Denotational Semantics

Ioannis Markakis

Lectures for Part II CST 2025/2026

Practicalities

- My mail: im496@cam.ac.uk.
- Do not hesitate to ask questions!
- Feel free to give me feedback at any point!

Introduction

What is this course about?

- Formal methods: mathematical tools for the specification, development, analysis and verification of software and hardware systems.

What is this course about?

- **Formal methods**: mathematical tools for the specification, development, analysis and verification of software and hardware systems.
- **Programming language theory**: design, implementation, tooling and reasoning for/about programming languages.

What is this course about?

- **Formal methods**: mathematical tools for the specification, development, analysis and verification of software and hardware systems.
- **Programming language theory**: design, implementation, tooling and reasoning for/about programming languages.
- **Programming language semantics**: what is the (mathematical) meaning of a program?

What is this course about?

- Formal methods: mathematical tools for the specification, development, analysis and verification of software and hardware systems.
- Programming language theory: design, implementation, tooling and reasoning for/about programming languages.
- Programming language semantics: what is the (mathematical) meaning of a program?

Goal: give an **abstract** and **compositional** (mathematical) model of programs.

Why study semantics?

- **Insight**: exposes the mathematical "essence" of programming language ideas.

Why study semantics?

- Insight: exposes the mathematical “essence” of programming language ideas.
- Documentation: precise but intuitive, machine-independent specification.

Why study semantics?

- **Insight**: exposes the mathematical "essence" of programming language ideas.
- **Documentation**: precise but intuitive, machine-independent specification.
- **Language design**: feedback from semantics (functional programming, monads & handlers, linearity...).

Why study semantics?

- Insight: exposes the mathematical “essence” of programming language ideas.
- Documentation: precise but intuitive, machine-independent specification.
- Language design: feedback from semantics (functional programming, monads & handlers, linearity...).
- Rigour: powerful way to justify formal methods.

Styles of formal semantics

- Operational
- Axiomatic
- Denotational

Styles of formal semantics

- **Operational**: meaning of a program in terms of the *steps of computation* it takes during execution (see Part IB Semantics).
- **Axiomatic**
- **Denotational**

Styles of formal semantics

- **Operational**: meaning of a program in terms of the *steps of computation* it takes during execution (see Part IB Semantics).
- **Axiomatic**: meaning of a program in terms of a *program logic* to reason about it (see Part II Hoare Logic & Model Checking).
- Denotational

Styles of formal semantics

- **Operational**: meaning of a program in terms of the *steps of computation* it takes during execution (see Part IB Semantics).
- **Axiomatic**: meaning of a program in terms of a *program logic* to reason about it (see Part II Hoare Logic & Model Checking).
- **Denotational**: meaning of a program defined abstractly as object of some suitable *mathematical structure* (see this course).

Denotational semantics in a nutshell

$$
\begin{array}{rcl}
\text{Syntax} & \xrightarrow{[\![-]\!]} & \text{Semantics} \\
\text{Program } P & \mapsto & \text{Denotation } [\![P]\!] \\
\\
\text{Arithmetic expression} & \mapsto & \text{Number} \\
\text{Boolean circuit} & \mapsto & \text{Boolean function} \\
\text{Recursive program} & \mapsto & \text{Partial recursive function} \\
 & \cdots &
\end{array}
$$

Denotational semantics in a nutshell

$$\text{Syntax} \xrightarrow{[\![-]\!]} \text{Semantics}$$

$$\text{Program } P \mapsto \text{Denotation } [\![P]\!]$$

Arithmetic expression $\mapsto$ Number

Boolean circuit $\mapsto$ Boolean function

Recursive program $\mapsto$ Partial recursive function

$\cdots$

Type $\mapsto$ Domain

Program $\mapsto$ Continuous functions between domains

Properties of denotational semantics

Abstraction

- mathematical object, implementation/machine independent;
- captures the concept of a programming language construct;
- should relate to practical implementations, though...

Properties of denotational semantics

Abstraction

- mathematical object, implementation/machine independent;
- captures the concept of a programming language construct;
- should relate to practical implementations, though...

Compositionality

- The denotation of a whole is defined using the *denotations* of its parts;
- $[\![P]\!]$ represents the contribution of P to *any* program containing P;
- More flexible and expressive than whole-program semantics.

Introduction

A basic example

Imp syntax

Programs

$$C \in \mathbf{Prog} ::= \mathtt{skip} \mid L := A \mid C;C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ B\ \mathtt{do}\ C$$

Imp syntax

Programs

ranges over a set $\mathbb{L}$ of *locations*

$$C \in \mathbf{Prog} ::= \texttt{skip} \mid L := A \mid C; C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ B\ \texttt{do}\ C$$

Imp syntax

Arithmetic expressions

$$A \in \mathbf{Aexp} ::= \underline{n} \mid L \mid A + A \mid \ldots$$

Programs

$$C \in \mathbf{Prog} ::= \texttt{skip} \mid L := A \mid C; C \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid \texttt{while } B \texttt{ do } C$$

ranges over *integers*

Arithmetic expressions

$$A \in \mathbf{Aexp} ::= \underline{n} \mid L \mid A + A \mid \ldots$$

Programs

$$C \in \mathbf{Prog} ::= \texttt{skip} \mid L := A \mid C;C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ B\ \texttt{do}\ C$$

Arithmetic expressions

$$A \in \mathbf{Aexp} ::= \underline{n} \mid L \mid A + A \mid \ldots$$

Boolean expressions

$$B \in \mathbf{Bexp} ::= \mathtt{true} \mid \mathtt{false} \mid A = A \mid \neg B \mid \ldots$$

Programs

$$C \in \mathbf{Prog} ::= \mathtt{skip} \mid L := A \mid C;C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ B\ \mathtt{do}\ C$$

Denotation functions – naïvely

$$\mathcal{A}: \quad \mathbf{Aexp} \to \mathbb{Z}$$

where

$$\mathbb{Z} \quad = \quad \{\dots, -1, 0, 1, \dots\}$$

Denotation functions – naïvely

$$\begin{aligned} \mathcal{A} : &\quad \mathbf{Aexp} \to \mathbb{Z} \\ \mathcal{B} : &\quad \mathbf{Bexp} \to \mathbb{B} \end{aligned}$$

where

$$\begin{aligned} \mathbb{Z} &= \{\ldots, -1, 0, 1, \ldots\} \\ \mathbb{B} &= \{\text{true}, \text{false}\} \end{aligned}$$

Arithmetic expressions?

$$\mathcal{A}[\![\underline{n}]\!] \quad = \quad n$$

$$\mathcal{A}[\![A_1 + A_2]\!] \quad = \quad \mathcal{A}[\![A_1]\!] + \mathcal{A}[\![A_2]\!]$$

Arithmetic expressions?

$$\begin{aligned}
\mathcal{A}[\![\underline{n}]\!] &= n \\
\mathcal{A}[\![A_1 + A_2]\!] &= \mathcal{A}[\![A_1]\!] + \mathcal{A}[\![A_2]\!] \\
\mathcal{A}[\![L]\!] &= ???
\end{aligned}$$

Denotation functions – less naïvely

$$\text{State} = (\mathbb{L} \to \mathbb{Z})$$

Denotation functions – less naïvely

$$\text{State} = (\mathbb{L} \to \mathbb{Z})$$

$$\begin{aligned} \mathcal{A} &: \textbf{Aexp} \to (\text{State} \to \mathbb{Z}) \\ \mathcal{B} &: \textbf{Bexp} \to (\text{State} \to \mathbb{B}) \end{aligned}$$

where

$$\begin{aligned} \mathbb{Z} &= \{\dots, -1, 0, 1, \dots\} \\ \mathbb{B} &= \{\text{true}, \text{false}\}. \end{aligned}$$

Denotation functions – less naïvely

$$\text{State} = (\mathbb{L} \to \mathbb{Z})$$

$$\begin{aligned}
\mathcal{A} &: \mathbf{Aexp} \to (\text{State} \to \mathbb{Z}) \\
\mathcal{B} &: \mathbf{Bexp} \to (\text{State} \to \mathbb{B}) \\
\mathcal{C} &: \mathbf{Prog} \to (\text{State} \to \text{State})
\end{aligned}$$

where

$$\begin{aligned}
\mathbb{Z} &= \{\ldots, -1, 0, 1, \ldots\} \\
\mathbb{B} &= \{\text{true}, \text{false}\}.
\end{aligned}$$

Semantics of arithmetic expressions

$$\mathcal{A}[\![\underline{n}]\!] \quad = \quad \lambda s \in \text{State} \,.\, n$$

$$\mathcal{A}[\![A_1 + A_2]\!] \quad = \quad \lambda s \in \text{State} \,.\, \mathcal{A}[\![A_1]\!](s) + \mathcal{A}[\![A_2]\!](s)$$

Semantics of arithmetic expressions

$$
\begin{aligned}
\mathcal{A}[\![\underline{n}]\!] &= \lambda s \in \mathrm{State}\,.\, n \\
\mathcal{A}[\![A_1 + A_2]\!] &= \lambda s \in \mathrm{State}\,.\, \mathcal{A}[\![A_1]\!](s) + \mathcal{A}[\![A_2]\!](s) \\
\mathcal{A}[\![L]\!] &= \lambda s \in \mathrm{State}.\, s(L)
\end{aligned}
$$

Semantics of boolean expressions

$$\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!] &= \lambda s \in \text{State}.\ \text{true} \\
\mathcal{B}[\![\texttt{false}]\!] &= \lambda s \in \text{State}.\ \text{false} \\
\mathcal{B}[\![A_1 = A_2]\!] &= \lambda s \in \text{State}.\ \text{eq}\,(\mathcal{A}[\![A_1]\!](s), \mathcal{A}[\![A_2]\!](s))
\end{aligned}$$

where $\text{eq}\,(a, a') = \begin{cases} \text{true} & \text{if } a = a' \\ \text{false} & \text{if } a \neq a' \end{cases}$

$$\mathcal{C}[\![\texttt{skip}]\!] \quad = \quad \lambda s \in \text{State}.\ s$$

Semantics of programs

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \lambda s \in \mathrm{State}.\, s \\
\mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C']\!] &= \lambda s \in \mathrm{State}.\, \mathrm{if}\,(\mathcal{B}[\![B]\!](s), \mathcal{C}[\![C]\!](s), \mathcal{C}[\![C']\!](s))
\end{aligned}
$$

$$
\text{where } \mathrm{if}\,(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases}
$$

$$\mathcal{C}[\![\texttt{skip}]\!] = \lambda s \in \mathrm{State}.\ s$$

This is compositionality!

$$\mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C']\!] = \lambda s \in \mathrm{State}.\ \mathrm{if}\,(\mathcal{B}[\![B]\!](s), \mathcal{C}[\![C]\!](s), \mathcal{C}[\![C']\!](s))$$

where $\mathrm{if}\,(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases}$

Semantics of programs

$$\mathcal{C}[\![\texttt{skip}]\!] = \lambda s \in \mathrm{State}.\, s$$

$$\mathcal{C}[\![\texttt{if } B \texttt{ then } C \texttt{ else } C']\!] = \lambda s \in \mathrm{State}.\, \mathrm{if}\,(\mathcal{B}[\![B]\!](s), \mathcal{C}[\![C]\!](s), \mathcal{C}[\![C']\!](s))$$

$$\text{where } \mathrm{if}\,(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases}$$

$$\mathcal{C}[\![L := A]\!] = \lambda s \in \mathrm{State}.\, s[L \mapsto \mathcal{A}[\![A]\!](s)]$$

$$\text{where } s[L \mapsto n](L') = \begin{cases} n & \text{if } L' = L \\ s(L) & \text{otherwise} \end{cases}$$

$$
\begin{array}{rcl}
\mathcal{C}[\![\texttt{skip}]\!] & = & \lambda s \in \mathrm{State}.\ s \\[2ex]
\mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C']\!] & = & \lambda s \in \mathrm{State}.\ \mathrm{if}\,(\mathcal{B}[\![B]\!](s), \mathcal{C}[\![C]\!](s), \mathcal{C}[\![C']\!](s)) \\[2ex]
 & & \text{where}\ \ \mathrm{if}\,(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases} \\[2ex]
\mathcal{C}[\![L := A]\!] & = & \lambda s \in \mathrm{State}.\ s[L \mapsto \mathcal{A}[\![A]\!](s)] \\[2ex]
 & & \text{where}\ s[L \mapsto n](L') = \begin{cases} n & \text{if } L' = L \\ s(L) & \text{otherwise} \end{cases} \\[2ex]
\mathcal{C}[\![C; C']\!] & = & \mathcal{C}[\![C']\!] \circ \mathcal{C}[\![C]\!] = \lambda s \in \mathrm{State}.\ \mathcal{C}[\![C']\!]\,(\mathcal{C}[\![C]\!](s))
\end{array}
$$

Introduction

A semantics for loops

Semantics of loops?

This is all very nice, but...

$$[\![\texttt{while}\ B\ \texttt{do}\ C]\!] = ???$$

Semantics of loops?

This is all very nice, but...

$$[\![\texttt{while } B \texttt{ do } C]\!] = ???$$

Remember:

- $\langle \texttt{while } B \texttt{ do } C, s\rangle \rightsquigarrow \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, s\rangle$
- we want **compositional** semantics: $[\![\texttt{while } B \texttt{ do } C]\!]$ in terms of $[\![C]\!]$ and $[\![B]\!]$
- we want denotational semantics **compatible** with the operational semantics

Loop as a fixpoint

$$\begin{aligned}
[\![\texttt{while } B \texttt{ do } C]\!] &= [\![\texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}]\!] \\
&= \lambda s \in \mathrm{State}.\, \mathrm{if}\,([\![B]\!](s), ([\![\texttt{while } B \texttt{ do } C]\!] \circ [\![C]\!])(s), s)
\end{aligned}$$

Loop as a fixpoint

$$
\begin{aligned}
[\![\texttt{while}\ B\ \texttt{do}\ C]\!] &= [\![\texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}]\!] \\
&= \lambda s \in \mathrm{State}.\, \mathrm{if}\, ([\![B]\!](s), ([\![\texttt{while}\ B\ \texttt{do}\ C]\!] \circ [\![C]\!])(s), s)
\end{aligned}
$$

We don't have a direct definition for $[\![\texttt{while}\ B\ \texttt{do}\ C]\!]$, but a fixed point equation!

$$
[\![\texttt{while}\ B\ \texttt{do}\ C]\!] = F_{[\![B]\!],[\![C]\!]}([\![\texttt{while}\ B\ \texttt{do}\ C]\!])
$$

where

$$
\begin{aligned}
F_{b,c} :\quad (\mathrm{State} \to \mathrm{State}) &\quad\to\quad (\mathrm{State} \to \mathrm{State}) \\
w &\quad\mapsto\quad \lambda s \in \mathrm{State}.\ \mathrm{if}\, (b(s), (w \circ c)(s), s)
\end{aligned}
$$

- Why/when does $w = F_{b,c}(w)$ have a solution?
- What if it has several solutions? Which one should be our $[\![\texttt{while}\ B\ \texttt{do}\ C]\!]$?

Introduction

A taste of domain theory

Total functions are not enough

Forget about State for a second, consider these equations ($f \in \mathbb{Z} \to \mathbb{Z}$) :

$$f(x) = f(x) + 1 \tag{1}$$

$$f(x) = f(x) \tag{2}$$

What about their fixed points?

Total functions are not enough

Forget about State for a second, consider these equations ($f \in \mathbb{Z} \to \mathbb{Z}$):

$$f(x) = f(x) + 1 \tag{1}$$

$$f(x) = f(x) \tag{2}$$

What about their fixed points?

- No function satisfies equation (1)!
- All functions satisfy equation (2)!

Partial functions to the rescue

Both functions should diverge!

Partial functions to the rescue

Both functions should diverge!

New rule: We may use partial functions $f \in \mathbb{Z} \rightharpoonup \mathbb{Z}$

Both functions should diverge!

New rule: We may use partial functions $f \in \mathbb{Z} \rightharpoonup \mathbb{Z}$

$$f(x) = f(x) + 1$$

has a unique solution: the function $\bot$ that is everywhere undefined

Both functions should diverge!

New rule: We may use partial functions $f \in \mathbb{Z} \rightharpoonup \mathbb{Z}$

$$f(x) = f(x) + 1$$

has a unique solution: the function $\bot$ that is everywhere undefined

But

$$f(x) = f(x)$$

has even more solutions now - all partial functions. Which one should we pick?

'Information order' on partial functions

Partial order on $\mathbb{Z} \rightharpoonup \mathbb{Z}$:

$w \sqsubseteq w'$ iff for all $s \in \mathbb{Z}$, if w is defined at s, so is w' and moreover $w(s) = w'(s)$
iff the graph of w is included in the graph of w'

'Information order' on partial functions

Partial order on $\mathbb{Z} \rightharpoonup \mathbb{Z}$:

$w \sqsubseteq w'$ iff for all $s \in \mathbb{Z}$, if w is defined at s, so is w' and moreover $w(s) = w'(s)$
iff the graph of w is included in the graph of w'

Least element $\bot \in \mathbb{Z} \rightharpoonup \mathbb{Z}$:

$\bot$ = totally undefined partial function

Partial order on $\mathbb{Z} \rightharpoonup \mathbb{Z}$:

$w \sqsubseteq w'$	iff	for all $s \in \mathbb{Z}$, if w is defined at s, so is w' and moreover $w(s) = w'(s)$
	iff	the graph of w is included in the graph of w'

Least element $\bot \in \mathbb{Z} \rightharpoonup \mathbb{Z}$:

$\bot$ = totally undefined partial function

$\bot$ is the **least** solution to $f(x) = f(x)$ making it a 'canonical' choice.

Back to loops (An example)

$$\mathcal{C} : \mathbf{Prog} \to (\mathrm{State} \rightharpoonup \mathrm{State}) \qquad \mathrm{State} = \{X, Y\} \to \mathbb{Z}$$

Back to loops (An example)

$$\mathcal{C} : \textbf{Prog} \to (\text{State} \rightharpoonup \text{State}) \qquad\qquad \text{State} = \{X, Y\} \to \mathbb{Z}$$

$$[\![\texttt{while}\ X > 0\ \texttt{do}\ (Y := X * Y; X := X - 1)]\!]$$

Back to loops (An example)

$$\mathcal{C} : \mathbf{Prog} \to (\mathrm{State} \rightharpoonup \mathrm{State}) \qquad \mathrm{State} = \{X, Y\} \to \mathbb{Z}$$

$$[\![\texttt{while}\ X > 0\ \texttt{do}\ (Y := X * Y; X := X - 1)]\!]$$

should be some w such that:

$$w = F_{[\![X>0]\!],[\![Y:=X*Y;X:=X-1]\!]}(w).$$

Back to loops (An example)

$$\mathcal{C} : \textbf{Prog} \to (\text{State} \rightharpoonup \text{State}) \qquad\qquad \text{State} = \{X, Y\} \to \mathbb{Z}$$

$$[\![\texttt{while}\ X > 0\ \texttt{do}\ (Y := X * Y; X := X - 1)]\!]$$

should be some w such that:

$$w = F_{[\![X>0]\!],[\![Y:=X*Y;X:=X-1]\!]}(w).$$

That is, we are looking for a fixed point of the following function F:

$$\begin{aligned} F : (\text{State} \rightharpoonup \text{State}) &\to (\text{State} \rightharpoonup \text{State}) \\ w &\mapsto \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w\,([X \mapsto x-1, Y \mapsto x \cdot y]) & \text{if } x > 0\end{cases}\end{aligned}$$

Approximating the least fixed point

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w\left([X \mapsto x-1, Y \mapsto x \cdot y]\right) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

Approximating the least fixed point

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w\left([X \mapsto x-1, Y \mapsto x \cdot y]\right) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_1[X \mapsto x, Y \mapsto y] = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ \text{undefined} & \text{if } x \geq 1\end{cases}$$

Approximating the least fixed point

$$F(w) = \lambda[X \mapsto x, Y \mapsto y]. \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w([X \mapsto x-1, Y \mapsto x \cdot y]) & \text{if } x > 0 \end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_2[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto y] & \text{if } x = 1 \\ \text{undefined} & \text{if } x \geq 2 \end{cases}$$

Approximating the least fixed point

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ w([X \mapsto x-1, Y \mapsto x \cdot y]) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_3[X \mapsto x, Y \mapsto y] = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ [X \mapsto 0, Y \mapsto y] & \text{if } x = 1\\ [X \mapsto 0, Y \mapsto 2y] & \text{if } x = 2\\ \text{undefined} & \text{if } x \geq 3\end{cases}$$

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ w\,([X \mapsto x-1, Y \mapsto x \cdot y]) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_n[X \mapsto x, Y \mapsto y] = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } 0 < x < n\\ \text{undefined} & \text{if } x \geq n\end{cases}$$

$$w_0 \sqsubseteq w_1 \sqsubseteq \ldots \sqsubseteq w_n \sqsubseteq \ldots$$

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w\left([X \mapsto x-1, Y \mapsto x \cdot y]\right) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_n[X \mapsto x, Y \mapsto y] = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } 0 < x < n \\ \text{undefined} & \text{if } x \geq n\end{cases}$$

$$w_0 \sqsubseteq w_1 \sqsubseteq \ldots \sqsubseteq w_n \sqsubseteq \ldots \sqsubseteq w_\infty?$$

Approximating the least fixed point

$$F(w) = \lambda[X \mapsto x, Y \mapsto y].\begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ w\left([X \mapsto x-1, Y \mapsto x \cdot y]\right) & \text{if } x > 0\end{cases}$$

Define recursively $w_n = F^n(w)$, that is $w_0 = \bot$ and $w_{n+1} = F(w_n)$.

$$w_n[X \mapsto x, Y \mapsto y] = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } 0 < x < n\\ \text{undefined} & \text{if } x \geq n\end{cases}$$

$$w_0 \sqsubseteq w_1 \sqsubseteq \ldots \sqsubseteq w_n \sqsubseteq \ldots \sqsubseteq w_\infty$$

$$w_\infty[X \mapsto x, Y \mapsto y] = \bigsqcup_{n \in \mathbb{N}} w_n = \begin{cases}[X \mapsto x, Y \mapsto y] & \text{if } x \leq 0\\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } x > 0\end{cases}$$

We have our semantics

$$F(w_\infty)[X \mapsto x, Y \mapsto y]$$

We have our semantics

$$F(w_\infty)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w_\infty[X \mapsto x - 1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} \qquad \text{(definition of } F\text{)}$$

We have our semantics

$$F(w_\infty)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w_\infty[X \mapsto x-1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} \qquad \text{(definition of } F\text{)}$$

$$= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto 1 \cdot y] & \text{if } x = 1 \\ [X \mapsto 0, Y \mapsto (x-1)! \cdot x \cdot y] & \text{if } x > 0 \end{cases} \qquad \text{(definition of } w_\infty\text{)}$$

We have our semantics

$$
\begin{aligned}
F(w_\infty)[X \mapsto x, Y \mapsto y] &= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w_\infty[X \mapsto x-1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} && \text{(definition of } F\text{)} \\
&= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto 1 \cdot y] & \text{if } x = 1 \\ [X \mapsto 0, Y \mapsto (x-1)! \cdot x \cdot y] & \text{if } x > 0 \end{cases} && \text{(definition of } w_\infty\text{)} \\
&= w_\infty[X \mapsto x, Y \mapsto y]
\end{aligned}
$$

We have our semantics

$$
\begin{aligned}
F(w_\infty)[X \mapsto x, Y \mapsto y] &= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w_\infty[X \mapsto x-1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} && \text{(definition of } F\text{)} \\
&= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto 1 \cdot y] & \text{if } x = 1 \\ [X \mapsto 0, Y \mapsto (x-1)! \cdot x \cdot y] & \text{if } x > 0 \end{cases} && \text{(definition of } w_\infty\text{)} \\
&= w_\infty[X \mapsto x, Y \mapsto y]
\end{aligned}
$$

We have our semantics

$$F(w_\infty)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w_\infty[X \mapsto x-1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} \quad \text{(definition of } F\text{)}$$

$$= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto 1 \cdot y] & \text{if } x = 1 \\ [X \mapsto 0, Y \mapsto (x-1)! \cdot x \cdot y] & \text{if } x > 0 \end{cases} \quad \text{(definition of } w_\infty\text{)}$$

$$= w_\infty[X \mapsto x, Y \mapsto y]$$

- $F(w_\infty) = w_\infty$ *i.e.* w_∞ is a fixed point of F;
- It is the least fixed point;
- Using w_∞ as the denotation of while is compatible with the operational semantics!

$$[\![\texttt{while } X > 0 \texttt{ do } (Y := X * Y; X := X - 1)]\!] = w_\infty$$

The rest of this course

The course can be roughly divided into two parts:

I: domain theory

II: denotational semantics for the language PCF