Denotational semantics
Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2015–2016

Course 4
4 March 2016
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
| [] -> []
| x::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 $D \simeq \text{memory} \rightarrow \text{memory}$  
  
  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* $D_a \rightarrow D_b$  
  
  challenge: choose $D$ 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 \( \lambda \)-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
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 \( \mathbb{I} \overset{\text{def}}{=} \mathbb{B} \cup \mathbb{Z} \):
  - boolean operations: \( \neg, \land, \lor \)
  - integers \( \mathbb{Z} \)
- integer operations: \( +, -, \times, \div, <, \leq \)
- boolean operations: \( \neg, \land, \lor \)
- polymorphic operations: \( =, \neq \)
Deterministic imperative programs

A simple imperative language: IMP

Statements

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

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

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

- environments \( \mathcal{E} \overset{\text{def}}{=} \mathcal{V} \rightarrow I \) map variables in \( \mathcal{V} \) to values in \( I \)
- \( E[expr] \) returns a value in \( I \)
- \( \overset{\_}{\_} \) denotes partial functions (as opposed to \( \rightarrow \))
  necessary because some operations are undefined
  - \( 1 + \text{true}, 1 \land 2 \)
  - \( 3/0 \)
- defined by structural induction on abstract syntax trees
  \((next\ slide)\)

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

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

- \[ E[\text{c}] \rho \quad \text{def} \quad c \quad \in \mathbb{I} \]
- \[ E[\text{V}] \rho \quad \text{def} \quad \rho(\text{V}) \quad \in \mathbb{I} \]
- \[ E[\neg e] \rho \quad \text{def} \quad \neg v \quad \in \mathbb{Z} \quad \text{if} \ v = E[\text{e}] \rho \in \mathbb{Z} \]
- \[ E[\neg e] \rho \quad \text{def} \quad \neg v \quad \in \mathbb{B} \quad \text{if} \ v = E[\text{e}] \rho \in \mathbb{B} \]
- \[ E[\text{e}_1 + \text{e}_2] \rho \quad \text{def} \quad v_1 + v_2 \quad \in \mathbb{Z} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \]
- \[ E[\text{e}_1 - \text{e}_2] \rho \quad \text{def} \quad v_1 - v_2 \quad \in \mathbb{Z} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \]
- \[ E[\text{e}_1 \times \text{e}_2] \rho \quad \text{def} \quad v_1 \times v_2 \quad \in \mathbb{Z} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \]
- \[ E[\text{e}_1 / \text{e}_2] \rho \quad \text{def} \quad \frac{v_1}{v_2} \quad \in \mathbb{Z} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \setminus \{0\} \]
- \[ E[\text{e}_1 \land \text{e}_2] \rho \quad \text{def} \quad v_1 \land v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{B}, v_2 = E[\text{e}_2] \rho \in \mathbb{B} \]
- \[ E[\text{e}_1 \lor \text{e}_2] \rho \quad \text{def} \quad v_1 \lor v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{B}, v_2 = E[\text{e}_2] \rho \in \mathbb{B} \]
- \[ E[\text{e}_1 < \text{e}_2] \rho \quad \text{def} \quad v_1 < v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \]
- \[ E[\text{e}_1 \leq \text{e}_2] \rho \quad \text{def} \quad v_1 \leq v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{Z}, v_2 = E[\text{e}_2] \rho \in \mathbb{Z} \]
- \[ E[\text{e}_1 = \text{e}_2] \rho \quad \text{def} \quad v_1 = v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{I}, v_2 = E[\text{e}_2] \rho \in \mathbb{I} \]
- \[ E[\text{e}_1 \neq \text{e}_2] \rho \quad \text{def} \quad v_1 \neq v_2 \quad \in \mathbb{B} \quad \text{if} \ v_1 = E[\text{e}_1] \rho \in \mathbb{I}, v_2 = E[\text{e}_2] \rho \in \mathbb{I} \]

undefined otherwise
Statement semantics

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

- 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 \( f(z) \)
Statement semantics: loops

How do we handle loops?

The semantics of loops must satisfy:

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

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;

\[\Rightarrow\] we use fixpoints of operators over partially ordered sets.
Flat ordering \((\perp, \sqsubseteq)\) on \(\mathbb{I}\):

\[\perp \overset{\text{def}}{=} \mathbb{I} \cup \{\perp\}\]

\[a \sqsubseteq b \iff a = \perp \lor a = b\]

- Every chain is finite, and so has a \(\sqcup\) (lub)
- \(\Rightarrow\) it is a pointed complete partial order (cpo)

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

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

Note that \((E \rightarrow E) \simeq (E \rightarrow E_{\perp})\)
\(\Rightarrow\) we will now use total functions only.
Deterministic imperative programs

Poset 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 \(S\lbrack s\rbrack x\) is undefined, so is \((S\lbrack s'\rbrack \circ S\lbrack s\rbrack) x\)

  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: \(\mathcal{E}_\perp \overset{\xi}{\rightarrow} \mathcal{E}_\perp\)

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

- \(\mathcal{E}_\perp \overset{\xi}{\rightarrow} \mathcal{E}_\perp\) 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
  
  \[\sqcup F = \lambda x. \sqcup \{ f(x) \mid f \in F \}\]
To solve the semantic equation, we use a fixpoint of a functional. We use actually the least fixpoint. (most precise for the information order)

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

where:

\[ F : (E_\bot \rightarrow E_\bot) \rightarrow (E_\bot \rightarrow E_\bot) \]

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

**Theorem**

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

rember our equation on \( S[\textbf{while } e \textbf{ do } s] \)? it can be rewritten exactly as: \( S[\textbf{while } e \textbf{ do } s] = F(S[\textbf{while } e \textbf{ do } s]) \)
Recall **Kleene’s theorem:**

**Kleene’s theorem**

A continuous function on a cpo has a least fixpoint

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

- **base cases:** $S\llbracket \text{skip} \rrbracket$ and $S\llbracket X \leftarrow e \rrbracket$ are continuous
- $S\llbracket \text{if } e \text{ then } s_1 \text{ else } s_2 \rrbracket$: by induction hypothesis, as $S\llbracket s_1 \rrbracket$ and $S\llbracket s_2 \rrbracket$ are continuous
- $S\llbracket s_1; s_2 \rrbracket$: by induction hypotheses and because $\circ$ respects continuity
- $F$ is continuous in $(\mathcal{E}_\bot \xrightarrow{\mathcal{E}_\bot} \mathcal{E}_\bot) \xrightarrow{\mathcal{E}_\bot} (\mathcal{E}_\bot \xrightarrow{\mathcal{E}_\bot})$ by induction hypotheses
  $\implies \text{lfp } F$ exists by Kleene’s theorem
  moreover, $\text{lfp } F$ is continuous (simple consequence of Kleene’s proof)
  $\implies S\llbracket \text{while } e \text{ do } s \rrbracket$ is continuous
Join semantics of loops

Recall another fact about Kleene’s fixpoints: \( \text{lfp } F = \bigcup_{n \in \mathbb{N}} F^n(\bot) \)

- \( F^0(\bot) = \bot \) is completely undefined (no information)
- \( F^1(\bot)(\rho) = \begin{cases} \rho & \text{if } E[e] \rho = \text{false} \\ \bot & \text{otherwise} \end{cases} \)
  environment if the loop is never entered (partial information)
- \( F^2(\bot)(\rho) = \begin{cases} \rho & \text{if } E[e] \rho = \text{false} \\ S[s] \rho & \text{else if } E[e](S[s] \rho) = \text{false} \\ \bot & \text{otherwise} \end{cases} \)
  environment if the loop is iterated at most once
- \( F^n(\bot)(\rho) \)
  environment if the loop is iterated at most \( n - 1 \) times
- \( \bigcup_{n \in \mathbb{N}} F^n(\bot) \)
  environment when exiting the loop whatever the number of iterations (total information)
In our semantics $S[\text{stat}] \rho = \bot$ can mean:

- either $\text{stat}$ 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} \)
Non-determinism
Why non-determinism?

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

- model partially unknown environments (user input)
- abstract away unknown program parts (libraries)
- abstract away too complex parts (rounding errors in floats)
- handle a set of programs as a single one (parametric programs)

Kinds of non-determinism

- data non-determinism: `expr ::= random()`
- control non-determinism: `stat ::= either s_1 or s_2`

  but we can write “either s_1 or s_2” as “if random() = 0 then s_1 else s_2”

Consequence on semantics and verification

we want to verify all the possible executions

⇒ the semantics should express all the possible executions
We extend IMP to NIMP, an imperative language with non-determinism.

**NIMP language**

\[
\text{expr} ::= \begin{align*}
&X & \quad \text{(variable)} \\
&c & \quad \text{(constant)} \\
&[c_1, c_2] & \quad \text{(constant interval)} \\
&\Diamond \text{expr} & \quad \text{(unary operation)} \\
&\text{expr} \Diamond \text{expr} & \quad \text{(binary operation)}
\end{align*}
\]

**NIMP** has the same statements as **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?
Non-determinism

Expression semantics

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

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

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

- a statement can output a set of environments
  $\implies$ use $\mathcal{E} \to \mathcal{P}(\mathcal{E})$

- to allow composition, extend it to $\mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

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

Statement semantics

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

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

Statement semantics

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

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

- \((P(E), \subseteq, \cup, \cap, \emptyset, E)\) 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 \( S[s] \in P(E) \overset{\cup}{\rightarrow} P(E) \)

  it is really the image function of a function in \( E \rightarrow P(E) \)

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

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

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

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

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

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

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

If `stat` is deterministic (no `[c_1, c_2]` in expressions)
the semantics is equivalent to our semantics on \( \mathcal{E}_⊥ → \mathcal{E}_⊥ \)

Justification: \( (\{ E \subseteq \mathcal{E} \mid |E| \leq 1 \}, \subseteq, \cup, \emptyset) \) is isomorphic to \( (\mathcal{E}_⊥, \sqsubseteq, \sqcup, \bot) \)

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

- \( \emptyset \): 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 “Angelico” semantics, useful for 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 $\bot$, and on different order relations $\sqsubseteq$

(powerset order, angelic semantics)

(mixed order, natural semantics)

(Egli-Milner order, natural semantics)

(this is a complex subject, we will say no more)
Link between operational and denotational semantics
Motivation

Are the operational and denotational semantics consistent with each other?

Note that:

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

⇒ the denotational semantics must be proven faithful (in some sense) to the operational model to be of any use
transition systems for our non-deterministic language

labelled syntax

\[ \ell \text{stat} \ ::=  \begin{cases}  
\ell \text{skip} \\
\ell \ X \leftarrow \text{expr} \\
\ell \text{if expr then stat else stat} \\
\ell \text{while expr do stat} \\
\ell \text{stat; stat} 
\end{cases} \]

\( \ell \in \mathcal{L} \): 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
Link between operational and denotational semantics

Transition systems for our language

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

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

\[
\begin{align*}
\tau[\ell_1 \text{skip} \ell_2] & \overset{\text{def}}{=} \{(\ell_1, \rho) \rightarrow (\ell_2, \rho) \mid \rho \in E\} \\
\tau[\ell_1 X \leftarrow e \ell_2] & \overset{\text{def}}{=} \{(\ell_1, \rho) \rightarrow (\ell_2, \rho[X \mapsto v]) \mid \rho \in E, \ v \in E[e] \rho\} \\
\tau[\ell_1 \text{if } e \text{ then } \ell_2 s_1 \text{ else } \ell_3 s_2 \ell_4] & \overset{\text{def}}{=} \\
& \{ (\ell_1, \rho) \rightarrow (\ell_2, \rho) \mid \rho \in E, \ \text{true} \in E[e] \rho \} \cup \\
& \{ (\ell_1, \rho) \rightarrow (\ell_3, \rho) \mid \rho \in E, \ \text{false} \in E[e] \rho \} \cup \\
& \tau[\ell_2 s_1 \ell_4] \cup \tau[\ell_3 s_2 \ell_4] \\
\tau[\ell_1 \text{while } e \text{ do } \ell_3 s_4] & \overset{\text{def}}{=} \\
& \{ (\ell_1, \rho) \rightarrow (\ell_2, \rho) \mid \rho \in E \} \cup \\
& \{ (\ell_2, \rho) \rightarrow (\ell_3, \rho) \mid \rho \in E, \ \text{true} \in E[e] \rho \} \cup \\
& \{ (\ell_2, \rho) \rightarrow (\ell_4, \rho) \mid \rho \in E, \ \text{false} \in E[e] \rho \} \cup \tau[\ell_3 s_4] \\
\tau[\ell_1 s_1; \ell_2 s_2 \ell_3] & \overset{\text{def}}{=} \tau[\ell_1 s_1 \ell_2] \cup \tau[\ell_2 s_2 \ell_3]
\end{align*}
\]

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:** \( I \overset{\text{def}}{=} \{ (\ell_e, \rho) \mid \rho \in E \} \)
  
  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:** \( OK \overset{\text{def}}{=} \{ (\ell_x, \rho) \mid \rho \in E \} \)
    
    note that \( OK \subseteq B \)

  - **Error:** \( ERR \overset{\text{def}}{=} B \cap \{ (\ell, \rho) \mid \ell \neq \ell_x, \rho \in E \} \)

  \[ B = ERR \cup OK \]

  \[ ERR \cap OK = \emptyset \]
Reminder: maximal trace semantics

**Trace:** in $\Sigma^\infty$ (finite or infinite sequence of states)
- starting in an initial state $I$
- following transitions $\rightarrow$
- can only end in a blocking state $B$ (traces are maximal)

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

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

- **infinite traces:**
  $t[s]^{\omega} \overset{\text{def}}{=} \{ (\sigma_0, \ldots) | \sigma_0 \in I, \forall i \in \mathbb{N}: \sigma_i \rightarrow \sigma_{i+1} \}$
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 \( \mathcal{P}(\Sigma \times \Sigma_\perp) \):
  \[
  N[s] \overset{\text{def}}{=} A[s] \cup \{ (\sigma, \perp) \mid \exists (\sigma_0, \ldots) \in t[s]^{\omega} : \sigma = \sigma_0 \}
  \]
  (models the terminating and non-terminating behaviors; can prove total correctness)

Exercise: compute the semantics of “\textbf{while } X > 0 \textbf{ 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

1. \[ \alpha(X) \overset{\text{def}}{=} \lambda R.\{ \rho' \mid \rho \in R, ((\ell_e, \rho), (\ell_x, \rho')) \in X \} \] (image of a relation)

2. \[ \alpha^{-1}(Y) = \{ ((\ell_e, \rho), (\ell_x, \rho')) \mid \rho \in E, \rho' \in Y(\{\rho\}) \} \]

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

\[ \implies \text{our operational and denotational semantics match} \]

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])
Link between operational and denotational semantics

Semantic diagram

\[
\begin{align*}
&\text{denotational world} \\
&\text{transition system (small step)} \\
&\text{statement} \\
&\alpha \\
&S[ s ] \quad \text{denotational} \\
&\rightarrow \\
&\text{natural} \\
&N[ s ] \\
&\text{traces} \\
&t[ s ] \\
&\text{big step} \\
&A[ s ]
\end{align*}
\]}
Recall that traces can be expressed as fixpoints:

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

- \( t[s]^{\omega} = (\text{gfp } F) \cap (\text{I}\Sigma^\infty) \) 
  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:

- \( S[s] = \alpha(\text{lfp } F) \) 
  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,\ldots

**Thesis**

All semantics can be expressed through fixpoints

(again [Cousot02])
Higher-order programs
### PCF language (introduced by Scott in 1969)

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td>integers</td>
<td></td>
</tr>
<tr>
<td><code>bool</code></td>
<td>booleans</td>
<td></td>
</tr>
<tr>
<td><code>type → type</code></td>
<td>functions</td>
<td></td>
</tr>
<tr>
<td><code>term</code></td>
<td>X</td>
<td>(variable X ∈ V)</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>(constant)</td>
</tr>
<tr>
<td></td>
<td>λX^{type}.term</td>
<td>(abstraction)</td>
</tr>
<tr>
<td></td>
<td>term term</td>
<td>(application)</td>
</tr>
<tr>
<td></td>
<td>Y^{type} term</td>
<td>(recursion)</td>
</tr>
<tr>
<td></td>
<td>Ω^{type}</td>
<td>(failure)</td>
</tr>
</tbody>
</table>

**PCF** (programming computable functions) is a λ-calculus with:

- a monomorphic type system
  
- explicit type annotations X^{type}, Y^{type}, Ω^{type}
  
- an explicit recursion combiner Y
  
- constants, including Z, B and a few built-in functions
  (arithmetic and comparisons in 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 $\mathcal{E} \overset{\text{def}}{=} \mathbb{V} \rightarrow (\bigcup_{t \in \text{type}} D_t)$

Domain definition by induction on the syntax of types

- $D_{\text{int}} \overset{\text{def}}{=} \mathbb{Z}_\perp$
- $D_{\text{bool}} \overset{\text{def}}{=} \mathbb{B}_\perp$
- $D_{t_1 \rightarrow t_2} \overset{\text{def}}{=} (D_{t_1} \overset{c}{\rightarrow} D_{t_2})_\perp$
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 $\lambda X_{\text{int}}.\Omega_{\text{int}}$

- we restrict $\rightarrow$ to continuous functions
  (to be able to take fixpoints)

(see [Scott93])
**Environments:** \( \mathcal{E} \overset{\text{def}}{=} \forall \rightarrow (\bigcup_{t \in \text{type}} \mathcal{D}_t) \)

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

\[
\begin{align*}
T[ X ] \rho & \overset{\text{def}}{=} \rho(X) \\
T[ c ] \rho & \overset{\text{def}}{=} c \\
T[ \lambda X^t.m ] \rho & \overset{\text{def}}{=} \lambda x. T[ m ] (\rho[X \mapsto x]) \\
T[ m_1 \ m_2 ] \rho & \overset{\text{def}}{=} (T[ m_1 ] \rho)(T[ m_2 ] \rho) \\
T[ Y^t \ m ] \rho & \overset{\text{def}}{=} \text{lfp} (T[ m ] \rho) \\
T[ \Omega^t ] \rho & \overset{\text{def}}{=} \bot^t
\end{align*}
\]

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

- states are terms: $\Sigma \overset{\text{def}}{=} \text{term}$
- transitions correspond to reductions:
  
  $$(\lambda X^t.m_1) \; m_2 \rightarrow m_1[X \mapsto m_2] \quad (\lambda\text{-reduction})$$

  $$\Omega^t \rightarrow \Omega^t \quad \text{(failure)}$$

  $$Y^t \; m \rightarrow m \; (Y^t \; m) \quad \text{(iteration)}$$

  $\text{plus} \; c_1 \; c_2 \rightarrow (c_1 + c_2) \quad \text{(arithmetic)}$

  $\text{if} \; \text{true} \; m_1 \; m_2 \rightarrow m_1 \quad \text{(if-then-else)}$

  $\text{if} \; \text{false} \; m_1 \; m_2 \rightarrow m_2 \quad \text{(if-then-else)}$

  $$\frac{m_1 \rightarrow m'_1}{m_1 \; m_2 \rightarrow m'_1 \; m_2} \quad \text{(context rule)}$$

- big-step semantics $m \Downarrow$: maximal reductions
  
  $$m \Downarrow = m' \overset{\text{def}}{=} m \rightarrow^* m' \land \not\exists m'': m' \rightarrow m''$$

  ($\text{PCF is deterministic}$)
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 $\approx$:
  $m_1 \approx m_2 \overset{\text{def}}{\iff} (\forall c: c[m_1] \Downarrow = c[m_2] \Downarrow \text{ when } c[m_1] \in \text{ground})$
  (don’t look at a function’s syntax, force its full evaluation and look at the value result)
Full abstraction: \( \forall m_1, m_2 : m_1 \approx m_2 \iff T\llbracket m_1 \rrbracket = T\llbracket m_2 \rrbracket \)

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
\( \forall s_1, s_2 \in \text{stat}: S\llbracket s_1 \rrbracket = S\llbracket s_2 \rrbracket \iff \forall c: A\llbracket c[s_1] \rrbracket = A\llbracket c[s_2] \rrbracket \)

Intuitive explanation:
Domains such as \( D_{t_1 \to t_2} \) 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 \( \lambda X^{t_1 \to t_2}.m \), \( T\llbracket m_1 \rrbracket = T\llbracket m_2 \rrbracket \) imposes \( T\llbracket m_1 \rrbracket f = T\llbracket m_2 \rrbracket f \) for all \( f \in D_{t_1 \to t_2} \), including many \( f \) that are not computable.
It is actually possible to construct \( m_1, m_2 \) where \( T\llbracket m_1 \rrbracket f \neq T\llbracket m_2 \rrbracket f \) 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 \( D_{t_1 \to t_2} \) (next slide)
- restrict \( D_{t_1 \to t_2} \) to contain less non-program objects

Fruitful but complex research topic…
Higher-order programs

Full abstraction

**Example:** the parallel or function *por*

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

*por* can observe *a* and *b* concurrently, and return as soon as one returns true.

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

We have the following non-obvious result:

- *por* cannot be defined in **PCF**
  
  (*por* is a parallel construct, **PCF** is a sequential language)

- **PCF**+*por* is fully abstract

(see [Ong95], [Winskel97] for references on the subject)
Recursive domain equations
recursion equations

Untyped higher order language

\[ \lambda \text{–calculus (with arithmetic)} \]

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

- 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 . . .
Recursive domain equations

How to choose the domain of denotations $T\left[ m \right]$?

- We need a unique domain $\mathcal{D}$ for all terms
  
  (no type information to help us)

- $\lambda X.X$ is a function
  
  $\implies$ it should have denotation in $(\mathcal{X} \to \mathcal{Y})_\perp$ for some $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{D}$

- $\lambda X.X$ is polymorphic; it accepts any term as argument
  
  $\implies \mathcal{D} \subseteq \mathcal{X}, \mathcal{Y}$

We have a domain equation to solve:

$$\mathcal{D} \simeq (\mathbb{Z} \cup \mathbb{B} \cup (\mathcal{D} \to \mathcal{D}))_\perp$$

**Problem:** no solution in set theory

($\mathcal{D} \to \mathcal{D}$ has a strictly larger cardinal than $\mathcal{D}$)
Inverse limits

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

- $D_0 \overset{\text{def}}{=} \{ \bot \}$
- $D_{i+1} \overset{\text{def}}{=} F(D_i)$

We require the existence of continuous retractions:

- $\gamma_i : D_i \xrightarrow{c} D_{i+1}$ (embedding)
- $\alpha_i : D_{i+1} \xrightarrow{c} D_i$ (projection)
- $\alpha_i \circ \gamma_i = \lambda x. x$ ($D_i \simeq$ a subset of $D_{i+1}$)
- $\gamma_i \circ \alpha_i \sqsubseteq \lambda x. x$ ($D_{i+1}$ can be approximated by $D_i$)

This is denoted: $D_0 \xrightarrow{\alpha_0} D_1 \xrightarrow{\alpha_1} \cdots$

**Inverse limit:** $D_\infty \overset{\text{def}}{=} \{(a_0, a_1, \ldots) \mid \forall i : a_i \in D_i \land a_i = \alpha(a_{i+1}) \}$

(infinite sequences of elements; able to represent an element of any $D_i$)
**Inverse limits**

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

**Theorem**

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

**Application** to \( \lambda \)-calculus

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

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

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

\( \Rightarrow \) 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 monotonic
  
  Recall that $\sqsubseteq$ is an information order
  
  A function cannot give a more precise result with less information
  
  E.g.: if $f(a) = \bot$ for some $a \neq \bot$, then $f(\bot) = \bot$

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

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

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

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


Research articles and surveys:


