracket op \rrbracket_O!$

Where clear from the context, we write  $\llbracket e \rrbracket$  instead of  $\llbracket e \rrbracket_E$ .

## Expression Semantics — Examples

**Examples:** Let  $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$ :

$$\llbracket x + y \rrbracket (s_1) = \llbracket x \rrbracket (s_1) + \llbracket y \rrbracket (s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$\llbracket 7 - q \rrbracket (s_1) = \llbracket 7 \rrbracket (s_1) - \llbracket q \rrbracket (s_1) = 7 - s_1(q) = 7 - \perp = \perp$$

**uninit. var.!**

$$\llbracket 12 / z \rrbracket (s_1) = \llbracket 12 \rrbracket (s_1) / \llbracket z \rrbracket (s_1) = 12 / s_1(z) = 12 / 0 = \perp$$

$$\llbracket x \&\&y \rrbracket (s_1) = \llbracket x \rrbracket (s_1) \wedge \llbracket y \rrbracket (s_1) = s_1(x) \wedge s_1(y) = 5 \wedge 42 = \perp$$

**wrong type!**

Writing “ $\perp$ ” here is short-hand for indicating **undefined** terms.

## Statement Semantics

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

For  $s : State$ , i.e.,  $s : Id \mapsto SVal$ , and  $p, p_1, p_2 : Stmt$  and  $e : Expr$  and  $v : Id$ :

$$\llbracket \text{skip} \rrbracket_S (s) = s$$

$$\llbracket v := e \rrbracket_S (s) = s \oplus \{v \mapsto \llbracket e \rrbracket_E (s)\}$$

— **undefined** if  $\llbracket e \rrbracket_E (s)$  is undefined!

$$\llbracket p_1 ; p_2 \rrbracket_S = \llbracket p_2 \rrbracket_S \circ \llbracket p_1 \rrbracket_S$$

$$\llbracket \text{if } e \text{ then } p_1 \text{ else } p_2 \rrbracket_S (s) = \begin{cases} \llbracket p_1 \rrbracket_S (s) & \text{if } \llbracket e \rrbracket_E (s) = \text{True} \\ \llbracket p_2 \rrbracket_S (s) & \text{if } \llbracket e \rrbracket_E (s) = \text{False} \\ \text{undefined} & \text{otherwise} \end{cases}$$

## Relating Simple Denotational and Operational Semantics

### Simple Denotational

$$\llbracket e \rrbracket_E (\sigma_1) = v$$

 $\Leftrightarrow$ 

### Simple Operational

$$\sigma_1(e) \Rightarrow v$$

$$\llbracket s \rrbracket_S (\sigma_1) = \sigma_2$$

 $\Leftrightarrow$ 

$$\sigma_1(s) \Rightarrow \sigma_2$$

- $\llbracket e \rrbracket_E$  and  $\llbracket s \rrbracket_S$  are **explicit functions**
- These can be considered as results of **function abstraction** from the operational semantics of  $e$  and  $s$ .

## $\lambda$ -Calculus (Textbook 8.1) — Motivation for the $\lambda$ -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.  $\lambda$ -abstraction is such a notation:

$$f = \lambda x \bullet 2 * x - 3$$

This is equivalent to the above. Therefore:

$$f \ 5 = (\lambda x \bullet 2 * x - 3) \ 5 = 2 * 5 - 3$$

$\lambda$ -abstraction **binds** a variable (here:  $x$ ). Application of a  $\lambda$ -abstraction to an argument is **reduced** to the body of the abstraction with the bound variable replaced by the argument.

## $\lambda$ -Terms

Now the formal definition of **untyped  $\lambda$ -terms**: An untyped  $\lambda$ -term is either

- a **variable**  $x, y, z, \dots$ , or
- a **function application**  $(M) N$  of one untyped  $\lambda$ -term  $F$  (the function) to another  $A$  (the argument), or
- a **function abstraction**  $\lambda x \bullet B$  of an untyped  $\lambda$ -term  $B$  (the body) over a variable  $x$ .

**Note:** Every untyped  $\lambda$ -term can be used as function in function applications!

**Note:** We add and omit parentheses using the rules that are used in Haskell:

- $\lambda$ -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  $\lambda$ -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  $\lambda$ -term  $M$  is defined inductively over the construction of  $\lambda$ -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  $\alpha$ -conversion.

## $\alpha$ -Conversion

If  $y \notin FV(M)$ , and if there is no  $\lambda$ -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” “[ ]”):

$$C[ \lambda x \bullet M ] \equiv_{\alpha} C[ \lambda y \bullet M[x \setminus y] ]$$

$\alpha$ -Conversion	=	renaming of bound variables
----------------------	---	-----------------------------

## Substitution

**Substitution** is replacement of free variables by terms:

- $v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$
- $(M N)[x \setminus t] = M[x \setminus t] N[x \setminus t]$
- $(\lambda v \bullet M)[x \setminus t] = \begin{cases} \lambda v \bullet M & \text{if } v = x \vee x \notin FV(M) \\ \lambda v \bullet (M[x \setminus t]) & \text{if } v \neq x \wedge x \in FV(M) \wedge v \notin FV(t) \\ \text{not permitted!} & \text{if } v \neq x \wedge x \in FV(M) \wedge v \in FV(t) \end{cases}$

Where a substitution  $[x \setminus t]$  is not permitted for a term  $M$ , an  $\alpha$ -conversion  $M \equiv_{\alpha} M'$  is always possible such that the substitution is permitted for  $M'$ .

**Example:**

$$(\lambda z \bullet f(z x))[x \setminus f z] \equiv_{\alpha} (\lambda y \bullet f(y x))[x \setminus f z] = \lambda y \bullet f(y(f z))$$

## $\beta$ -Reduction

The central reduction rule of  $\lambda$ -calculus:

$$(\lambda x \bullet B) A \rightarrow_{\beta} B[x \setminus A]$$

This can also be applied in any context  $C[ \ ]$ :

$$C[ (\lambda x \bullet B) A ] \rightarrow_{\beta} C[ B[x \setminus A] ]$$

**Example:**

$$\begin{aligned} (\lambda x \bullet \lambda z \bullet x(z x)) (\lambda y \bullet z y) &\rightarrow_{\beta} (\lambda z \bullet x(z x))[x \setminus (\lambda y \bullet z y)] \\ &\equiv_{\alpha} (\lambda u \bullet x(u x))[x \setminus (\lambda y \bullet z y)] \\ &= \lambda u \bullet ((\lambda y \bullet z y)(u(\lambda y \bullet z y))) \\ &\rightarrow_{\beta} \lambda u \bullet ((z y)[y \setminus (u(\lambda y \bullet z y))]) \\ &= \lambda u \bullet (z(u(\lambda y \bullet z y))) \end{aligned}$$

## 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  $f(\perp) = \perp$

## Recursive Function Definitions

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

Start (Haskell):

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

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

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

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

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

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 recognize a **fixedpoint equation** (stating that  $fact$  is a fix edpoint of  $\tau$ ):

$$fact = \tau\ fact$$

## 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  $\tau$

## 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  $\lambda$ -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  $\lambda$ -calculus, but not in most typed systems.

## General Fixedpoint Combinators

- In typed  $\lambda$ -calculi, no pure  $\lambda$ -term is a fix edpointcombinator
- One can always **extend the calculus**:
  - For at least some types  $t$ , the fix edpointcombinator  $Y_t : (t \rightarrow t) \rightarrow t$  is added to the terms.
  - The fix edpointrules  $Y_t f \rightarrow_Y f(Y_t f)$  are added to the rules.
- **Note**: This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{N} \rightarrow \mathbb{N}} \tau 3 \rightarrow_Y \tau (Y_{\mathbb{N} \rightarrow \mathbb{N}} \tau) 3 \rightarrow_Y \tau (\tau (Y_{\mathbb{N} \rightarrow \mathbb{N}} \tau)) 3 \rightarrow_Y \dots$$

- 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  $F$ ”

## while-Loop Semantics

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

For  $s : State$ , i.e.,  $s : Id \mapsto SVal$ , and  $p, p_1, p_2 : Stmt$  and  $e : Expr$  and  $v : Id$ :

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

## 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  $k : \mathbb{N}$ , we have:

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

## Semantics with Exceptions — Simple Statements

$$\llbracket \_ \rrbracket_S : Stmt \rightarrow (Store \mapsto (Store + (Store \times Num)))$$

$$\llbracket \text{skip} \rrbracket_S = Left$$

$$\llbracket s_1 ; s_2 \rrbracket_S (s) = \begin{cases} \llbracket s_2 \rrbracket_S (t) & \text{if } \llbracket s_1 \rrbracket_S (s) = Left\ t \\ Right(t, e) & \text{if } \llbracket s_1 \rrbracket_S (s) = Right\ (t, e) \end{cases}$$

$$\llbracket \text{try } s_1 \text{ catch } (i) s_2 \rrbracket_S (s) = \begin{cases} t & \text{if } \llbracket s_1 \rrbracket_S (s) = Left\ t \\ \llbracket s_2 \rrbracket_S (t \oplus \{i \mapsto e\}) & \text{if } \llbracket s_1 \rrbracket_S (s) = Right\ (t, e) \\ \perp & \text{if } s \notin \text{dom } \llbracket s_1 \rrbracket_S \end{cases}$$

## Semantics with Exceptions — Expressions

$$Expr \rightarrow (Store \rightarrow (Val + Num))$$

$$\llbracket \text{throw } e \rrbracket_S (s) = \begin{cases} Right\ (s, val) & \text{if } \llbracket e \rrbracket_E (s) = Left\ val \\ Right\ (s, exc) & \text{if } \llbracket e \rrbracket_E (s) = Right\ exc \end{cases}$$

$$\llbracket v := e \rrbracket_S (s) = \begin{cases} Left\ (s \oplus \{v \mapsto val\}) & \text{if } \llbracket e \rrbracket_E (s) = Left\ val \\ Right\ (s, exc) & \text{if } \llbracket e \rrbracket_E (s) = Right\ exc \end{cases}$$

$$\llbracket \text{if } b \text{ then } s_1 \text{ else } s_2 \rrbracket_S (s) = \begin{cases} \llbracket s_1 \rrbracket_S (s) & \text{if } \llbracket b \rrbracket_E (s) = Left\ True \\ \llbracket s_2 \rrbracket_S (s) & \text{if } \llbracket b \rrbracket_E (s) = Left\ False \\ Right\ (s, exc_{\mathbf{B}}) & \text{if } \llbracket b \rrbracket_E (s) = Left\ n \wedge n \in Num \\ Right\ (s, exc) & \text{if } \llbracket b \rrbracket_E (s) = Right\ exc \end{cases}$$

## Output

**Semantic Domains** for simple imperative programs with **print** statements:

$SVal$	$= Bool + Num$	storable values
$Store$	$= Id \mapsto SVal$	(simple) stores
$State$	$= Store^\perp \times [Num]$	<b>states including output</b>
$Val$	$= SVal$	values
$Store \mapsto Val$		(expression semantics)
$State \rightarrow State$		<b>(new statement semantics)</b>

In case of program errors or nontermination, **previous output is not lost!**

$$\llbracket \text{print } e \rrbracket_S = \lambda (s, ns) : State \bullet \begin{cases} (s, n:ns) & \text{if } n = \llbracket e \rrbracket_E (s) \in Num \\ (\perp, ns) & \text{otherwise} \end{cases}$$

**Note:** statement semantics here is *oversimplified* — fixpoint construction in  $State \rightarrow State$  does not work, except with Haskell-like list domains.

## Input

- 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**:

$$State = Store^\perp \times [Num] \times [Num]$$

$$\llbracket \text{read } v \rrbracket_S =$$

$$\lambda (s, outs, ins) : State \bullet \begin{cases} (s \oplus \{v \mapsto in\}, outs, ins') & \text{if } ins = in:ins' \\ (\perp, outs, ins) & \text{if } ins = [] \end{cases}$$

## Scope

Nested scopes with shadowing of identifiers are modelled as **stacks** (lists) of **environments**:

$Env = Id \mapsto SVal^\perp$       **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:

$SVal = Bool + Num + (Id \mapsto SVal)$       storable values  
 $State = Id \mapsto SVal$       (simple) stores  
 $State \mapsto SVal$       (expression semantics)  
 $State \mapsto State$       (statement semantics)

New record field expressions:

$$\llbracket e.f \rrbracket_E = \lambda s : State \bullet (\llbracket e \rrbracket_E(s)) f$$

New record construction expressions (not in C or Oberon, but e.g. in Ada):

$$\llbracket \mathbf{record}(f_1 = e_1, \dots, f_n = e_n) \rrbracket_E = \lambda s : State \bullet \{f_1 \mapsto \llbracket e_1 \rrbracket_E(s), \dots, f_n \mapsto \llbracket e_n \rrbracket_E(s)\}$$

New record field assignment statements:

$$\llbracket [r.f := e] \rrbracket_S = \lambda s : State \bullet s \oplus \{r \mapsto ((s \ r) \oplus \{f \mapsto \llbracket e \rrbracket_E(s)\})\}$$