Denotational Semantics

Meven Lennon-Bertrand

Lectures for Part II CST 2024/2025

Practicalities

- My mail: mgapb2@cam.ac.uk. Do not hesitate to ask questions!
- Course notes will be updated, keep an eye on the course webpage.

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?

- Insight: exposes the mathematical “essence” of programming language ideas.

Why?

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

Why?

- **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?

- **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

- **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).

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

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
	...	
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 *denotation* 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

$\boxed{\text{Programs}}$

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

Programs

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

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

Arithmetic expressions

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

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

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

Imp syntax

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

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

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

where

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

Denotation functions – naïvely

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

where

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

Arithmetic expressions?

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

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

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

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

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

where

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

Semantics of arithmetic expressions

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

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

Semantics of boolean expressions

$$
\begin{array}{rcl}
\mathcal{B}[\![\texttt{true}]\!] & = & \lambda s \in \mathrm{State}.\ \mathrm{true} \\[1em]
\mathcal{B}[\![\texttt{false}]\!] & = & \lambda s \in \mathrm{State}.\ \mathrm{false} \\[1em]
\mathcal{B}[\![A_1 = A_2]\!] & = & \lambda s \in \mathrm{State}.\ \mathrm{eq}\,(\mathcal{A}[\![A_1]\!]\,(s), \mathcal{A}[\![A_2]\!]\,(s)) \\
 & & \text{where } \mathrm{eq}(a, a') = \begin{cases} \mathrm{true} & \text{if } a = a' \\ \mathrm{false} & \text{if } a \neq a' \end{cases}
\end{array}
$$

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

Semantics of programs

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \lambda s \in \mathrm{State}.\ s \\
\mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C']\!] &= \lambda s \in \mathrm{State}.\ \mathrm{if}\left(\mathcal{B}[\![B]\!](s), \mathcal{C}[\![C]\!](s), \mathcal{C}[\![C']\!](s)\right) \\
&\quad \text{where } \mathrm{if}(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases}
\end{aligned}
$$

Semantics of programs

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \lambda s \in \mathrm{State}.\, s \\
\mathcal{C}[\![\texttt{if } B \texttt{ then } C \texttt{ else } C']\!] &= \lambda s \in \mathrm{State}.\, \mathrm{if}\,(\mathcal{B}[\![B]\!]\,(s), \mathcal{C}[\![C]\!]\,(s), \mathcal{C}[\![C']\!]\,(s)) \\
&\quad \text{where } \mathrm{if}(b, x, x') = \begin{cases} x & \text{if } b = \mathrm{true} \\ x' & \text{if } b = \mathrm{false} \end{cases}
\end{aligned}
$$

This is compositionality!

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

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

Introduction

A semantics for loops

This is all very nice, but...

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

Semantics of loops?

This is all very nice, but...

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

Remember:

- $(\texttt{while}\ B\ \texttt{do}\ C, s) \rightsquigarrow (\texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}, s)$
- we want a *compositional* semantic: $[\![\texttt{while}\ B\ \texttt{do}\ C]\!]$ in terms of $[\![C]\!]$ and $[\![B]\!]$

Loop as a fixpoint

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

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

Not 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 $$
\begin{array}{rrcl}
F_{b,c}: & (\mathrm{State} \to \mathrm{State}) & \to & (\mathrm{State} \to \mathrm{State}) \\
& w & \mapsto & \lambda s \in \mathrm{State}.\ \mathrm{if}(b(s), w \circ c(s), s).
\end{array}
$$

Now we have a goal

- 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

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 Eq. (1)!
- All functions satisfy Eq. (2)!

Both functions should diverge!

Partial functions to the rescue

Both functions should diverge!

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

Both functions should diverge!

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

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

has a unique solution: the nowhere-defined function $\perp$

Both functions should diverge!

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

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

has a unique solution: the nowhere-defined function $\perp$

But

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

Has even more solutions now...

An order on partial functions

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

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

An order on partial functions

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

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

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

$\perp$ = totally undefined partial function

An order on partial functions

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

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

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

$\perp$ = totally undefined partial function

$\perp$ is the least solution to $f(x) = f(x)$, making it “canonical”.

Back to loops

$$\mathcal{C} : \mathbf{Prog} \to (\mathrm{State} \rightharpoonup \mathrm{State})$$

$$\mathcal{C} : \mathbf{Prog} \to (\mathrm{State} \rightharpoonup \mathrm{State})$$

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

$$\mathcal{C} : \mathbf{Prog} \to (\mathrm{State} \rightharpoonup \mathrm{State})$$

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

should be some w such that:

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

$$\mathcal{C} : \mathbf{Prog} \to (\text{State} \rightharpoonup \text{State})$$

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

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 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

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

Approximating the least fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

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

Approximating the least fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

$$w_2[X \mapsto x, Y \mapsto y] = F(w_1)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto y] & \text{if } x = 1 \\ \text{undefined} & \text{if } x \geq 2 \end{cases}$$

Approximating the least fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

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

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 &= \bot \\ w_{n+1} &= F(w_n) \end{cases}$.

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

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

Approximating the least fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

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

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

Approximating the least fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$.

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

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

$$w_\infty[X \mapsto x, Y \mapsto y] = \bigsqcup_{i \in \mathbb{N}} w_i = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x < 0 \\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } x \ge 0 \end{cases}$$

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

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

We have our semantics

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

We have our semantics

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

- $F(w_\infty) = w_\infty$ *i.e.* w_∞ is a fixed point of F;
- actually, the least fixed point;
- which agrees with the operational semantics (!)

The rest of this course

Part I domain theory → building mathematical tools

Part II denotational semantics for PCF