**Operational semantics** (state and trace) (last two weeks)
Defined as small execution steps, over low-level internal configurations.
Transitions are chained to define maximal traces.

**Denotational semantics** (today)
Direct functions from programs to mathematical objects, defined by induction on the program syntax, ignoring intermediate steps and execution details.

⇒ Higher-level, more abstract, more modular.
Tries to decouple a program meaning from its execution.
Focus on the mathematical structures that represent programs.
(founded by Strachey and Scott in the 70s: [Scott-Strachey71])

“Assembly” semantics vs. “Functional programming” semantics.
often: semantics for practical verification vs. semantics for computer theorists
Two very different programs

Bubble sort in C

```c
int swapped;
do {
    swapped = 0;
    for (int i=1; i<n; i++) {
        if (a[i-1] > a[i]) {
            swap(&a[i-1], &a[i]);
            swapped = 1;
        }
    }
}
while (swapped);
```

Quick sort in OCaml

```ocaml
let rec sort = function
| [] -> []
| a::rest ->
    let lo, hi =
        List.partition
            (fun y -> y < x) rest
    in
    (sort lo) @ [x] @ (sort hi)
```

different **languages** (C / OCaml)
different **algorithms** (bubble sort / quick sort)
different **programming principles** (loop / recursion)
different **data-types** (array / list)

Can we give them the same semantics?
Denotation worlds

- **imperative programs**
  
  effect of a program: mutate a memory state
  natural denotation: input/output function
  domain $\mathcal{D} \simeq \text{memory} \rightarrow \text{memory}$
  
  challenge: build a whole program denotation from denotations of atomic language constructs (modularity)

- **functional programs**
  
  effect of a program: return a value (without any side-effect)
  model a program of type $a \rightarrow b$ as a function in $\mathcal{D}_a \rightarrow \mathcal{D}_b$
  
  challenge: choose $\mathcal{D}$ to allow polymorphic or untyped languages

- other paradigms: parallel, probabilistic, etc.

$\Rightarrow$ very rich theory of mathematical structures

Scott domains, cartesian closed categories, coherent spaces, event structures, game semantics, etc. We will not present them in this overview!
Course overview

- **Imperative programs**
  - IMP: deterministic programs
  - NIMP: handling non-determinism
  - linking denotational and operational semantics

- **Higher-order programs**
  - PCF: monomorphic typed programs
  - linking denotational and operational semantics: full abstraction
  - untyped λ-calculus: recursive domain equations

- **Practical session** (room INFO 4)
  - program the denotational semantics of a simple imperative (non-)deterministic language (IMP, NIMP)
Deterministic imperative programs
Deterministic imperative programs

A simple imperative language: IMP

**IMP** expressions

\[
expr ::= X \quad \text{(variable)} \\
| \ c \quad \text{(constant)} \\
| \diamond expr \quad \text{(unary operation)} \\
| \ expr \diamond expr \quad \text{(binary operation)}
\]

- variables in a fixed set \( X \in \mathbb{V} \)
- constants \( I \overset{\text{def}}{=} \mathbb{B} \cup \mathbb{Z} \):
  - booleans \( \mathbb{B} \overset{\text{def}}{=} \{ \text{true}, \text{false} \} \)
  - integers \( \mathbb{Z} \)
- operations \( \diamond \):
  - integer operations: +, −, ×, /, <, ≤
  - boolean operations: ¬, ∧, ∨
  - polymorphic operations: =, ≠
A simple imperative language: IMP

Statements

\[
\begin{align*}
\textit{stat} & ::= \quad \text{skip} \quad \text{(do nothing)} \\
& \quad | \quad X \leftarrow \text{expr} \quad \text{(assignment)} \\
& \quad | \quad \text{stat}; \text{stat} \quad \text{(sequence)} \\
& \quad | \quad \text{if} \ \text{expr} \ \text{then} \ \text{stat} \ \text{else} \ \text{stat} \quad \text{(conditional)} \\
& \quad | \quad \text{while} \ \text{expr} \ \text{do} \ \text{stat} \quad \text{(loop)}
\end{align*}
\]

(inspired from the presentation in [Benton96])
Expression semantics

\[ E[\text{expr}] : \mathcal{E} \rightarrow I \]

- environments \( \mathcal{E} \) defined as \( \mathcal{V} \rightarrow I \) map variables in \( \mathcal{V} \) to values in \( I \)

- \( E[\text{expr}] \) returns a value in \( I \)

\( \rightarrow \) denotes partial functions (as opposed to \( \rightarrow \))

necessary because some operations are undefined

- \( 1 + \text{true}, 1 \land 2 \) (type mismatch)
- \( 3/0 \) (invalid value)

- defined by structural induction on abstract syntax trees

(\textit{next slide})

when we use the notation \( X[y] \), \( y \) is a syntactic object; \( X \) serves to distinguish
between different semantic functions with different signatures, often varying with the
kind of syntactic object \( y \) (expression, statement, etc.);

\( X[y]z \) is the application of the function \( X[y] \) to the object \( z \)
Deterministic imperative programs

Expression semantics

\[ E[\text{expr}] : \mathcal{E} \rightarrow \mathbb{I} \]

\[
\begin{align*}
E[c] \rho & \overset{\text{def}}{=} c \in \mathbb{I} \\
E[V] \rho & \overset{\text{def}}{=} \rho(V) \in \mathbb{I} \\
E[-e] \rho & \overset{\text{def}}{=} -\nu \in \mathbb{Z} \quad \text{if } \nu = E[e] \rho \in \mathbb{Z} \\
E[\neg e] \rho & \overset{\text{def}}{=} \neg\nu \in \mathbb{B} \quad \text{if } \nu = E[e] \rho \in \mathbb{B} \\
E[e_1 + e_2] \rho & \overset{\text{def}}{=} v_1 + v_2 \in \mathbb{Z} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \\
E[e_1 - e_2] \rho & \overset{\text{def}}{=} v_1 - v_2 \in \mathbb{Z} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \\
E[e_1 \times e_2] \rho & \overset{\text{def}}{=} v_1 \times v_2 \in \mathbb{Z} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \\
E[e_1/e_2] \rho & \overset{\text{def}}{=} v_1/v_2 \in \mathbb{Z} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \setminus \{0\} \\
E[e_1 \land e_2] \rho & \overset{\text{def}}{=} v_1 \land v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{B}, v_2 = E[e_2] \rho \in \mathbb{B} \\
E[e_1 \lor e_2] \rho & \overset{\text{def}}{=} v_1 \lor v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{B}, v_2 = E[e_2] \rho \in \mathbb{B} \\
E[e_1 < e_2] \rho & \overset{\text{def}}{=} v_1 < v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \\
E[e_1 \leq e_2] \rho & \overset{\text{def}}{=} v_1 \leq v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{Z}, v_2 = E[e_2] \rho \in \mathbb{Z} \\
E[e_1 = e_2] \rho & \overset{\text{def}}{=} v_1 = v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{I}, v_2 = E[e_2] \rho \in \mathbb{I} \\
E[e_1 \neq e_2] \rho & \overset{\text{def}}{=} v_1 \neq v_2 \in \mathbb{B} \quad \text{if } v_1 = E[e_1] \rho \in \mathbb{I}, v_2 = E[e_2] \rho \in \mathbb{I} \\
\text{undefined otherwise}
\end{align*}
\]
Statement semantics

\[ S[\text{stat}] : \mathcal{E} \rightarrow \mathcal{E} \]

- maps an environment before the statement to an environment after the statement
- partial function due to
  - errors in expressions
  - non-termination
- also defined by structural induction
Statement semantics

\( S[ \text{stat} ] : \mathcal{E} \rightarrow \mathcal{E} \)

- **skip**: do nothing
  \[ S[ \text{skip} ] \rho \overset{\text{def}}{=} \rho \]

- **assignment**: evaluate expression and mutate environment
  \[ S[ X \leftarrow e ] \rho \overset{\text{def}}{=} \rho[X \mapsto v] \quad \text{if } E[ e ] \rho = v \]

- **sequence**: function composition
  \[ S[ s_1 ; s_2 ] \overset{\text{def}}{=} S[ s_2 ] \circ S[ s_1 ] \]

- **conditional**
  \[ S[ \text{if } e \text{ then } s_1 \text{ else } s_2 ] \rho \overset{\text{def}}{=} \begin{cases} S[ s_1 ] \rho & \text{if } E[ e ] \rho = \text{true} \\ S[ s_2 ] \rho & \text{if } E[ e ] \rho = \text{false} \\ \text{undefined} & \text{otherwise} \end{cases} \]

\[ f[x \mapsto y] \] denotes the function that maps \( x \) to \( y \), and any \( z \neq x \) to \( f(z) \)
How do we handle loops?

The semantics of loops must satisfy:

\[
S[\text{while } e \text{ do } s] \rho =
\begin{cases}
\rho & \text{if } E[e] \rho = \text{false} \\
S[\text{while } e \text{ do } s](S[s] \rho) & \text{if } E[e] \rho = \text{true} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

This is a recursive definition; we must prove that:

- the equation has solution(s);
- in case there are several solutions, there is a single “right” one;

\[\implies\text{ we use fixpoints of operators over partially ordered sets.}\]
Flat orders and partial functions

Flat ordering \((\perp, \sqsubseteq)\) on \(\mathbb{I}\):
- \(\perp \overset{\text{def}}{=} \mathbb{I} \cup \{\perp\}\) (pointed set)
- \(a \sqsubseteq b \overset{\text{def}}{\iff} a = \perp \lor a = b\) (partial order)
- every chain is finite, and so has a lub \(\sqcup\)
  \(\implies\) it is a pointed complete partial order (cpo)

\(\perp\) denotes the value “undefined” (\(\sqsubseteq\) is an information order)

Similarly for \(\mathcal{E}_\perp \overset{\text{def}}{=} \mathcal{E} \cup \{\perp\}\).

Note that \((\mathcal{E} \rightarrow \mathcal{E}) \simeq (\mathcal{E} \rightarrow \mathcal{E}_\perp)\)
\(\implies\) we will now use total functions only.
Polset of continuous partial functions

Partial order structure on partial functions \((\mathcal{E}_\perp \rightarrow \mathcal{E}_\perp, \sqsubseteq)\)

- \(\mathcal{E}_\perp \rightarrow \mathcal{E}_\perp\) extends \(\mathcal{E} \rightarrow \mathcal{E}_\perp\)
  - domain = co-domain \(\implies\) allows composition \(\circ\)
  - \(f \in \mathcal{E} \rightarrow \mathcal{E}_\perp\) extended with \(f(\perp) \overset{\text{def}}{=} \perp\) (strictness)
    \(\implies\) if \(S[\,s\,]\,x\) is undefined, so is \((S[\,s'\,] \circ S[\,s\,])x\)

  such functions are monotonic and continuous
  \(a \sqsubseteq b \implies f(a) \sqsubseteq f(b)\) and \(f(\sqcup X) = \sqcup \{ f(x) \mid x \in X \}\)

  \(\implies\) we restrict \(\mathcal{E}_\perp \rightarrow \mathcal{E}_\perp\) to continuous functions: \(\mathcal{E}_\perp \overset{\varepsilon}{\rightarrow} \mathcal{E}_\perp\)

- point-wise order \(\sqsubseteq\) on functions
  \(f \sqsubseteq g \overset{\text{def}}{\iff} \forall x: f(x) \sqsubseteq g(x)\)

- \(\mathcal{E}_\perp \overset{\varepsilon}{\rightarrow} \mathcal{E}_\perp\) has a least element: \(\perp \overset{\text{def}}{=} \lambda x.\perp\)

- by point-wise lub \(\sqcup\) of chains, it is also complete \(\implies\) a cpo
  \(\sqcup F = \lambda x.\, \sqcup \{ f(x) \mid f \in F \}\)
Fixpoint semantics of loops

To solve the semantic equation, we use a **fixpoint** of a functional.

We use actually the **least fixpoint**. (Most precise for the information order)

\[ S[\text{while } e \text{ do } s] \overset{\text{def}}{=} \text{lfp} \ F \]

where:

\[ F : (\mathcal{E}_{\bot} \to \mathcal{E}_{\bot}) \to (\mathcal{E}_{\bot} \to \mathcal{E}_{\bot}) \]

\[ F(f)(\rho) = \begin{cases} 
\rho & \text{if } E[e] \rho = \text{false} \\
\text{⊥} & \text{otherwise} \\
f(S[s] \rho) & \text{if } E[e] \rho = \text{true}
\end{cases} \]

**Theorem**

\( \text{lfp} \ F \) is well-defined

Remember our equation on \( S[\text{while } e \text{ do } s] \)?
It can be rewritten exactly as:

\[ S[\text{while } e \text{ do } s] = F(S[\text{while } e \text{ do } s]) \]
Recall **Kleene’s theorem:**

A continuous function on a cpo has a least fixpoint

To use the theorem we prove that $S[\text{stat}]$ is continuous (and is well-defined) by induction on the syntax of stat:

- **base cases:** $S[\text{skip}]$ and $S[\text{X} \leftarrow \text{e}]$ are continuous
- $S[\text{if e then } s_1 \text{ else } s_2]$: by induction hypothesis, as $S[s_1]$ and $S[s_2]$ are continuous
- $S[s_1; s_2]$: by induction hypotheses and because $\circ$ respects continuity
- $F$ is continuous in $(\mathcal{E}_\bot \xrightarrow{\varepsilon} \mathcal{E}_\bot) \xrightarrow{\varepsilon} (\mathcal{E}_\bot \xrightarrow{\varepsilon} \mathcal{E}_\bot)$ by induction hypotheses
  $\implies \text{lfp } F \text{ exists by Kleene’s theorem}$

moreover, lfp $F$ is continuous (simple consequence of Kleene’s proof)
$\implies S[\text{while e do s}]$ is continuous
Recall another fact about Kleene’s fixpoints:  
\[
\text{lfp } F = \bigcup_{n \in \mathbb{N}} F^n(\perp)
\]

- \(F^0(\perp) = \perp\) is completely undefined (no information)
- \(F^1(\perp)(\rho) = \begin{cases} 
\rho & \text{if } E[e] \rho = \text{false} \\
\perp & \text{otherwise}
\end{cases}\) environment if the loop is never entered (partial information)
- \(F^2(\perp)(\rho) = \begin{cases} 
\rho & \text{if } E[e] \rho = \text{false} \\
S[s][\rho] & \text{else if } E[e](S[s][\rho]) = \text{false} \\
\perp & \text{otherwise}
\end{cases}\) environment if the loop is iterated at most once
- \(F^n(\perp)(\rho)\) environment if the loop is iterated at most \(n - 1\) times
- \(\bigcup_{n \in \mathbb{N}} F^n(\perp)\) environment when exiting the loop whatever the number of iterations (total information)
Error vs. non-termination

In our semantics $S[\text{stat}] \rho = \bot$ can mean:

- either stat starting on input $\rho$ loops for ever
- or it stops prematurely with an error

Note: we could distinguish between the two cases by:

- adding an error value $\Omega$, distinct from $\bot$
- propagating it in the semantics, bypassing computations
  (no further computation after an error)
Rewriting the semantics using total functions on cpos with $\bot$:

- $E⟦\text{expr}⟧ : \mathcal{E}_\bot \xrightarrow{c} \mathcal{I}_\bot$
  returns $\bot$ for an error or if its argument is $\bot$

- $S⟦\text{stat}⟧ : \mathcal{E}_\bot \xrightarrow{c} \mathcal{E}_\bot$
  - $S⟦\text{skip}⟧ \rho \overset{\text{def}}{=} \rho$
  - $S⟦e_1; e_2⟧ \overset{\text{def}}{=} S⟦e_2⟧ \circ S⟦e_1⟧$
  - $S⟦X \leftarrow e⟧ \rho \overset{\text{def}}{=} \begin{cases} 
  \bot & \text{if } E⟦e⟧ \rho = \bot \\
  \rho[X \mapsto E⟦e⟧ \rho] & \text{otherwise}
  \end{cases}$
  - $S⟦\text{if } e \text{ then } s_1 \text{ else } s_2⟧ \rho \overset{\text{def}}{=} \begin{cases} 
  S⟦s_1⟧ \rho & \text{if } E⟦e⟧ \rho = \text{true} \\
  S⟦s_2⟧ \rho & \text{if } E⟦e⟧ \rho = \text{false} \\
  \bot & \text{otherwise}
  \end{cases}$
  - $S⟦\text{while } e \text{ do } s⟧ \overset{\text{def}}{=} \text{lfp } F$
  where $F(f)(\rho) = \begin{cases} 
  \rho & \text{if } E⟦e⟧ \rho = \text{false} \\
  f(S⟦s⟧ \rho) & \text{if } E⟦e⟧ \rho = \text{true} \\
  \bot & \text{otherwise}
  \end{cases}$
Why non-determinism?

It is useful to consider non-deterministic programs, to:

- model partially unknown environments
- abstract away unknown program parts
- abstract away too complex parts
- handle a set of programs as a single one

Kinds of non-determinism

- data non-determinism: \( expr ::= \text{random()} \)
- control non-determinism: \( stat ::= \text{either } s_1 \text{ or } s_2 \)
  but we can write “either \( s_1 \) or \( s_2 \)” as “if \( \text{random()} = 0 \) then \( s_1 \) else \( s_2 \)”

Consequence on semantics and verification

we want to verify all the possible executions
\[ \Rightarrow \] the semantics should express all the possible executions
We extend \textbf{IMP} to \textbf{NIMP}, an imperative language with non-determinism.

\textbf{NIMP} language

\[
expr ::= \ X \hspace{1cm} \text{(variable)}
\]
\[
| \quad c \hspace{1cm} \text{(constant)}
\]
\[
| \quad [c_1, c_2] \hspace{1cm} \text{(constant interval)}
\]
\[
| \quad \Diamond expr \hspace{1cm} \text{(unary operation)}
\]
\[
| \quad expr \Diamond expr \hspace{1cm} \text{(binary operation)}
\]

\textbf{NIMP} has the same statements as \textbf{IMP}

\[c_1 \in \mathbb{Z} \cup \{-\infty\}, \ c_2 \in \mathbb{Z} \cup \{+\infty\}\]

\([c_1, c_2]\) means: return a fresh random value between \(c_1\) and \(c_2\) each time the expression is evaluated

Question: is \([0, 1] = [0, 1]\) true or false?
Expression semantics

\[ E[\ expr ] : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{I}) \]

\[
\begin{align*}
E[\ V ] \rho & \overset{\text{def}}{=} \{\rho(V)\} \\
E[\ c ] \rho & \overset{\text{def}}{=} \{c\} \\
E[\ [c_1, c_2]] \rho & \overset{\text{def}}{=} \{c \in \mathbb{Z} | c_1 \leq c \leq c_2\} \\
E[\ -e ] \rho & \overset{\text{def}}{=} \{-v | v \in E[\ e ] \rho \cap \mathbb{Z}\} \\
E[\ -e ] \rho & \overset{\text{def}}{=} \{-v | v \in E[\ e ] \rho \cap \mathbb{B}\} \\
E[\ e_1 + e_2 ] \rho & \overset{\text{def}}{=} \{v_1 + v_2 | v_1 \in E[\ e_1 ] \rho \cap \mathbb{Z}, v_2 \in E[\ e_2 ] \rho \cap \mathbb{Z}\} \\
E[\ e_1/e_2 ] \rho & \overset{\text{def}}{=} \{v_1/v_2 | v_1 \in E[\ e_1 ] \rho \cap \mathbb{Z}, v_2 \in E[\ e_2 ] \rho \cap \mathbb{Z} \setminus \{0\}\} \\
E[\ e_1 < e_2 ] \rho & \overset{\text{def}}{=} \{\text{true} | \exists v_1 \in E[\ e_1 ] \rho, v_2 \in E[\ e_2 ] \rho : v_1 \in \mathbb{Z}, v_2 \in \mathbb{Z}, v_1 < v_2\} \cup \\
& \quad \{\text{false} | \exists v_1 \in E[\ e_1 ] \rho, v_2 \in E[\ e_2 ] \rho : v_1 \in \mathbb{Z}, v_2 \in \mathbb{Z}, v_1 \geq v_2\} \\
\end{align*}
\]

\[
\begin{itemize}
  \item we output a set of values, to account for non-determinism
  \item we can have \( E[\ e ] \rho = \emptyset \) due to errors
  \begin{itemize}
    \item (no need for a special \( \Omega \) nor \( \bot \) element)
  \end{itemize}
\end{itemize}
\]
Semantic domain:

- A statement can output a set of environments
  \[ \mathcal{E} \rightarrow \mathcal{P}(\mathcal{E}) \]

- To allow composition, extend it to
  \[ \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \]

- Non-termination and errors can be modeled by \( \emptyset \)
  (no need for a special \( \Omega \) nor \( \bot \) element)
Statement semantics

\[ S[ \text{stat} ] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \]

- **S[ skip ]** \( R \overset{\text{def}}{=} R \)
- **S[ s_1; s_2 ]** \( \overset{\text{def}}{=} S[ s_2 ] \circ S[ s_1 ] \)
- **S[ X \leftarrow e ]** \( R \overset{\text{def}}{=} \{ \rho[X \mapsto v] | \rho \in R, v \in E[e]_{\rho} \} \)
  - pick an environment \( \rho \)
  - pick an expression value \( v \) in \( E[e]_{\rho} \)
  - generate an updated environment \( \rho[X \mapsto v] \)
- **S[ if e then s_1 else s_2 ]** \( R \overset{\text{def}}{=} \)
  \[ S[ s_1 ] \{ \rho \in R | \text{true} \in E[e]_{\rho} \} \cup S[ s_2 ] \{ \rho \in R | \text{false} \in E[e]_{\rho} \} \]
  - filter environments according to the value of \( e \)
  - execute both branch independently
  - join them with \( \cup \)
Statement semantics

- \( S[\text{while } e \text{ do } s] \) \( R \) = \( \{ \rho \in \text{lfp } F \mid \text{false } \in E[e] \rho \} \)

where \( F(X) \) = \( R \cup S[s] \{ \rho \in X \mid \text{true } \in E[e] \rho \} \)

**Justification:** \( \text{lfp } F \) exists

- \((\mathcal{P}(E), \subseteq, \cup, \cap, \emptyset, E)\) forms a complete lattice

- all semantic functions and \( F \) are monotonic and continuous
  
in fact, they are strict complete join morphisms
  \( S[s] (\cup_{i \in \Delta} X_i) = \cup_{i \in \Delta} S[s] X_i \) and \( S[s] \emptyset = \emptyset \)
  
  which we write as \( S[s] \in \mathcal{P}(E) \xrightarrow{\cup} \mathcal{P}(E) \)

  it is really the *image function* of a function in \( E \to \mathcal{P}(E) \)

  \( S[s] X = \cup \{ S[s] \{x\} \mid x \in X \} \)

- we can apply both Kleene’s and Tarski’s fixpoint theorems
Join semantics of loops

\[ S[\text{while } e \text{ do } s] R \overset{\text{def}}{=} \{ \rho \in \text{lfp } F \mid \text{false} \in E[e] \rho \} \]

where \[ F(X) \overset{\text{def}}{=} R \cup S[ s ] \{ \rho \in X \mid \text{true} \in E[e] \rho \} \]

(F applies a loop iteration to \( X \) and adds back the environments \( R \) before the loop)

Recall that \( \text{lfp } F = \bigcup_{n \in \mathbb{N}} F^n(\emptyset) \)

- \( F^0(\emptyset) = \emptyset \)
- \( F^1(\emptyset) = R \)
  - environments before entering the loop
- \( F^2(\emptyset) = R \cup S[ s ] \{ \rho \in R \mid \text{true} \in E[e] \rho \} \)
  - environments after zero or one loop iteration
- \( F^n(\emptyset) : \text{environments after at most } n - 1 \text{ loop iterations} \)
  - (just before testing the condition to determine if we should iterate a \( n \)-th time)
- \( \bigcup_{n \in \mathbb{N}} F^n(\emptyset) : \text{loop invariant} \)
“Angelic” non-determinism and termination

If \textit{stat} is \textbf{deterministic} (no \([c_1, c_2]\) in expressions) the semantics is \textbf{equivalent} to our semantics on \(\mathcal{E}_\perp \xrightarrow{\_} \mathcal{E}_\perp\)

\textbf{Justification:} \(\{ E \subseteq \mathcal{E} \mid |E| \leq 1 \}, \subseteq, \cup, \emptyset \) is isomorphic to \(\mathcal{E}_\perp, \subseteq, \cup, \perp \)

In general, we can have several outputs for \(S[\textit{stat} ]\) \(\{\rho\} \subseteq \mathcal{E} \cup \{\Omega\}\):

- \(\emptyset\): the program never terminates at all
- \(\{\Omega\}\): the program never terminates correctly
- \(R \subseteq \mathcal{E} \setminus \{\Omega\}\): when the program terminates, it terminates correctly, in an environment in \(R\)

\(\implies\) we cannot express that a program always terminates!

This is called the “\textit{Angelic}” semantics, useful for \textit{partial correctness}. 
Note on non-determinism and termination

Other (more complex) ways to mix non-termination and non-determinism exist.

Based on distinguishing $\emptyset$ and $\perp$, and on different order relations $\sqsubseteq$.

This is a complex subject, we will say no more.

- **powerset order**
  - angelic semantics

- **mixed order**
  - natural semantics

- **Egli-Milner order**
  - natural semantics
Link between operational and denotational semantics
Motivation

Are the operational and denotational semantics consistent with each other?

Note that:

- systems are actually described \textit{operationally} (previous courses)
- the denotational semantics is a \textit{more abstract} representation (more suitable for some reasoning on the system)

$\Rightarrow$ the denotational semantics must be proven faithful (in some sense) to the operational model to be of any use
### Labelled syntax

\[ \ell \text{stat} \quad ::= \quad \ell \text{skip} \]

\[ \mid \ell X \leftarrow \text{expr} \]

\[ \mid \ell \text{if expr then stat else stat} \]

\[ \mid \ell \text{while expr do stat} \]

\[ \mid \ell \text{stat; stat} \]

\( \ell \in \mathcal{L} \): control labels

- statements are decorated with **unique control labels** \( \ell \in \mathcal{L} \)
- program configurations in \( \Sigma \overset{\text{def}}{=} \mathcal{L} \times \mathcal{E} \)  
  (lower-level than \( \mathcal{E} \): we must track program locations)
- transition relation \( \tau \subseteq \Sigma \times \Sigma \)  
  models atomic execution steps
Transition systems for our language

\( \tau \) is defined by induction on the syntax of statements

\((\sigma, \sigma') \in \tau\) is denoted as \(\sigma \rightarrow \sigma'\)

\[
\begin{align*}
\tau[\ell_1\text{skip}\ell_2] & \overset{\text{def}}{=} \{(\ell_1, \rho) \rightarrow (\ell_2, \rho) | \rho \in \mathcal{E}\} \\
\tau[\ell_1 X \leftarrow e\ell_2] & \overset{\text{def}}{=} \{(\ell_1, \rho) \rightarrow (\ell_2, \rho[X \mapsto v]) | \rho \in \mathcal{E}, v \in \mathcal{E}[e] \rho\} \\
\tau[\ell_1\text{if } e\text{ then } \ell_2 s_1 \text{ else } \ell_3 s_2 \ell_4] & \overset{\text{def}}{=} \\
& \{ (\ell_1, \rho) \rightarrow (\ell_2, \rho) | \rho \in \mathcal{E}, \text{true} \in \mathcal{E}[e] \rho \} \cup \\
& \{ (\ell_1, \rho) \rightarrow (\ell_3, \rho) | \rho \in \mathcal{E}, \text{false} \in \mathcal{E}[e] \rho \} \cup \\
& \tau[\ell_2 s_1 \ell_4] \cup \tau[\ell_3 s_2 \ell_4] \\
\tau[\ell_1\text{while } e\text{ do } \ell_2 s_1 \ell_3 s_2 \ell_4] & \overset{\text{def}}{=} \\
& \{ (\ell_1, \rho) \rightarrow (\ell_2, \rho) | \rho \in \mathcal{E} \} \cup \\
& \{ (\ell_2, \rho) \rightarrow (\ell_3, \rho) | \rho \in \mathcal{E}, \text{true} \in \mathcal{E}[e] \rho \} \cup \\
& \{ (\ell_2, \rho) \rightarrow (\ell_4, \rho) | \rho \in \mathcal{E}, \text{false} \in \mathcal{E}[e] \rho \} \cup \tau[\ell_3 s_2 \ell_2] \\
\end{align*}
\]

\[
\begin{align*}
\tau[\ell_1 s_1; \ell_2 s_2 s_3] & \overset{\text{def}}{=} \tau[\ell_1 s_1 s_2] \cup \tau[\ell_2 s_2 s_3] \\
\end{align*}
\]

Defines the small-step semantics of a statement

(the semantics of expressions is still in denotational form)
Special states

Given a labelled statement $\ell e s^\ell x$ and its transition system, we define:

- **initial states**: $I \overset{\text{def}}{=} \{ (\ell e, \rho) \mid \rho \in \mathcal{E} \}$
  - note that $\sigma \rightarrow \sigma' \implies \sigma' \notin I$

- **blocking states**: $B \overset{\text{def}}{=} \{ \sigma \in \Sigma \mid \forall \sigma' \in \Sigma, \sigma \not\rightarrow \sigma' \}$

- **correct termination**: $OK \overset{\text{def}}{=} \{ (\ell x, \rho) \mid \rho \in \mathcal{E} \}$
  - note that $OK \subseteq B$

- **error**: $ERR \overset{\text{def}}{=} B \cap \{ (\ell, \rho) \mid \ell \neq \ell x, \rho \in \mathcal{E} \}$

$B = ERR \cup OK$

$ERR \cap OK = \emptyset$
**Trace:** in $\Sigma^\infty$

- starting in an initial state $I$
- following transitions $\rightarrow$
- can only end in a blocking state $B$

i.e.: $t[s] = t[s]^* \cup t[s]^\omega$ where

- **finite traces:**
  
  $t[s]^* \overset{\text{def}}{=} \{ (\sigma_0, \ldots, \sigma_n) \mid n \geq 0, \sigma_0 \in I, \sigma_n \in B, \forall i < n : \sigma_i \rightarrow \sigma_{i+1} \}$

- **infinite traces:**

  $t[s]^\omega \overset{\text{def}}{=} \{ (\sigma_0, \ldots) \mid \sigma_0 \in I, \forall i \in \mathbb{N} : \sigma_i \rightarrow \sigma_{i+1} \}$
From traces to big-step semantics

**Big-step semantics:** abstraction of traces
only remembers the input-output relations

many variants exist:

- **“angelic”** semantics, in $\mathcal{P}(\Sigma \times \Sigma)$:
  
  $$A[s] \overset{\text{def}}{=} \{ (\sigma, \sigma') \mid \exists (\sigma_0, \ldots, \sigma_n) \in t[s]^* : \sigma = \sigma_0, \sigma' = \sigma_n \}$$

  (only give information on the terminating behaviors; can only prove partial correctness)

- **natural** semantics, in $\mathcal{P}(\Sigma \times \Sigma_{\bot})$:
  
  $$N[s] \overset{\text{def}}{=} A[s] \cup \{ (\sigma, \bot) \mid \exists (\sigma_0, \ldots) \in t[s]^{\omega} : \sigma = \sigma_0 \}$$

  (models the terminating and non-terminating behaviors; can prove total correctness)

**Exercise:** compute the semantics of “while $X > 0$ do $X \leftarrow X - [0, 1]$”
The angelic denotational and big-step semantics are isomorphic
(isomorphism between relations and strict complete join morphisms)

\[ S[s] = \alpha(A[s]) \]
where

- \( \alpha(X) \overset{\text{def}}{=} \lambda R. \{ \rho' | \rho \in R, ((\ell_e, \rho), (\ell_x, \rho')) \in X \} \) (image of a relation)
- \( \alpha^{-1}(Y) = \{ ((\ell_e, \rho), (\ell_x, \rho')) | \rho \in E, \rho' \in Y(\{\rho\}) \} \)

Proof idea: by induction on the syntax of \( s \)

\( \implies \) our operational and denotational semantics match

Also, the denotational semantics is an abstraction of the natural semantics
(it forgets about infinite computations)

Thesis

All semantics can be compared for equivalence or abstraction

this can be made formal in the abstract interpretation theory
(see [Cousot02])
Semantic diagram

- Link between operational and denotational semantics

**denotational world**
- \( S[s] \)
- denotational
- traces
- transition system (small step)
- statement

**operational world**
- \( A[s] \)
- big step
- natural
- \( N[s] \)

\( \alpha \)
Recall that traces can be expressed as fixpoints:

- $t^* = (\text{lfp } F) \cap (I \Sigma^\infty)$
  - $(\cap (I \Sigma^\infty) \text{ restricts to traces starting in } I)$
  - where $F(X) \overset{\text{def}}{=} B \cup \{(\sigma, \sigma_0, \ldots, \sigma_n) | \sigma \rightarrow \sigma_0 \land (\sigma_0, \ldots, \sigma_n) \in X\}$

- $t^\omega = (\text{gfp } F) \cap (I \Sigma^\infty)$
  - where $F(X) \overset{\text{def}}{=} \{(\sigma, \sigma_0, \ldots) | \sigma \rightarrow \sigma_0 \land (\sigma_0, \ldots) \in X\}$

This also holds for the angelic denotational semantics:

- $S = \alpha(\text{lfp } F)$
  - $(\alpha \text{ converts relations to functions})$
  - where $F(X) \overset{\text{def}}{=} (B \times B) \cup \{(\sigma, \sigma'') | \exists \sigma': \sigma \rightarrow \sigma' \land (\sigma', \sigma'') \in X\}$

and many others: natural, denotational, big-step, denotational, ...
Higher-order programs
Monomorphic typed higher order language

PCF language (introduced by Scott in 1969)

\[
\text{type} ::= \text{int} \quad \text{(integers)} \\
\quad | \quad \text{bool} \quad \text{(booleans)} \\
\quad | \quad \text{type} \rightarrow \text{type} \quad \text{(functions)}
\]

\[
\text{term} ::= \text{X} \quad \text{(variable X } \in \mathbb{V}) \\
\quad | \quad \text{c} \quad \text{(constant)} \\
\quad | \quad \lambda X^{\text{type}}.\text{term} \quad \text{(abstraction)} \\
\quad | \quad \text{term term} \quad \text{(application)} \\
\quad | \quad \text{Y}^{\text{type}} \text{term} \quad \text{(recursion)} \\
\quad | \quad \Omega^{\text{type}} \quad \text{(failure)}
\]

PCF (programming computable functions) is a \(\lambda\)-calculus with:

- a monomorphic type system (unlike ML)
- explicit type annotations \(X^{\text{type}}, \ Y^{\text{type}}, \ \Omega^{\text{type}}\) (unlike ML)
- an explicit recursion combiner \(Y\) (unlike untyped \(\lambda\)-calculus)
- constants, including \(\mathbb{Z}, \ \mathbb{B}\) and a few built-in functions (arithmetic and comparisons in \(\mathbb{Z}\), if-then-else, etc.)
What should be the domain of $T[\text{term}]$?

**Difficulty:** term contains heterogeneous objects: constants, functions, second order functions, etc.

**Solution:** use the type information

each term $m$ can be given a type $\text{typ}(m)$

use one semantic domain $D_t$ per type $t$

then $T[\text{term}] : \mathcal{E} \rightarrow D_{\text{typ}(m)}$ where $\mathcal{E} \overset{\text{def}}{=} \forall \rightarrow (\bigcup_{t \in \text{type}} D_t)$

Domain definition by induction on the syntax of types

- $D_{\text{int}} \overset{\text{def}}{=} \mathbb{Z}_\bot$
- $D_{\text{bool}} \overset{\text{def}}{=} \mathbb{B}_\bot$
- $D_{t_1 \rightarrow t_2} \overset{\text{def}}{=} (D_{t_1} \overset{c}{\rightarrow} D_{t_2})_\bot$
Order on semantic domains

**Order:** all domains are cpos

- $\mathcal{D}_{\text{int}} \overset{\text{def}}{=} \mathbb{Z}_\bot$, $\mathcal{D}_{\text{bool}} \overset{\text{def}}{=} \mathbb{B}_\bot$ use a flat ordering

- $\mathcal{D}_{t_1 \rightarrow t_2} \overset{\text{def}}{=} (\mathcal{D}_{t_1} \overset{c}{\rightarrow} \mathcal{D}_{t_2})_\bot$

with order $f \sqsubseteq g \iff f = \bot \lor (f, g \neq \bot \land \forall x: f(x) \sqsubseteq g(x))$

- $\mathcal{D}_{t_1} \overset{c}{\rightarrow} \mathcal{D}_{t_2}$ is ordered point-wise

- each domain has its fresh minimal $\bot$ element

  (to distinguish $\Omega_{\text{int} \rightarrow \text{int}}$ from $\lambda X.\Omega_{\text{int}}$)

- we restrict $\rightarrow$ to continuous functions

  (to be able to take fixpoints)

(see [Scott93])
Denotational semantics

**Environments:** $\mathcal{E} \defeq \forall \to (\bigcup_{t \in \text{type}} \mathcal{D}_t)$

**Semantics:** $T[m] : \mathcal{E} \to \mathcal{D}_{\text{typ}}(m)$

- $T[X] \rho \defeq \rho(X)$
- $T[c] \rho \defeq c$
- $T[\lambda X^t.m] \rho \defeq \lambda x. T[m] (\rho[X \mapsto x])$
- $T[m_1 m_2] \rho \defeq (T[m_1] \rho)(T[m_2] \rho)$
- $T[Y^t m] \rho \defeq \text{lfp } (T[m] \rho)$
- $T[\Omega^t] \rho \defeq \bot$

- program functions $\lambda$ are mapped to mathematical functions $\lambda$
- program recursion $Y$ is mapped to fixpoints lfp
- errors and non-termination are mapped to (typed) $\bot$
- we should prove that $T[m]$ is indeed continuous (by induction) so that lfp exists, and also that $T[m_1]$ is indeed a function (by soundness of typing)
Higher-order programs

Operational semantics

Operational semantics: based on the $\lambda$–calculus

- states are terms: $\Sigma \overset{\text{def}}{=} \text{term}$
- transitions correspond to reductions:
  
  \[
  (\lambda X^t.m_1) m_2 \rightarrow m_1[X \mapsto m_2] \quad (\lambda\text{–reduction})
  \]
  
  \[
  \Omega^t \rightarrow \Omega^t \quad (\text{failure})
  \]
  
  \[
  Y^t m \rightarrow m (Y^t m) \quad (\text{iteration})
  \]
  
  plus $c_1 c_2 \rightarrow (c_1 + c_2)$ \quad (arithmetic)
  
  if true $m_1 m_2 \rightarrow m_1$ \quad (if-then-else)
  
  if false $m_1 m_2 \rightarrow m_2$ \quad (if-then-else)
  
  \[
  \frac{m_1 \rightarrow m_1'}{m_1 m_2 \rightarrow m_1' m_2} \quad (\text{context rule})
  \]
  
  …

- big-step semantics $m \Downarrow$: maximal reductions
  
  $m \Downarrow = m' \overset{\text{def}}{\iff} m \rightarrow^* m' \land \forall m'': m' \rightarrow m''$

(PCF is deterministic)
Higher-order programs

Links between operational and denotational semantics

How do we check that operational and denotational semantics match?

check that they have the same view of “semantically equal programs”

- **denotational way**: we can use $T[m_1] = T[m_2]$
- we need an **operational way** to compare functions
  comparing the syntax is too fine grained,
  Example: $(\lambda X^{\text{int}}.0) \neq (\lambda X^{\text{int}}.\text{minus} \ 1 \ 1)$, but they have the same denotation

**Observational equivalence**: observe terms in all contexts

- contexts $c$: terms with holes $\square$
- $c[m]$ term obtained by substituting $m$ in hole
- $\text{ground}$ is the set of terms of type $\text{int}$ or $\text{bool}$
- term equivalence $\approx$:
  
  $m_1 \approx m_2 \iff (\forall c: c[m_1] \downarrow = c[m_2] \downarrow \text{ when } c[m_1] \in \text{ground})$

  (don’t look at a function’s syntax, force its full evaluation and look at the value result)
Full abstraction: \[ \forall m_1, m_2: m_1 \approx m_2 \iff T[m_1] = T[m_2] \]

Unexpected result: for PCF, \( \Leftarrow \) holds (adequacy), but not \( \Rightarrow \)!

(full abstraction concept introduced by Milner in 1975, proof by Plotkin 1977)

Compare with: IMP, NIMP are fully abstract
\[ \forall s_1, s_2 \in \text{stat}: S[s_1] = S[s_2] \iff \forall c: A[c[s_1]] = A[c[s_2]] \]

Intuitive explanation:
Domains such as \( \mathcal{D}_{t_1 \rightarrow t_2} \) contain many functions, most of them do not correspond to any program (this is expected: many functions are not computable).

The problem is that, if \( m_1, m_2 \) have the form \( \lambda X^{t_1 \rightarrow t_2}.m \), \( T[m_1] = T[m_2] \) imposes \( T[m_1] f = T[m_2] f \) for all \( f \in \mathcal{D}_{t_1 \rightarrow t_2} \), including many \( f \) that are not computable.

It is actually possible to construct \( m_1, m_2 \) where \( T[m_1] f \neq T[m_2] f \) only for some non-program functions \( f \), so that \( m_1 \approx m_2 \) actually holds.

Two solutions come to mind:
- enrich the language to express more functions in \( \mathcal{D}_{t_1 \rightarrow t_2} \) (next slide)
- restrict \( \mathcal{D}_{t_1 \rightarrow t_2} \) to contain less non-program objects

Fruitful but complex research topic...
**Example:** the parallel or function \( \text{por} \)

\[
\text{por}(a)(b) \overset{\text{def}}{=} \begin{cases} 
\text{true} & \text{if } a = \text{true} \lor b = \text{true} \\
\text{false} & \text{if } a = \text{false} \land b = \text{false} \\
\bot & \text{otherwise}
\end{cases}
\]

\( \text{por} \) can observe \( a \) and \( b \) concurrently, and return as soon as one returns true

compare with sequential \( \text{or} \), where \( \forall b: \text{or}(\bot)(b) = \bot \)

We have the following non-obvious result:

- \( \text{por} \) cannot be defined in \( \text{PCF} \)
  \((\text{por} \text{ is a parallel construct, } \text{PCF} \text{ is a sequential language})\)

- \( \text{PCF}+\text{por} \) is fully abstract

(see [Ong95], [Winskel97] for references on the subject)
Recursive domain equations
we can write truly polymorphic functions: e.g., $\lambda X.X$
(in PCF we would have to choose a type: $\text{int} \to \text{int}$ or $\text{bool} \to \text{bool}$ or $(\text{int} \to \text{int}) \to (\text{int} \to \text{int})$ or ...)

no need for a recursion combinator $Y$
(we can define $Y \overset{\text{def}}{=} \lambda F.(\lambda X.F (X X))(\lambda X.F (X X))$, not typable in PCF)

operational semantics based on reduction, similarly to PCF

denotational semantics also similar to PCF, but...
How to choose the domain of denotations $T[m]$?

- we need a unique domain $D$ for all terms
  
  \[(\text{no type information to help us})\]

- $\lambda X.X$ is a function
  
  \[\Rightarrow \text{it should have denotation in } (X \rightarrow Y)_\bot \text{ for some } X, Y \subseteq D\]

- $\lambda X.X$ is polymorphic; it accepts any term as argument
  
  \[\Rightarrow D \subseteq X, Y\]

We have a domain equation to solve:

\[D \simeq (\mathbb{Z} \cup \mathbb{B} \cup (D \rightarrow D))_\bot\]

**Problem:** no solution in set theory

\[(D \rightarrow D \text{ has a strictly larger cardinal than } D)\]
Inverse limits

Given a fixpoint domain equation $\mathcal{D} = F(\mathcal{D})$
we construct an infinite sequence of domains:

- $\mathcal{D}_0 \overset{\text{def}}{=} \{ \bot \}$
- $\mathcal{D}_{i+1} \overset{\text{def}}{=} F(\mathcal{D}_i)$

We require the existence of continuous retractions:

- $\gamma_i : \mathcal{D}_i \overset{c}{\rightarrow} \mathcal{D}_{i+1}$ (embedding)
- $\alpha_i : \mathcal{D}_{i+1} \overset{c}{\rightarrow} \mathcal{D}_i$ (projection)
- $\alpha_i \circ \gamma_i = \lambda x. x$ ($\mathcal{D}_i \simeq$ a subset of $\mathcal{D}_{i+1}$)
- $\gamma_i \circ \alpha_i \sqsubseteq \lambda x. x$ ($\mathcal{D}_{i+1}$ can be approximated by $\mathcal{D}_i$)

This is denoted: $\mathcal{D}_0 \overset{\alpha_0}{\leftarrow} \overset{\gamma_0}{\rightarrow} \mathcal{D}_1 \overset{\alpha_1}{\leftarrow} \overset{\gamma_1}{\rightarrow} \cdots$

**Inverse limit:** $\mathcal{D}_\infty \overset{\text{def}}{=} \{ (a_0, a_1, \ldots) | \forall i: a_i \in \mathcal{D}_i \land a_i = \alpha(a_{i+1}) \}$

(infinite sequences of elements; able to represent an element of any $\mathcal{D}_i$)
Inverse limits: \[ \mathcal{D}_\infty \overset{\text{def}}{=} \{ (a_0, a_1, \ldots) \mid \forall i: a_i \in \mathcal{D}_i \land a_i = \alpha(a_{i+1}) \} \]

**Theorem**

\( \mathcal{D}_\infty \) is a cpo and \( F(\mathcal{D}_\infty) \) is isomorphic to \( \mathcal{D}_\infty \)

**Application** to λ-calculus

If we restrict ourself to continuous functions, retractions can be computed for \( F(\mathcal{D}) \overset{\text{def}}{=} (\mathbb{Z} \cup \mathbb{B} \cup (\mathcal{D} \rightarrow \mathcal{D})) \perp \)

\( (\gamma_i(f)) \overset{\text{def}}{=} \lambda x. f \)

\( \alpha_i(x) \overset{\text{def}}{=} x \) if \( x \in \mathbb{Z} \cup \mathbb{B} \cup \{ \perp \} \) and \( \alpha_i(f) \overset{\text{def}}{=} f(\perp) \) if \( f \in \mathcal{D}_i \rightarrow \mathcal{D}_i \)

\( \implies \) we found our semantic domain!

(pioneered by [Scott-Strachey71], see [Abramsky-Jung94] for a reference)
Restrictions of function spaces

The restriction to continuous functions seems merely technical but there are some valid justifications:

- all the denotations in IMP, NIMP, PCF were continuous \\
  \((this\ appeared\ naturally,\ not\ as\ an\ a\ priori\ restriction)\)

- intuitively, computable functions should at least be \textbf{monotonic}
  recall that \(\sqsubseteq\) is an information order
  a function cannot give a more precise result with less information
  e.g.: if \(f(a) = \perp\) for some \(a \neq \perp\), then \(f(\perp) = \perp\)

- \textbf{continuity} is also reasonable
  given a problem on an infinite data set \(S\)
  computers can only process finite parts \(S_i\) of \(S\)
  continuity ensures that the solution of \(S\) is contained in that of all \(S_i\)
  e.g.: if \(0 \sqsubseteq 1 \sqsubseteq \cdots \sqsubseteq \omega\) and \(\forall i < \omega: f(i) = 0\), then \(f(\omega)\) should also be 0
Solution domains of recursive equations can also give the semantics of a variety of inductive or polymorphic data-types

Examples:

- **integer lists:**
  \[ D = (\{\text{empty}\} \cup (\mathbb{Z} \times D)) \perp \]

- **pairs:**
  \[ D = (\mathbb{Z} \cup (D \times D)) \perp \]
  (allows arbitrary nested pairs, and also contains trees and lists)

- **records:**
  \[ D = (\mathbb{Z} \cup (\mathbb{N} \rightarrow D)) \perp \]
  (fields are named by integer position)

- **sum types:**
  \[ D = (\mathbb{Z} \cup (\{1\} \times D) \cup (\{2\} \times D)) \perp \]
  (we “tag” each case of the sum with an integer)
Courses and references on denotational semantics:


Research articles and surveys:


