Denotational Semantics

Tyng–Ruey Chuang

Institute of Information Science
Academia Sinica, Taiwan

2010 Formosan Summer School
on Logic, Language, and Computation
June 28 – July 9, 2010

This course note . . .

- . . . is prepared for the *2010 Formosan Summer School on Logic, Language, and Computation (FLOLAC)* held in Taipei, Taiwan,
- . . . is made available from the FLOLAC '10 web site:

  `http://flolac.iis.sinica.edu.tw/flolac10/`

  (please also check the above site for updated version)
- . . . and is released to the public under a Creative Commons Attribution-ShareAlike 3.0 Taiwan license:

  `http://creativecommons.org/licenses/by-sa/3.0/tw/`

Course outline

Unit 1. Basic domain theory.

Unit 2. Denotational semantics of functional programs and **While** programs.

Unit 3. Non-standard semantics.

Each unit consists of 2 hours of lecture and 1 hour of lab/tutor.

Syntax and Semantics

- Syntax is about the form of sentences in a language.

Syntax and Semantics

- Syntax is about the form of sentences in a language.
- Semantics is about the meaning of sentences.

Syntax and Semantics

- Syntax is about the form of sentences in a language.
- Semantics is about the meaning of sentences.
- Syntax: `Let's keep in touch!`

Syntax and Semantics

- Syntax is about the form of sentences in a language.
- Semantics is about the meaning of sentences.
- Syntax: `Let's keep in touch!`
- Semantics: *Bye bye!*

Syntax and Semantics

- Syntax is about the form of sentences in a language.
- Semantics is about the meaning of sentences.
- Syntax: `Let's keep in touch!`
- Semantics: *Bye bye!*
- Syntax:

```
let f n = n * n
let k = f 10
```

Syntax and Semantics

- Syntax is about the form of sentences in a language.
- Semantics is about the meaning of sentences.
- Syntax: `Let's keep in touch!`
- Semantics: *Bye bye!*
- Syntax:

```
let f n = n * n
let k = f 10
```

- Semantics:
  f is a function computing the square of its argument n;
  k is the result of applying f to integer 10.

Semantics of Programming Languages

- The semantics of a programming language is a systematic way of giving meanings to programs written in the language.

Semantics of Programming Languages

- The semantics of a programming language is a systematic way of giving meanings to programs written in the language.
- Operational semantics: A program means what a machine interprets it to be.
- Denotational semantics: A program denotes a mathematical object independent of its machine execution.

Semantics of Programming Languages

- The semantics of a programming language is a systematic way of giving meanings to programs written in the language.
- Operational semantics: A program means what a machine interprets it to be.
- Denotational semantics: A program denotes a mathematical object independent of its machine execution.
- The denotational semantics and the operational semantics of a programming language shall closely relate to each other.

Semantics of Programming Languages

- The semantics of a programming language is a systematic way of giving meanings to programs written in the language.
- Operational semantics: A program means what a machine interprets it to be.
- Denotational semantics: A program denotes a mathematical object independent of its machine execution.
- The denotational semantics and the operational semantics of a programming language shall closely relate to each other.
- *Programs shall have precise and consistent meaning.*

Non-terminating Programs

What does program g mean?

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Non-terminating Programs

What does program g mean?

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Possible answers:

- $g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even,} \\ \mathrm{F} & \text{if } n \text{ is odd.} \end{cases}$

Non-terminating Programs

What does program g mean?

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Possible answers:

- $g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even,} \\ \mathrm{F} & \text{if } n \text{ is odd.} \end{cases}$
- $g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even,} \\ \mathrm{T} & \text{if } n \text{ is odd.} \end{cases}$

Non-terminating Programs

What does program g mean?

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Possible answers:

- $g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even,} \\ \mathrm{F} & \text{if } n \text{ is odd.} \end{cases}$
- $g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even,} \\ \mathrm{T} & \text{if } n \text{ is odd.} \end{cases}$
- $g_3(n)$ is *undefined* for all n, as the execution will not terminate (or, will not terminate normally).

Non-terminating Programs

What does program g mean?

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Possible answers:

- $g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even,} \\ \mathrm{F} & \text{if } n \text{ is odd.} \end{cases}$
- $g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even,} \\ \mathrm{T} & \text{if } n \text{ is odd.} \end{cases}$
- $g_3(n)$ is *undefined* for all n, as the execution will not terminate (or, will not terminate normally).

Which interpretation is accurate?

Non-terminating Programs, Continued

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Which of the following meaning of g is more accurate?

- $g_3(n) = \uparrow$
- $g_4(n) = \begin{cases} \mathrm{T} & \text{if } n = 0, \\ \mathrm{F} & \text{if } n = 1, \\ \uparrow & \text{otherwise.} \end{cases}$
- $g_5(n) = \begin{cases} \mathrm{F} & \text{if } n = 0, \\ \mathrm{T} & \text{if } n = 1, \\ \uparrow & \text{otherwise.} \end{cases}$

Note: We use $\uparrow$ as a shorthand for non-termination or abnormal termination. Functions g_3, g_4, and g_5 are *partial* functions.

A Notation for Functions

For a (partial) function f, we use the notation $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} \qquad g_1 = \begin{array}{l} \{(2n, \mathrm{T}) \mid n \geq 0\} \cup \\ \{(2n+1, \mathrm{F}) \mid n \geq 0\} \end{array}$$

A Notation for Functions

For a (partial) function f, we use the notation $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} \qquad g_1 = \begin{array}{l} \{(2n, \mathrm{T}) \mid n \geq 0\} \cup \\ \{(2n+1, \mathrm{F}) \mid n \geq 0\} \end{array}$$

$$g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even} \\ \mathrm{T} & \text{if } n \text{ is odd} \end{cases} \qquad g_2 = \begin{array}{l} \{(2n, \mathrm{F}) \mid n \geq 0\} \cup \\ \{(2n+1, \mathrm{T}) \mid n \geq 0\} \end{array}$$

A Notation for Functions

For a (partial) function f, we use the notation $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} \qquad g_1 = \{(2n, \mathrm{T}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{F}) \mid n \geq 0\}$$

$$g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even} \\ \mathrm{T} & \text{if } n \text{ is odd} \end{cases} \qquad g_2 = \{(2n, \mathrm{F}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{T}) \mid n \geq 0\}$$

$$g_3(n) = \uparrow \qquad g_3 = \emptyset$$

A Notation for Functions

For a (partial) function f, we use the notation $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} \qquad g_1 = \{(2n, \mathrm{T}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{F}) \mid n \geq 0\}$$

$$g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even} \\ \mathrm{T} & \text{if } n \text{ is odd} \end{cases} \qquad g_2 = \{(2n, \mathrm{F}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{T}) \mid n \geq 0\}$$

$$g_3(n) = \uparrow \qquad g_3 = \emptyset$$

$$g_4(n) = \begin{cases} \mathrm{T} & \text{if } n = 0 \\ \mathrm{F} & \text{if } n = 1 \\ \uparrow & \text{otherwise} \end{cases} \qquad g_4 = \{(0, \mathrm{T}),\ (1, \mathrm{F})\}$$

A Notation for Functions

For a (partial) function f, we use the notation $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$g_1(n) = \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} \qquad g_1 = \{(2n, \mathrm{T}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{F}) \mid n \geq 0\}$$

$$g_2(n) = \begin{cases} \mathrm{F} & \text{if } n \text{ is even} \\ \mathrm{T} & \text{if } n \text{ is odd} \end{cases} \qquad g_2 = \{(2n, \mathrm{F}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{T}) \mid n \geq 0\}$$

$$g_3(n) = \uparrow \qquad g_3 = \emptyset$$

$$g_4(n) = \begin{cases} \mathrm{T} & \text{if } n = 0 \\ \mathrm{F} & \text{if } n = 1 \\ \uparrow & \text{otherwise} \end{cases} \qquad g_4 = \{(0, \mathrm{T}),\ (1, \mathrm{F})\}$$

$$g_5(n) = \begin{cases} \mathrm{F} & \text{if } n = 0 \\ \mathrm{T} & \text{if } n = 1 \\ \uparrow & \text{otherwise} \end{cases} \qquad g_5 = \{(0, \mathrm{F}),\ (1, \mathrm{T})\}$$

Data Types and Sets

In programming languages, a data type can be viewed as a set of values, along with predefined operations on values in the set.

- For type `int`, we think of the set $Z = \{\ldots, 2, -1, 0, 1, 2, \ldots\}$ along with integer operations $+, -, \times, \div, \ldots$
- For type `bool`, we think of the set $B = \{\mathrm{T}, \mathrm{F}\}$, along with boolean operations $\vee, \wedge, \neg, \ldots$.

Data Types and Sets

In programming languages, a data type can be viewed as a set of values, along with predefined operations on values in the set.

- For type `int`, we think of the set $Z = \{\ldots, 2, -1, 0, 1, 2, \ldots\}$ along with integer operations $+, -, \times, \div, \ldots$
- For type `bool`, we think of the set $B = \{\mathrm{T}, \mathrm{F}\}$, along with boolean operations $\vee, \wedge, \neg, \ldots$.

This view, however, does not address non-terminating programs.

- Which element in B gives meaning to `(g 0)`?
- Of the 5 meanings g_1, g_2, g_3, g_4, g_5 for g, which one most accurately describes g?

Data Types and Domains

To address non-termination,

- For each data type, an element $\perp$ is introduced to the set of values to denote computational divergence.
- A partial order is established among the elements in the new set. This set is called the *domain* for the data type.

Data Types and Domains

To address non-termination,

- For each data type, an element $\perp$ is introduced to the set of values to denote computational divergence.
- A partial order is established among the elements in the new set. This set is called the *domain* for the data type.

We use $\sqsubseteq$ to denote "semantically weaker". We write $x \sqsubseteq y$ to mean that x is less defined than y computationally. That is, x has less information content than y has.

Data Types and Domains

To address non-termination,

- For each data type, an element $\perp$ is introduced to the set of values to denote computational divergence.
- A partial order is established among the elements in the new set. This set is called the *domain* for the data type.

We use $\sqsubseteq$ to denote "semantically weaker". We write $x \sqsubseteq y$ to mean that x is less defined than y computationally. That is, x has less information content than y has.

- For type `bool`, we now think of the domain $\mathcal{B} = \{\perp, \mathrm{T}, \mathrm{F}\}$.

Data Types and Domains

To address non-termination,

- For each data type, an element $\perp$ is introduced to the set of values to denote computational divergence.
- A partial order is established among the elements in the new set. This set is called the *domain* for the data type.

We use $\sqsubseteq$ to denote "semantically weaker". We write $x \sqsubseteq y$ to mean that x is less defined than y computationally. That is, x has less information content than y has.

- For type `bool`, we now think of the domain $\mathcal{B} = \{\perp, \mathrm{T}, \mathrm{F}\}$.
- Elements in $\mathcal{B}$ are ordered by $\perp \sqsubseteq \mathrm{T}$ and $\perp \sqsubseteq \mathrm{F}$. But $\mathrm{T} \not\sqsubseteq \mathrm{F}$ and $\mathrm{F} \not\sqsubseteq \mathrm{T}$.

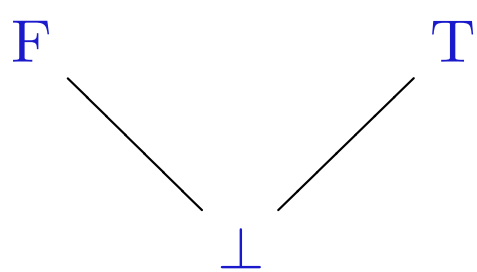
Data Types and Domains

To address non-termination,

- For each data type, an element $\perp$ is introduced to the set of values to denote computational divergence.
- A partial order is established among the elements in the new set. This set is called the *domain* for the data type.

We use $\sqsubseteq$ to denote "semantically weaker". We write $x \sqsubseteq y$ to mean that x is less defined than y computationally. That is, x has less information content than y has.

- For type `bool`, we now think of the domain $\mathcal{B} = \{\perp, \mathrm{T}, \mathrm{F}\}$.
- Elements in $\mathcal{B}$ are ordered by $\perp \sqsubseteq \mathrm{T}$ and $\perp \sqsubseteq \mathrm{F}$. But $\mathrm{T} \not\sqsubseteq \mathrm{F}$ and $\mathrm{F} \not\sqsubseteq \mathrm{T}$.
- Domain $\mathcal{B}$ illustrated:



Partially Ordered Set (poset)

Definition

A *partially ordered set* (poset) D is a set with a binary relation $\sqsubseteq_D \subseteq D \times D$ such that for every $x, y, z \in D$, the following properties fold:

1. (reflexive) $x \sqsubseteq_D x$.
2. (anti-symmetric) $x \sqsubseteq_D y$ and $y \sqsubseteq_D x$ implies $x = y$.
3. (transitive) $x \sqsubseteq_D y$ and $y \sqsubseteq_D z$ implies $x \sqsubseteq_D z$.

□

Partially Ordered Set (poset)

Definition

A *partially ordered set* (poset) D is a set with a binary relation $\sqsubseteq_D \subseteq D \times D$ such that for every $x, y, z \in D$, the following properties fold:

1. (reflexive) $x \sqsubseteq_D x$.
2. (anti-symmetric) $x \sqsubseteq_D y$ and $y \sqsubseteq_D x$ implies $x = y$.
3. (transitive) $x \sqsubseteq_D y$ and $y \sqsubseteq_D z$ implies $x \sqsubseteq_D z$.

□

- $B = \{\mathrm{T}, \mathrm{F}\}$ with $\sqsubseteq_B = \{(\mathrm{F}, \mathrm{F}), (\mathrm{T}, \mathrm{T})\}$ is a poset.

Partially Ordered Set (poset)

Definition

A *partially ordered set* (poset) D is a set with a binary relation $\sqsubseteq_D \subseteq D \times D$ such that for every $x, y, z \in D$, the following properties fold:

1. (reflexive) $x \sqsubseteq_D x$.
2. (anti-symmetric) $x \sqsubseteq_D y$ and $y \sqsubseteq_D x$ implies $x = y$.
3. (transitive) $x \sqsubseteq_D y$ and $y \sqsubseteq_D z$ implies $x \sqsubseteq_D z$.

□

- $B = \{\mathrm{T}, \mathrm{F}\}$ with $\sqsubseteq_B = \{(\mathrm{F}, \mathrm{F}), (\mathrm{T}, \mathrm{T})\}$ is a poset.
- $\mathcal{B} = \{\bot, \mathrm{F}, \mathrm{T}\}$ with $\sqsubseteq_{\mathcal{B}} = \{(\bot, \bot), (\mathrm{F}, \mathrm{F}), (\mathrm{T}, \mathrm{T}), (\bot, \mathrm{F}), (\bot, \mathrm{T})\}$ is also a poset.

Directed Set

Definition

Let D be a poset. A set $X \subseteq D$ is *directed* if

1. $X \neq \emptyset$.
2. For all $x, y \in X$ there is a $z \in X$ such that $x \sqsubseteq_D z$ and $y \sqsubseteq_D z$.

□

Directed Set

Definition

Let D be a poset. A set $X \subseteq D$ is *directed* if

1. $X \neq \emptyset$.
2. For all $x, y \in X$ there is a $z \in X$ such that $x \sqsubseteq_D z$ and $y \sqsubseteq_D z$.

□

- A directed set X of a poset D can be viewed as an approximation for some computation in D.

Directed Set

Definition

Let D be a poset. A set $X \subseteq D$ is *directed* if

1. $X \neq \emptyset$.
2. For all $x, y \in X$ there is a $z \in X$ such that $x \sqsubseteq_D z$ and $y \sqsubseteq_D z$.

□

- A directed set X of a poset D can be viewed as an approximation for some computation in D.
- It is an approximation because for every two elements $x, y \in X$, there is always a more defined element $z \in X$ which x and y can progress to.

Complete Partial Order (cpo)

Definition

Let D be a poset. D is a *complete partial order* (cpo) if

1. There is a least element $\bot_D \in D$ such that for all $x \in D$, $\bot_D \sqsubseteq_D x$.
2. Every directed set $X \subseteq D$ has a *least upper bound* (lub) $\bigsqcup X \in D$.

□

Complete Partial Order (cpo)

Definition

Let D be a poset. D is a *complete partial order* (cpo) if

1. There is a least element $\bot_D \in D$ such that for all $x \in D$, $\bot_D \sqsubseteq_D x$.
2. Every directed set $X \subseteq D$ has a *least upper bound* (lub) $\bigsqcup X \in D$.

□

- That is, for a cpo D and an approximation $X \subseteq D$, the approximation in X must progress to an unique element (the lub) in D (though not necessarily in X).

Complete Partial Order (cpo)

Definition

Let D be a poset. D is a *complete partial order* (cpo) if

1. There is a least element $\bot_D \in D$ such that for all $x \in D$, $\bot_D \sqsubseteq_D x$.
2. Every directed set $X \subseteq D$ has a *least upper bound* (lub) $\bigsqcup X \in D$.

□

- That is, for a cpo D and an approximation $X \subseteq D$, the approximation in X must progress to an unique element (the lub) in D (though not necessarily in X).
- We use cpo as the domain for denotational semantics. The terms cpo and domain are used interchangeably.

Complete Partial Order (cpo)

Definition

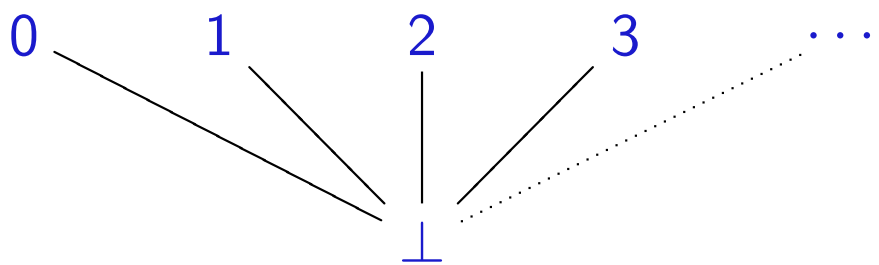
Let D be a poset. D is a *complete partial order* (cpo) if

1. There is a least element $\bot_D \in D$ such that for all $x \in D$, $\bot_D \sqsubseteq_D x$.
2. Every directed set $X \subseteq D$ has a *least upper bound* (lub) $\bigsqcup X \in D$.

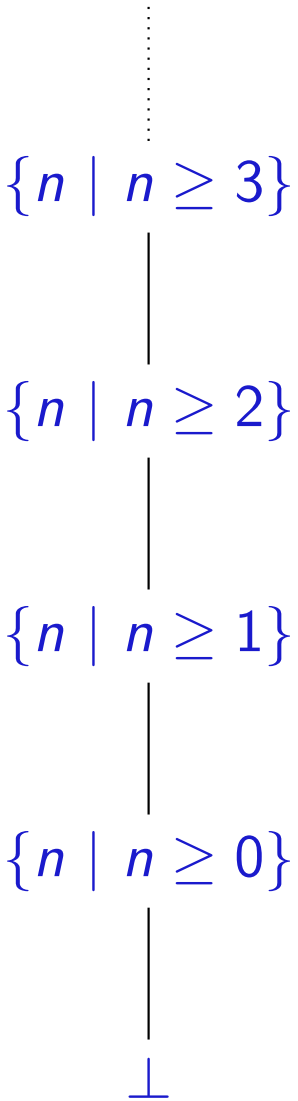
□

- That is, for a cpo D and an approximation $X \subseteq D$, the approximation in X must progress to an unique element (the lub) in D (though not necessarily in X).
- We use cpo as the domain for denotational semantics. The terms cpo and domain are used interchangeably.
- The subscript D in $\sqsubseteq_D$ and $\bot_D$ is often omitted if it is clear.

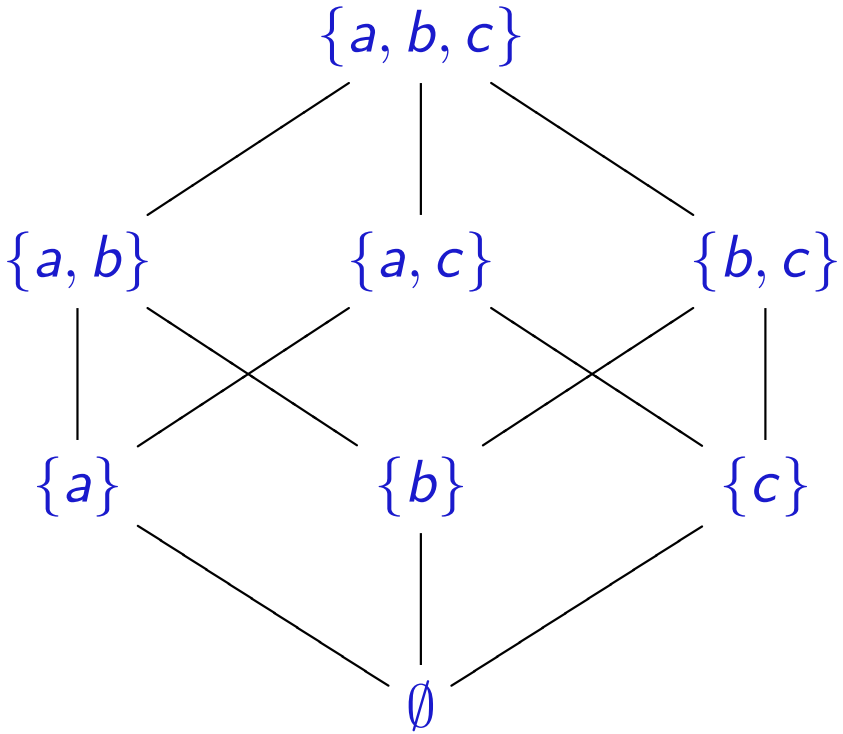
Domain $\mathcal{N}$



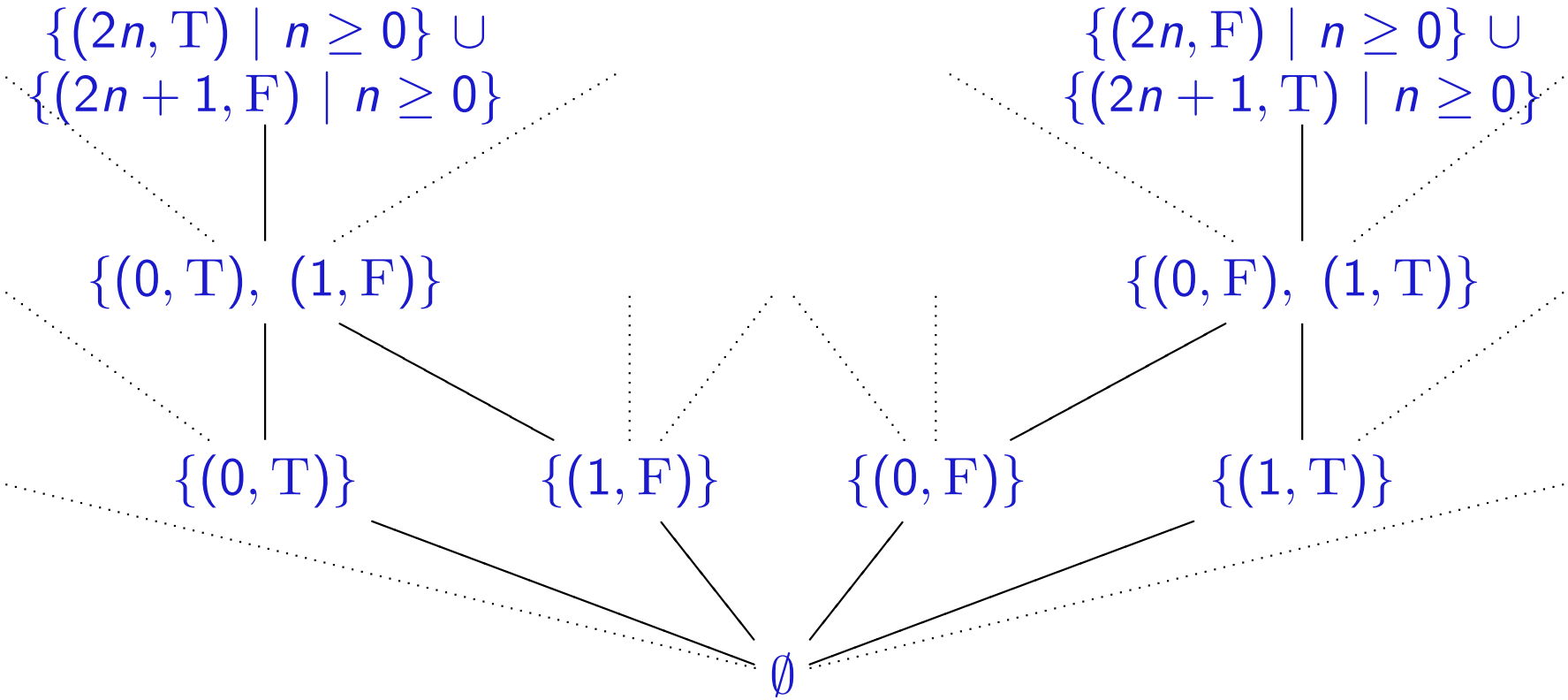
A Possible View of Natural Numbers



A Domain Built from Subsets



A Domain of Partial Functions



Domain Lifting

Definition

Let D be a domain. Define poset $\mathrm{lift}(D)$ by

1. $\mathrm{lift}(D) = D \cup \{\bot_{\mathrm{lift}(D)}\}$, $\bot_{\mathrm{lift}(D)} \notin D$.
2. $x \sqsubseteq_{\mathrm{lift}(D)} y$ if and only if $x = \bot_{\mathrm{lift}(D)}$ or $x \sqsubseteq_D y$.

□

Domain Lifting

Definition

Let D be a domain. Define poset $\mathrm{lift}(D)$ by

1. $\mathrm{lift}(D) = D \cup \{\bot_{\mathrm{lift}(D)}\}$, $\bot_{\mathrm{lift}(D)} \notin D$.
2. $x \sqsubseteq_{\mathrm{lift}(D)} y$ if and only if $x = \bot_{\mathrm{lift}(D)}$ or $x \sqsubseteq_D y$.

□

- If D is a domain then $\mathrm{lift}(D)$ forms a domain.
- $\mathrm{lift}(D)$ is called the *lifted* domain of D.

Domain *1* and Domain *2*

Example

Let *1* be the poset $\{\bot\}$ where $\bot \sqsubseteq_1 \bot$. *1* is a domain. □

Domain *1* and Domain *2*

Example

Let *1* be the poset $\{\bot\}$ where $\bot \sqsubseteq_1 \bot$. *1* is a domain. □

Example

Let $2 = \mathrm{lift}(1)$. Then *2* is a domain. *2* has only two elements, $\bot$ and $\top$, and $\bot \sqsubseteq_2 \top$. □

Domain *1* and Domain *2*

Example

Let 1 be the poset $\{\bot\}$ where $\bot \sqsubseteq_1 \bot$. 1 is a domain. □

Example

Let $2 = \mathrm{lift}(1)$. Then 2 is a domain. 2 has only two elements, $\bot$ and $\top$, and $\bot \sqsubseteq_2 \top$. □

For domain 2, the least upper bound operator $\sqcup$ and the greatest lower bound operator $\sqcap$ are defined by the following.

$\sqcup$	$\bot$	$\top$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

$\sqcap$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

(Look familiar?)

The Sum of Two Domains

Definition

Let D and D' be domains. Define poset $D + D'$ by

1. $D + D' = D \cup D' \cup \{\bot_{D+D'}\}$, where the elements in D are made distinct from the elements in D'. $\bot_{D+D'} \notin D \cup D'$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x = \bot_{D+D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

□

The Sum of Two Domains

Definition

Let D and D' be domains. Define poset $D + D'$ by

1. $D + D' = D \cup D' \cup \{\bot_{D+D'}\}$, where the elements in D are made distinct from the elements in D'. $\bot_{D+D'} \notin D \cup D'$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x = \bot_{D+D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

□

- If both D and D' are domains, then $D + D'$ is a domain too.
- $D + D'$ is called the *sum* of D and D'.

The Coalesced Sum of Two Domains

Definition

Let D and D' be domains. Define poset $D \oplus D'$ by

1. $D \oplus D' = D \cup D' \cup \{\bot_{D \oplus D'}\}$, where the elements in D are made distinct from the elements in D', except $\bot_D$ and $\bot_{D'}$. $\bot_{D \oplus D'} = \bot_D = \bot_{D'}$.
2. $x \sqsubseteq_{D \oplus D'} y$ if and only if $x = \bot_{D \oplus D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

□

The Coalesced Sum of Two Domains

Definition

Let D and D' be domains. Define poset $D \oplus D'$ by

1. $D \oplus D' = D \cup D' \cup \{\bot_{D \oplus D'}\}$, where the elements in D are made distinct from the elements in D', except $\bot_D$ and $\bot_{D'}$. $\bot_{D \oplus D'} = \bot_D = \bot_{D'}$.
2. $x \sqsubseteq_{D \oplus D'} y$ if and only if $x = \bot_{D \oplus D'}$ or $x \sqsubseteq_D y$ or $x \sqsubseteq_{D'} y$.

□

- If both D and D' are domains, then $D \oplus D'$ is a domain too.
- $D \oplus D'$ is called the *coalesced sum* of D and D'.

The Product of Two Domains

Definition

Let D and D' be domains. Define $D \times D'$ by

1. $D \times D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$,
   $\bot_{D \times D'} = \langle \bot_D, \bot_{D'} \rangle$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D \times D'} \langle d_2, d_2' \rangle$ if and only if $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

□

The Product of Two Domains

Definition

Let D and D' be domains. Define $D \times D'$ by

1. $D \times D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$,
   $\bot_{D \times D'} = \langle \bot_D, \bot_{D'} \rangle$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D \times D'} \langle d_2, d_2' \rangle$ if and only if $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

□

- If both D and D' are domains, then $D \times D'$ is a domain too.
- $D \times D'$ is called the *product* of D and D'.

The Smash Product of Two Domains

Definition

Let D and D' be domains. Define $D \otimes D'$ by

1. $D \otimes D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$.
   $\bot_{D \otimes D'} = \langle \bot_D, d' \rangle = \langle d, \bot_{D'} \rangle$ for any $d \in D$ and $d' \in D'$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D \otimes D'} \langle d_2, d_2' \rangle$ if and only if $\langle d_1, d_1' \rangle = \bot_{D \otimes D'}$, or $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

□

The Smash Product of Two Domains

Definition

Let D and D' be domains. Define $D \otimes D'$ by

1. $D \otimes D' = \{ \langle d, d' \rangle \mid d \in D,\ d' \in D' \}$.
   $\bot_{D \otimes D'} = \langle \bot_D, d' \rangle = \langle d, \bot_{D'} \rangle$ for any $d \in D$ and $d' \in D'$.
2. $\langle d_1, d_1' \rangle \sqsubseteq_{D \otimes D'} \langle d_2, d_2' \rangle$ if and only if $\langle d_1, d_1' \rangle = \bot_{D \otimes D'}$, or $d_1 \sqsubseteq_D d_2$ and $d_1' \sqsubseteq_{D'} d_2'$.

□

- If both D and D' are domains, then $D \otimes D'$ is a domain too.
- $D \otimes D'$ is called the *smashed product* of D and D'.

Continuous Function

Definition

Let D and D' be domains, and f be a *total* function from D to D'.

1. f is *monotonic* if and only if $f(d_1) \sqsubseteq_{D'} f(d_2)$ whenever $d_1 \sqsubseteq_D d_2$.
2. f is *continuous* if and only if $f(\bigsqcup X) = \bigsqcup f\{X\}$ for every directed set $X \subseteq D$, where $f\{X\}$ is defined as $\{f(x) \mid x \in X\}$.
3. f is *strict* if and only if $f(\bot_D) = \bot_{D'}$.

□

Continuous Function

Definition

Let D and D' be domains, and f be a *total* function from D to D'.

1. f is *monotonic* if and only if $f(d_1) \sqsubseteq_{D'} f(d_2)$ whenever $d_1 \sqsubseteq_D d_2$.
2. f is *continuous* if and only if $f(\bigsqcup X) = \bigsqcup f\{X\}$ for every directed set $X \subseteq D$, where $f\{X\}$ is defined as $\{f(x) \mid x \in X\}$.
3. f is *strict* if and only if $f(\bot_D) = \bot_{D'}$.

□

- If a function f is not strict, it is called *non–strict*.
- If a function is continuous then it is monotonic, but the reverse is not true.

Continuous Function Space

Definition

Let D and D' be domains. Define $D \to D'$ by

1. $D \to D' = \{f \mid f \text{ is a continuous function from } D \text{ to } D'\}$, and $\bot_{D \to D'} = \{(d, \bot_{D'}) \mid d \in D\}$.
2. $f \sqsubseteq_{D \to D'} g$ if and only if for all $d \in D, f(d) \sqsubseteq_{D'} g(d)$.

□

Continuous Function Space as Domain, I

Theorem (Scott)

The continuous function space $D \to D'$ is a domain if both D and D' are domains. □

Continuous Function Space as Domain, I

Theorem (Scott)

The continuous function space $D \to D'$ is a domain if both D and D' are domains. □

Proof.

We need to show that every directed set $F \subseteq D \to D'$ has a least upper bound (lub) and this lub is itself a continuous function.

Continuous Function Space as Domain, I

Theorem (Scott)

The continuous function space $D \to D'$ is a domain if both D and D' are domains. □

Proof.

We need to show that every directed set $F \subseteq D \to D'$ has a least upper bound (lub) and this lub is itself a continuous function. Let

$$F^{\sqcup} = \{(d, \bigsqcup \{f(d) \mid f \in F\}) \mid d \in D\}$$

Continuous Function Space as Domain, I

Theorem (Scott)

The continuous function space $D \to D'$ is a domain if both D and D' are domains. □

Proof.

We need to show that every directed set $F \subseteq D \to D'$ has a least upper bound (lub) and this lub is itself a continuous function. Let

$$F^{\sqcup} = \{(d, \bigsqcup \{f(d) \mid f \in F\}) \mid d \in D\}$$

Since F is directed, we know that, for any $d \in D$, $\{f(d) \mid f \in F\} \subseteq D'$ is directed as well. Because D' is a domain, $\bigsqcup \{f(d) \mid f \in F\}$ exists hence function $F^{\sqcup}$ is well defined. Moreover, by construction, we observe that $F^{\sqcup}$ is the lub of F.
(To be continued) □

Continuous Function Space as Domain, II

Proof (Continued).

Why is $\{f(d) \mid f \in F\} \subseteq D'$ directed, and why is $F^{\sqcup}$ the lub of F?

Continuous Function Space as Domain, II

Proof (Continued).

Why is $\{f(d) \mid f \in F\} \subseteq D'$ directed, and why is $F^{\sqcup}$ the lub of F?

Let $u, v \in \{f(d) \mid f \in F\}$. We have $u = f\ d$ and $v = g\ d$ for some $f, g \in F$. As F is directed, there is a $h \in F$ such that $f \sqsubseteq_{D \to D'} h$ and $g \sqsubseteq_{D \to D'} h$. That is, $f\ d \sqsubseteq_{D'} h\ d$ and $g\ d \sqsubseteq_{D'} h\ d$. Since $h\ d \in \{f(d) \mid f \in F\}$, we conculde the set is directed.

Continuous Function Space as Domain, II

Proof (Continued).

Why is $\{f(d) \mid f \in F\} \subseteq D'$ directed, and why is $F^{\sqcup}$ the lub of F?

Let $u, v \in \{f(d) \mid f \in F\}$. We have $u = f\ d$ and $v = g\ d$ for some $f, g \in F$. As F is directed, there is a $h \in F$ such that $f \sqsubseteq_{D \to D'} h$ and $g \sqsubseteq_{D \to D'} h$. That is, $f\ d \sqsubseteq_{D'} h\ d$ and $g\ d \sqsubseteq_{D'} h\ d$. Since $h\ d \in \{f(d) \mid f \in F\}$, we conculde the set is directed.

We first observe that $f \sqsubseteq_{D \to D'} F^{\sqcup}$ for all $f \in F$. That is, $F^{\sqcup}$ is an upper bound of F. Suppose w is also an upper bound of F. That is, $f \sqsubseteq_{D \to D'} w$ for all $f \in F$; hence, for any $d \in D$, $f\ d \sqsubseteq_{D'} w\ d$. Taking the lub at both sides of $\sqsubseteq$, we arrive at $\bigsqcup\{f(d) \mid f \in F\} = F^{\sqcup}\ d \sqsubseteq_{D'} w\ d$ for any $d \in D$. That is, $F^{\sqcup}$ is the lub of F. $\square$

Continuous Function Space as Domain, III

Proof (Continued).

Is $F^{\sqcup}$ a continuous function? For all directed set $X \subseteq D$, we have

$$
\begin{aligned}
& F^{\sqcup}(\bigsqcup X) & \\
= \quad & \bigsqcup_{f \in F} f(\bigsqcup X) & \text{(Definition of } F^{\sqcup}\text{)} \\
= \quad & \bigsqcup_{f \in F}(\bigsqcup_{x \in X} f(x)) & (X \subseteq D \text{ is directed; each } f \text{ is continuous}) \\
= \quad & \bigsqcup_{x \in X}(\bigsqcup_{f \in F} f(x)) & \text{(Rearranging indices)} \\
= \quad & \bigsqcup_{x \in X} F^{\sqcup}(x) & \text{(Definition of } F^{\sqcup}\text{)} \\
= \quad & \bigsqcup F^{\sqcup}\{X\} & \text{(Definition of } \bigsqcup X\text{)}
\end{aligned}
$$

We conclude $F^{\sqcup}$ is continuous. □

Continuous Function Space as Domain, III

Proof (Continued).

Is $F^{\sqcup}$ a continuous function? For all directed set $X \subseteq D$, we have

$$
\begin{aligned}
& F^{\sqcup}(\bigsqcup X) & \\
= \quad & \bigsqcup_{f \in F} f(\bigsqcup X) & \text{(Definition of } F^{\sqcup}\text{)} \\
= \quad & \bigsqcup_{f \in F}(\bigsqcup_{x \in X} f(x)) & (X \subseteq D \text{ is directed; each } f \text{ is continuous}) \\
= \quad & \bigsqcup_{x \in X}(\bigsqcup_{f \in F} f(x)) & \text{(Rearranging indices)} \\
= \quad & \bigsqcup_{x \in X} F^{\sqcup}(x) & \text{(Definition of } F^{\sqcup}\text{)} \\
= \quad & \bigsqcup F^{\sqcup}\{X\} & \text{(Definition of } \bigsqcup X\text{)}
\end{aligned}
$$

We conclude $F^{\sqcup}$ is continuous. □

From now on, we write $\bigsqcup F$ to denote the function $F^{\sqcup}$, the least upper bound of a directed set of continuous functions F.

Why Continuous Function?

What are the motivations behind using continuous function spaces as the semantic domains of functions written in a programming language? Several reasons:

Why Continuous Function?

What are the motivations behind using continuous function spaces as the semantic domains of functions written in a programming language? Several reasons:

- We shall only admit *monotonic* functions. If x contains less information than y does, surely $f(x)$ shall yield less information than $f(y)$ does, regardless of what f is.

Why Continuous Function?

What are the motivations behind using continuous function spaces as the semantic domains of functions written in a programming language? Several reasons:

- We shall only admit *monotonic* functions. If x contains less information than y does, surely $f(x)$ shall yield less information than $f(y)$ does, regardless of what f is.
- If X is an approximation, then the result of applying f to $\bigsqcup X$ shall agree with $\bigsqcup f\{X\}$. That is, f can be understood as an approximation too.

Why Continuous Function?

What are the motivations behind using continuous function spaces as the semantic domains of functions written in a programming language? Several reasons:

- We shall only admit *monotonic* functions. If x contains less information than y does, surely $f(x)$ shall yield less information than $f(y)$ does, regardless of what f is.
- If X is an approximation, then the result of applying f to $\bigsqcup X$ shall agree with $\bigsqcup f\{X\}$. That is, f can be understood as an approximation too.
- In particular, we don't want to admit (non-continuous) functions that "jump" arbitrarily at the limit of an approximation.

Why Continuous Function?

What are the motivations behind using continuous function spaces as the semantic domains of functions written in a programming language? Several reasons:

- We shall only admit *monotonic* functions. If x contains less information than y does, surely $f(x)$ shall yield less information than $f(y)$ does, regardless of what f is.
- If X is an approximation, then the result of applying f to $\bigsqcup X$ shall agree with $\bigsqcup f\{X\}$. That is, f can be understood as an approximation too.
- In particular, we don't want to admit (non-continuous) functions that "jump" arbitrarily at the limit of an approximation.
- Continuous function spaces are themselves complete partial orders so work well with other semantic domains.

Fixed Points and The Least Fixed Point

Definition

Let D be a poset and let $f \in D \to D$ be a total function.

1. $x \in D$ is a *fixed point* of f if and only if $f(x) = x$.
2. x is the *least fixed point* of f if and only if x is a fixed point of f, and for every fixed point $d \in D$ of f, it implies $x \sqsubseteq_D d$.

□

Fixed Points and The Least Fixed Point

Definition

Let D be a poset and let $f \in D \to D$ be a total function.

1. $x \in D$ is a *fixed point* of f if and only if $f(x) = x$.
2. x is the *least fixed point* of f if and only if x is a fixed point of f, and for every fixed point $d \in D$ of f, it implies $x \sqsubseteq_D d$.

□

- Function $f(x) = x$, where $x \in \mathcal{B}$, have three fixed points: $\bot$, F, and T.
- $\bot$ is the least fixed point of f.

The Least Fixed Point Theorem

Theorem (Kleene)

Let D be a domain.

1. *Every function $f \in D \to D$ has a least fixed point.*
2. *There exists a function $\mathrm{fix} \in (D \to D) \to D$ such that for every function $f \in D \to D$, $\mathrm{fix}\,(f)$ is the least fixed point of f.*

□

The Least Fixed Point Theorem

Theorem (Kleene)

Let D be a domain.

1. *Every function $f \in D \to D$ has a least fixed point.*
2. *There exists a function $\mathrm{fix} \in (D \to D) \to D$ such that for every function $f \in D \to D$, $\mathrm{fix}\,(f)$ is the least fixed point of f.*

□

Proof.

1. $X_f = \{\bot_D,\ \ f(\bot_D),\ \ f(f(\bot_D)),\ \ \ldots,\ \ f^{(n)}(\bot_D),\ \ \ldots\}$ is a directed set because $\bot_D \sqsubseteq_D f(\bot_D)$, $f(\bot_D) \sqsubseteq_D f(f(\bot_D))$, $\ldots$.
By the continuity of f,

$$f(\bigsqcup X_f) = \bigsqcup f\{X_f\} = \bigsqcup X_f$$

Hence, $\bigsqcup X_f$ is a fixed point of f. (To be continued) □

The Least Fixed Point Theorem, Continued

Proof (Continued).

Moreover, suppose that d too is a fixed point of f. Then

$$\bot_D \sqsubseteq_D d,\ f(\bot_D) \sqsubseteq_D f(d) = d,\ \ldots,\ f^{(n)}(\bot_D) \sqsubseteq_D f^{(n)}(d) = d,\ \ldots$$

Taking the lub of both sides, it follows that $\bigsqcup X_f \sqsubseteq_D d$.

The Least Fixed Point Theorem, Continued

Proof (Continued).

Moreover, suppose that d too is a fixed point of f. Then

$$\bot_D \sqsubseteq_D d,\ f(\bot_D) \sqsubseteq_D f(d) = d,\ \ldots,\ f^{(n)}(\bot_D) \sqsubseteq_D f^{(n)}(d) = d,\ \ldots$$

Taking the lub of both sides, it follows that $\bigsqcup X_f \sqsubseteq_D d$.

2. Define function *fix* by

$$fix\,(f) = \bigsqcup X_f$$

Then $fix\,(f)$ is the least fixed point of f. Moreover, by rearranging indices, we can show that, for all directed set $F \subseteq D \to D$

$$fix\,(\bigsqcup F) = \bigsqcup\ fix\{F\}$$

That is, f is continuous hence $f \in (D \to D) \to D$. □

Why The Least Fixed Point?

The least fixed point of a function f can be used to give meaning to a recursively defined function g.

Why The Least Fixed Point?

The least fixed point of a function f can be used to give meaning to a recursively defined function g.

- Take the following recursive definition of g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Why The Least Fixed Point?

The least fixed point of a function f can be used to give meaning to a recursively defined function g.

- Take the following recursive definition of g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

- We define a non-recursive function f

```
let f g n = if (n mod 2 = 0)
              then not (g (n+1))
              else not (g (n-1))
```

Why The Least Fixed Point?

The least fixed point of a function f can be used to give meaning to a recursively defined function g.

- Take the following recursive definition of g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

- We define a non-recursive function f

```
let f g n = if (n mod 2 = 0)
              then not (g (n+1))
              else not (g (n-1))
```

- If f has a meaning $f \in (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B})$, then by the least fixed point theorem, $fix(f) = f(fix(f))$. This matches the recursive definition of g.

Why The Least Fixed Point?

The least fixed point of a function f can be used to give meaning to a recursively defined function g.

- Take the following recursive definition of g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

- We define a non-recursive function f

```
let f g n = if (n mod 2 = 0)
              then not (g (n+1))
              else not (g (n-1))
```

- If f has a meaning $f \in (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B})$, then by the least fixed point theorem, $fix(f) = f(fix(f))$. This matches the recursive definition of g.
- We then assign $g = fix(f) \in \mathcal{N} \to \mathcal{B}$ as the meaning of g.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.
- For built-in constants of a data type, map them to the corresponding values in the domain for the type.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.
- For built-in constants of a data type, map them to the corresponding values in the domain for the type.
  - `()` is mapped to $\top \in 2$, `true` is mapped to $\mathrm{T} \in \mathcal{B}$, `not` is mapped to a function $\mathit{not} \in \mathcal{B} \to \mathcal{B}$ where $\mathit{not} = \{(\bot, \bot),\ (\mathrm{F}, \mathrm{T}),\ (\mathrm{T}, \mathrm{F})\}$.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.
- For built-in constants of a data type, map them to the corresponding values in the domain for the type.
  - `()` is mapped to $\top \in 2$, `true` is mapped to $\mathrm{T} \in \mathcal{B}$, `not` is mapped to a function $\mathit{not} \in \mathcal{B} \to \mathcal{B}$ where $\mathit{not} = \{(\bot, \bot),\ (\mathrm{F}, \mathrm{T}),\ (\mathrm{T}, \mathrm{F})\}$.
  - We write $[\![()]\!]_2 = \top$, $[\![\texttt{true}]\!]_{\mathcal{B}} = \mathrm{T}$, and $[\![\texttt{not}]\!]_{\mathcal{B}\to\mathcal{B}} = \mathit{not}$, etc.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.
- For built-in constants of a data type, map them to the corresponding values in the domain for the type.
  - `()` is mapped to $\top \in 2$, `true` is mapped to $\mathrm{T} \in \mathcal{B}$, `not` is mapped to a function $\mathit{not} \in \mathcal{B} \to \mathcal{B}$ where $\mathit{not} = \{(\bot, \bot),\ (\mathrm{F}, \mathrm{T}),\ (\mathrm{T}, \mathrm{F})\}$.
  - We write $[\![\texttt{()}]\!]_2 = \top$, $[\![\texttt{true}]\!]_{\mathcal{B}} = \mathrm{T}$, and $[\![\texttt{not}]\!]_{\mathcal{B}\to\mathcal{B}} = \mathit{not}$, etc.
- For a user-defined term, construct a semantic equation based on its definition, reusing existing terms and constants.

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain 2, type `bool` is domain $\mathcal{B}$, type `nat` is domain $\mathcal{N}$, etc.
  - For a user-defined data type, construct a domain equation to the specification of the type, then solve the equation.
- For built-in constants of a data type, map them to the corresponding values in the domain for the type.
  - `()` is mapped to $\top \in 2$, `true` is mapped to $\mathrm{T} \in \mathcal{B}$, `not` is mapped to a function $\mathit{not} \in \mathcal{B} \to \mathcal{B}$ where $\mathit{not} = \{(\bot, \bot),\ (\mathrm{F}, \mathrm{T}),\ (\mathrm{T}, \mathrm{F})\}$.
  - We write $[\![\texttt{()}]\!]_2 = \top$, $[\![\texttt{true}]\!]_{\mathcal{B}} = \mathrm{T}$, and $[\![\texttt{not}]\!]_{\mathcal{B}\to\mathcal{B}} = \mathit{not}$, etc.
- For a user-defined term, construct a semantic equation based on its definition, reusing existing terms and constants.
- If the definition is recursive, compute the least fixed point.

Compose Meaning for Programs, An Example

For the following program g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

Compose Meaning for Programs, An Example

For the following program g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

We compose the following function $f \in (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B})$:

$$f\ g\ n = \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ (\mathit{mod}\ n\ 2)\ 0)\ (\mathit{not}\ (g\ (\mathit{plus}\ n\ 1)))\ (\mathit{not}\ (g\ (\mathit{minus}\ n\ 1)))$$

Compose Meaning for Programs, An Example

For the following program g:

```
let rec g n = if (n mod 2 = 0)
                then not (g (n+1))
                else not (g (n-1))
```

We compose the following function $f \in (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B})$:

$$f\ g\ n = \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ (\mathit{mod}\ n\ 2)\ 0)\ (\mathit{not}\ (g\ (\mathit{plus}\ n\ 1)))\ (\mathit{not}\ (g\ (\mathit{minus}\ n\ 1)))$$

where functions

$$\begin{aligned} \mathit{if\text{-}then\text{-}else} &\in \mathcal{B} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N} \\ \mathit{eq} &\in \mathcal{N} \to \mathcal{N} \to \mathcal{B} \\ \mathit{not} &\in \mathcal{B} \to \mathcal{B} \\ \mathit{mod}, \mathit{plus}, \mathit{minus} &\in \mathcal{N} \to \mathcal{N} \to \mathcal{N} \end{aligned}$$

if-then-else	$\bot$	x	y	$=$	$\bot$
if-then-else	T	x	y	$=$	x
if-then-else	F	x	y	$=$	y
eq	$\bot$	y		$=$	$\bot$
eq	x	$\bot$		$=$	$\bot$
eq	x	y		$=$	T
eq	x	y		$=$	F
not	$\bot$			$=$	$\bot$
not	F			$=$	T
not	T			$=$	F
$\ldots$					
minus	$\bot$	y		$=$	$\bot$
minus	x	$\bot$		$=$	$\bot$
minus	x	y		$=$	$\bot$
minus	x	y		$=$	$x - y$

where $x = y \neq \bot$

otherwise

where $x < y$

otherwise

The Least Fixed Point Iteration

Start with $\bot_{\mathcal{N}\to\mathcal{B}} = \{(n, \bot) \mid n \in \mathcal{N}\}$, we compute the least upper bound of the following directed set

$$\{\bot_{\mathcal{N}\to\mathcal{B}}, \quad f(\bot_{\mathcal{N}\to\mathcal{B}}), \quad f(f(\bot_{\mathcal{N}\to\mathcal{B}})), \quad \ldots\}$$

The Least Fixed Point Iteration

Start with $\bot_{\mathcal{N}\to\mathcal{B}} = \{(n, \bot) \mid n \in \mathcal{N}\}$, we compute the least upper bound of the following directed set

$$\{\bot_{\mathcal{N}\to\mathcal{B}},\quad f(\bot_{\mathcal{N}\to\mathcal{B}}),\quad f(f(\bot_{\mathcal{N}\to\mathcal{B}})),\quad \ldots\}$$

Note that $f(\bot_{\mathcal{N}\to\mathcal{B}})$ computes to

$$g\ n\ =\ \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ (\mathit{mod}\ n\ 2)\ 0)\ \bot\ \bot$$

This is simplified to

$$g\ n\ =\ \bot$$

The Least Fixed Point Iteration

Start with $\bot_{\mathcal{N}\to\mathcal{B}} = \{(n, \bot) \mid n \in \mathcal{N}\}$, we compute the least upper bound of the following directed set

$$\{\bot_{\mathcal{N}\to\mathcal{B}},\quad f(\bot_{\mathcal{N}\to\mathcal{B}}),\quad f(f(\bot_{\mathcal{N}\to\mathcal{B}})),\quad \ldots\}$$

Note that $f(\bot_{\mathcal{N}\to\mathcal{B}})$ computes to

$$g\ n\ =\ \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ (\mathit{mod}\ n\ 2)\ 0)\ \bot\ \bot$$

This is simplified to

$$g\ n\ =\ \bot$$

That is, we have reached $\bot_{\mathcal{N}\to\mathcal{B}}$ as the least fixed point of f. We conclude that

$$[\![\mathrm{g}]\!]_{\mathcal{N}\to\mathcal{B}} = \bot_{\mathcal{N}\to\mathcal{B}} = \{(n, \bot) \mid n \in \mathcal{N}\}$$

That is, g will not terminate for any given input.

The Factorial Program

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

The Factorial Program

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

We compose the following function $f \in (\mathcal{N} \to \mathcal{N}) \to (\mathcal{N} \to \mathcal{N})$:

$$f\ fac\ n \quad = \quad \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ 1\ (\mathit{multi}\ n\ (\mathit{fac}\ (\mathit{minus}\ n\ 1)))$$

The Factorial Program

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

We compose the following function $f \in (\mathcal{N} \to \mathcal{N}) \to (\mathcal{N} \to \mathcal{N})$:

$$f\ fac\ n \quad = \quad \textit{if-then-else}\ (\textit{eq}\ n\ 0)\ 1\ (\textit{multi}\ n\ (\textit{fac}\ (\textit{minus}\ n\ 1)))$$

Start with $\bot_{\mathcal{N}\to\mathcal{N}} = \{(n, \bot) \mid n \in \mathcal{N}\}$, the least fixed point iteration for f will be

$$\begin{aligned}
f^{(0)}(\bot_{\mathcal{N}\to\mathcal{B}}) &= \{(n, \bot) \mid n \in \mathcal{N}\} \\
f^{(1)}(\bot_{\mathcal{N}\to\mathcal{B}}) &= \{(0,1)\} \cup \{(n, \bot) \mid n \notin \{0\}\} \\
f^{(2)}(\bot_{\mathcal{N}\to\mathcal{B}}) &= \{(0,1),\ (1,1)\} \cup \{(n, \bot) \mid n \notin \{0,1\}\} \\
f^{(3)}(\bot_{\mathcal{N}\to\mathcal{B}}) &= \{(0,1),\ (1,1),\ (2,2)\} \cup \{(n, \bot) \mid n \notin \{0,1,2\}\} \\
&\vdots \\
f^{(k+1)}(\bot_{\mathcal{N}\to\mathcal{B}}) &= \{(n, n!) \mid n \leq k\} \cup \{(n, \bot) \mid n \notin \{0,1,\ldots,k\}\} \\
&\vdots
\end{aligned}$$

Dealing With Mutual Recursion

For functions even and odd defined as

```
let rec even n = if n=0 then true  else odd  (n-1)
    and odd  n = if n=0 then false else even (n-1)
```

Dealing With Mutual Recursion

For functions even and odd defined as

```
let rec even n = if n=0 then true  else odd  (n-1)
    and odd  n = if n=0 then false else even (n-1)
```

We first define the following (non-recursive) function $f \in (\mathcal{N} \to \mathcal{B}) \times (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B}) \times (\mathcal{N} \to \mathcal{B})$:

$$f\ (even,\ odd) =$$
$$(\{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{T}\ (\mathit{odd}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\},$$
$$\{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{F}\ (\mathit{even}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\})$$

Dealing With Mutual Recursion

For functions even and odd defined as

```
let rec even n = if n=0 then true  else odd  (n-1)
    and odd  n = if n=0 then false else even (n-1)
```

We first define the following (non-recursive) function $f \in (\mathcal{N} \to \mathcal{B}) \times (\mathcal{N} \to \mathcal{B}) \to (\mathcal{N} \to \mathcal{B}) \times (\mathcal{N} \to \mathcal{B})$:

$$f\ (even,\ odd) =$$
$$(\{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{T}\ (\mathit{odd}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\},$$
$$\{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{F}\ (\mathit{even}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\})$$

Note that the least fixed point of f is a pair of functions $(even, odd)$ mutually satisfying

$$\begin{aligned} even &= \{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{T}\ (\mathit{odd}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\} \\ odd &= \{(n, \mathit{if\text{-}then\text{-}else}\ (\mathit{eq}\ n\ 0)\ \mathrm{F}\ (\mathit{even}\ (\mathit{minus}\ n\ 1))) \mid n \in \mathcal{N}\} \end{aligned}$$

Dealing With Mutual Recursion, Continued

We then start the least fixed point iteration with $(\bot_{\mathcal{N}\to\mathcal{B}},\ \bot_{\mathcal{N}\to\mathcal{B}})$, and get

	$\bot$	0	1	2	3	...
$even^{(0)}$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
$odd^{(0)}$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
$even^{(1)}$	$\bot$	T	$\bot$	$\bot$	$\bot$	...
$odd^{(1)}$	$\bot$	F	$\bot$	$\bot$	$\bot$	...
$even^{(2)}$	$\bot$	T	F	$\bot$	$\bot$	...
$odd^{(2)}$	$\bot$	F	T	$\bot$	$\bot$	...
$even^{(3)}$	$\bot$	T	F	T	$\bot$	...
$odd^{(3)}$	$\bot$	F	T	F	$\bot$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Modeling States of **While** Programs

- The execution of a **While** program results in a change to the state of the machine. A state is a mapping from variables to values where undefined variables are mapped to $\perp$.
- A state can be queried and updated. Let s be a state, x be a variable, and v be a value. Then,
  - $s\ x$ returns the value associated to variable x in state s.
  - $s[x \mapsto v]$ is the state identical to s except now variable x is mapped to v.
- Let $s = [x \mapsto 5,\ y \mapsto 7,\ z \mapsto 0]$, then
  - $s\ y$ is 7,
  - $s[x \mapsto 3]$ is $[x \mapsto 3,\ y \mapsto 7,\ z \mapsto 0]$.

Evaluation at the Presence of States

The semantics of arithmetic and boolean expressions in **While** programs is now defined by evaluation at the presence of states. As an example, let state $s = [\mathrm{x} \mapsto 3,\ \mathrm{y} \mapsto 7,\ \mathrm{z} \mapsto 0]$, then

$$\begin{aligned} [\![\mathrm{x}\ +\ 1]\!]_{\mathcal{N}}\ s &= [\![\mathrm{x}]\!]_{\mathcal{N}}\ s + [\![1]\!]_{\mathcal{N}}\ s \\ &= (s\ \mathrm{x}) + [\![1]\!]_{\mathcal{N}} \\ &= 3 + 1 \\ &= 4 \end{aligned}$$

and

$$\begin{aligned} [\![\neg(\mathrm{x}\ =\ 1)]\!]_{\mathcal{B}}\ s &= \mathit{not}\ ([\![\mathrm{x}\ =\ 1]\!]_{\mathcal{B}}\ s) \\ &= \mathit{not}\ ([\![\mathrm{x}]\!]_{\mathcal{N}}\ s\ =\ [\![1]\!]_{\mathcal{N}}\ s) \\ &= \mathit{not}\ ((s\ \mathrm{x}) = [\![1]\!]_{\mathcal{N}}) \\ &= \mathit{not}\ (3 = 1)\ =\ \mathit{not}\ \mathrm{F}\ =\ \mathrm{T} \end{aligned}$$

Semantics of **While** statements

- $[\![x := a]\!]_{\mathcal{S}}\ s = s[x \mapsto [\![a]\!]_{\mathcal{N}}\ s]$
- $[\![\texttt{skip}]\!]_{\mathcal{S}} = \mathrm{id}$
- $[\![S_1 ; S_2]\!]_{\mathcal{S}} = [\![S_2]\!]_{\mathcal{S}} \circ [\![S_1]\!]_{\mathcal{S}}$
- $[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]_{\mathcal{S}} = \mathrm{cond}\ ([\![b]\!]_{\mathcal{B}},\ [\![S_1]\!]_{\mathcal{S}},\ [\![S_2]\!]_{\mathcal{S}})$
- $[\![\texttt{while } b \texttt{ do } S]\!]_{\mathcal{S}} = \mathrm{fix}\ F$
  where $F\ g = \mathrm{cond}\ ([\![b]\!]_{\mathcal{B}},\ g \circ [\![S]\!]_{\mathcal{S}},\ \mathrm{id})$

where formally

- $\mathcal{S}$ is the domain **State** $\to$ **State**,
- s is an element in domain **State**, and
- **State** is the mapping from variables to values.

Note that our notations are different from those in the textbook. Also $\mathcal{S}$ (and **State**, when viewed as a function) are continuous instead of being partial; $\perp_{\textbf{State}} = \{(x, \perp) \mid x \text{ is a variable}\}$.

Denotational Semantics of Assignment and Skip

$[\![x := e]\!]_{\mathcal{S}}\ s\ =\ s[x \mapsto [\![e]\!]_{\mathcal{N}}\ s]$

- compute the semantics of expression e at state s, which is an element in domain $\mathcal{N}$;
- map variable x to this element; and
- update the state s with the above mapping.

$[\![\texttt{skip}]\!]_{\mathcal{S}}\ =\ \mathrm{id}$

- id is the identity function: $\mathrm{id}\ x\ =\ x$;
- `skip` has no effect on the state: for all state s, $[\![\texttt{skip}]\!]_{\mathcal{S}}\ s\ =\ s$;
- which means $[\![\texttt{skip}]\!]_{\mathcal{S}}$ is the identity function!

Denotational Semantics of Sequencer and Conditional

$[\![S_1 \ ; \ S_2]\!]_{\mathcal{S}} \ = \ [\![S_2]\!]_{\mathcal{S}} \circ [\![S_1]\!]_{\mathcal{S}}$

- for all state s, $([\![S_2]\!]_{\mathcal{S}} \circ [\![S_1]\!]_{\mathcal{S}})\ s = [\![S_2]\!]_{\mathcal{S}}\ ([\![S_1]\!]_{\mathcal{S}}\ s)$;
- the result is s'' whenever $[\![S_1]\!]_{\mathcal{S}}\ s = s'$ and $[\![S_2]\!]_{\mathcal{S}}\ s' = s''$;
- note that if either s' or s'' is $\perp_{\mathcal{S}}$, the end result is also $\perp_{\mathcal{S}}$.

$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]_{\mathcal{S}} \ = \ \texttt{cond}\ ([\![b]\!]_{\mathcal{B}},\ [\![S_1]\!]_{\mathcal{S}},\ [\![S_2]\!]_{\mathcal{S}})$

- $\texttt{cond}$ is a function in domain $(\textbf{State} \to \mathcal{B}) \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$;
- it is defined by

$$\texttt{cond}\ (p, g_1, g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \mathrm{T} \\ g_2\ s & \text{if } p\ s = \mathrm{F} \end{cases}$$

  where s is a state;
- note that the result is $\perp_{\textbf{State}}$ if $(p\ s = \perp_{\mathcal{B}})$ or $(p\ s = \mathrm{T}$ and $g_1\ s = \perp_{\textbf{State}})$ or $(p\ s = \mathrm{F}$ and $g_2\ s = \perp_{\textbf{State}})$.

Denotational Semantics of While Loop

$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\mathcal{S}} = \text{fix}\ F$
where $F\ g = \texttt{cond}\ ([\![b]\!]_{\mathcal{B}},\ g \circ [\![S]\!]_{\mathcal{S}},\ \text{id})$

- observe that $\texttt{while}\ b\ \texttt{do}\ S$ has the same effect of $\texttt{if}\ b\ \texttt{then}\ (S\ ;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}$
- the two must have the same semantics:
  $[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\mathcal{S}} = \texttt{cond}\ ([\![b]\!]_{\mathcal{B}},\ [\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\mathcal{S}} \circ [\![S]\!]_{\mathcal{S}},\ \text{id})$
- $[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\mathcal{S}}$ is a fixed point of the functional F:
  $F\ g = \texttt{cond}\ ([\![b]\!]_{\mathcal{B}},\ g \circ [\![S]\!]_{\mathcal{S}},\ \text{id})$
- note that F is an element in domain $\mathcal{S} \to \mathcal{S}$, and
- the fixed point function fix is an element in domain $(\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$.

Which Fixed Point? An Exmaple, I

Consider this program: $\texttt{while}\ \neg(\texttt{x} = 0)\ \texttt{do}\ \texttt{skip}$

- the corresponding functional F is defined by

$$F\ g\ s = \texttt{cond}\ (\llbracket\neg(\texttt{x} = 0)\rrbracket_{\mathcal{B}},\ g \circ \llbracket\texttt{skip}\rrbracket_{\mathcal{S}},\ \text{id})\ s$$
$$= \begin{cases} g\ s & \text{if } s\ \texttt{x} \neq 0 \\ s & \text{if } s\ \texttt{x} = 0 \end{cases}$$

- function $g_0\ s = \begin{cases} \perp_{\mathbf{State}} & \text{if } s\ \texttt{x} \neq 0 \\ s & \text{if } s\ \texttt{x} = 0 \end{cases}$ is a fixed point of F,
- so is $g_1\ s = \begin{cases} \perp_{\mathbf{State}} & \text{if } s\ \texttt{x} \notin \{0,1\} \\ s & \text{if } s\ \texttt{x} \in \{0,1\} \end{cases}$ a fixed point, and
- so is $g_2\ s = \begin{cases} \perp_{\mathbf{State}} & \text{if } s\ \texttt{x} \notin \{0,1,2\} \\ s & \text{if } s\ \texttt{x} \in \{0,1,2\} \end{cases}$, and so on.

Clearly we want g_0 as the semantics of this program. How?

Which Fixed Point? An Exmaple, II

What is the least fixed point of functional F:

$$F\ g\ s \quad = \quad \begin{cases} g\ s & \text{if } s\ \mathtt{x} \neq 0 \\ s & \text{if } s\ \mathtt{x} = 0 \end{cases}$$

where $F \in \mathcal{S} \to \mathcal{S}$, $g \in \mathcal{S} = \mathbf{State} \to \mathbf{State}$, and $s \in \mathbf{State}$.
Start with $\bot_{\mathcal{S}}$ which is defined as

$$\bot_{\mathcal{S}}\ x \quad = \quad \bot_{\mathbf{State}}, \quad \text{for all } x \in \mathbf{State}$$

the least fixed point iteration for F will be

$$F^{(0)}(\bot_{\mathcal{S}})\ s \quad = \quad \bot_{\mathcal{S}}\ s = \bot_{\mathbf{State}}$$

$$F^{(1)}(\bot_{\mathcal{S}})\ s \quad = \quad \begin{cases} \bot_{\mathbf{State}} & \text{if } s\ \mathtt{x} \neq 0 \\ s & \text{if } s\ \mathtt{x} = 0 \end{cases}$$

$$F^{(2)}(\bot_{\mathcal{S}})\ s \quad = \quad \begin{cases} \bot_{\mathbf{State}} & \text{if } s\ \mathtt{x} \neq 0 \\ s & \text{if } s\ \mathtt{x} = 0 \end{cases}$$

Clearly the least fixed point is reached at $F^{(2)}(\bot_{\mathcal{S}})$.

Does the Least Fixed Point Always Exist?

Do all **While** programs have well–defined denotational semantics?
We just need to ensure that all semantic functions are continuous!
In particular,

- the semantic functions for all primitive arithmetic and boolean operators are continuous,
- the conditional function is continuous,
- that function composition is continuous, and
- the fixed point function is continuous!

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.
  - What is the range of possible values for x?

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.
  - What is the range of possible values for x?
  - Will the execution `(f x)` terminate if `x` has the value 0?

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.
  - What is the range of possible values for x?
  - Will the execution `(f x)` terminate if `x` has the value 0?
  - Will `(f x)` always terminate for all `x`?

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.
  - What is the range of possible values for x?
  - Will the execution `(f x)` terminate if `x` has the value 0?
  - Will `(f x)` always terminate for all `x`?
- For built-in data types and constants, we may use non-standard domains and their elements. For example, we may interpret `if ... then ... else ...` as

$$\begin{array}{llllll} \textit{if-then-else} & \{\bot\} & X & Y & = & \{\bot\} \\ \textit{if-then-else} & B & X & Y & = & X \cup Y \quad \text{otherwise} \end{array}$$

Other Interpretations

- Sometimes, we are not interested in the precise meaning of a program. Rather, we want a safe approximation which can be more easily computed.
  - What is the range of possible values for x?
  - Will the execution `(f x)` terminate if `x` has the value 0?
  - Will `(f x)` always terminate for all `x`?
- For built-in data types and constants, we may use non-standard domains and their elements. For example, we may interpret `if ... then ... else ...` as

$$\begin{array}{llllll} \textit{if-then-else} & \{\bot\} & X & Y & = & \{\bot\} \\ \textit{if-then-else} & B & X & Y & = & X \cup Y \quad \text{otherwise} \end{array}$$

- However, we need to be precise about these non-standard domains too!

Scott-closed Set

Definition

Let D be a domain. A set $X \subseteq D$ is *Scott–closed* if

1. If $Y \subseteq X$ and Y is directed, then $\bigsqcup Y \in X$.
2. If $x \in X, y \sqsubseteq_D x$, then $y \in X$.

□

Scott-closed Set

Definition

Let D be a domain. A set $X \subseteq D$ is *Scott–closed* if

1. If $Y \subseteq X$ and Y is directed, then $\bigsqcup Y \in X$.
2. If $x \in X, y \sqsubseteq_D x$, then $y \in X$.

□

The least Scott–closed set containing a set Y is written as Y^*.

Hoare Power Domain

Definition

Let D be a domain. Define $\mathrm{P}(D)$ by

1. $\mathrm{P}(D) = \{S \mid \emptyset \neq S \subseteq D,\ S \textit{ is Scott–closed }\}$, and $\bot_{\mathrm{P}(D)} = \{\bot_D\}$.
2. $S \sqsubseteq_{\mathrm{P}(D)} T$ if and only if $S \subseteq T$.

□

Hoare Power Domain

Definition

Let D be a domain. Define $\mathrm{P}(D)$ by

1. $\mathrm{P}(D) = \{S \mid \emptyset \neq S \subseteq D,\ S \textit{ is Scott–closed }\}$, and $\bot_{\mathrm{P}(D)} = \{\bot_D\}$.
2. $S \sqsubseteq_{\mathrm{P}(D)} T$ if and only if $S \subseteq T$.

□

- If D is a domain, then $\mathrm{P}(D)$ is a domain too. It is called the *Hoare power domain*.
- Not only can we apply P to domains, we can apply it to continuous functions as well. For a function $f \in D \to E$, the function $\mathrm{P}(f) \in \mathrm{P}(D) \to \mathrm{P}(E)$ is defined as

$$\mathrm{P}(f)\ (X) = \{f(x) \mid x \in X\}^*$$

Mappings between A Domain and Its Hoare Power Domain

- The function $\{\cdot\} \in D \to \mathrm{P}(D)$ is defined by

$$\{d\} = \{d\}^*$$

- For a function from $\mathrm{P}(D)$ to D, we can use the least upper bound function $\bigsqcup X$, where $X \in \mathrm{P}(D)$, if D is a complete lattice.

Mappings between A Domain and Its Hoare Power Domain

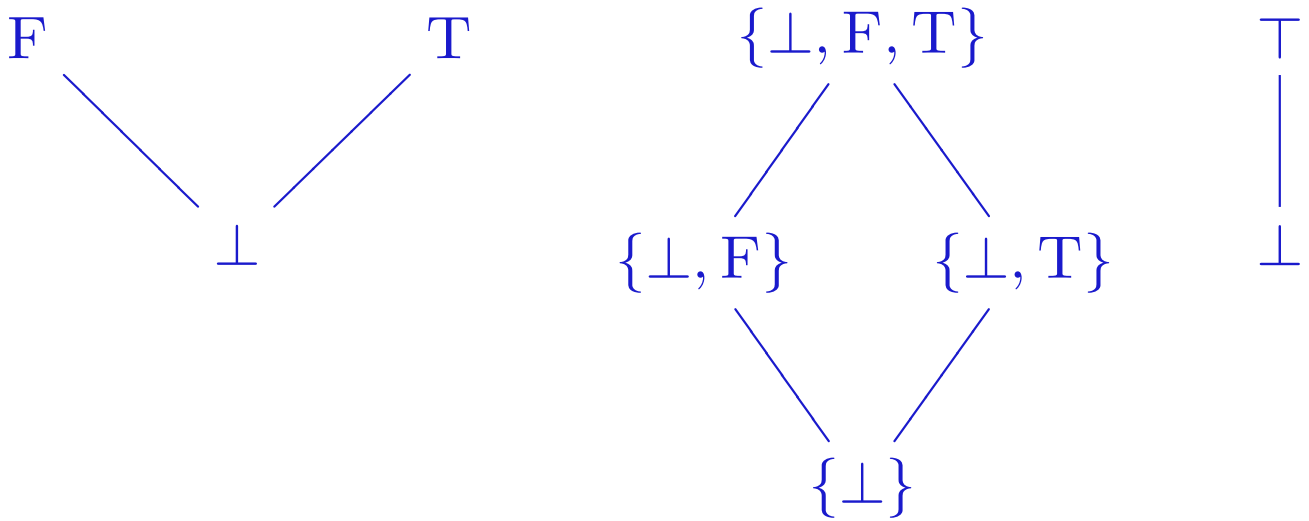
- The function $\{\cdot\} \in D \to \mathrm{P}(D)$ is defined by

$$\{d\} = \{d\}^*$$

- For a function from $\mathrm{P}(D)$ to D, we can use the least upper bound function $\bigsqcup X$, where $X \in \mathrm{P}(D)$, if D is a complete lattice.
- A complete lattice is a poset in which all subsets have a least upper bound.
- Note that both $\{\cdot\}$ and $\bigsqcup$ are continuous.

Concrete Domain, Hoare Power Domain, and Abstract Domain

Take $\mathcal{B}$ as an example. From $\mathcal{B}$ we can build the Hoare power domain $\mathrm{P}(\mathcal{B})$. We too can reduce $\mathcal{B}$ to a two-element abstract domain $\bar{\mathcal{B}} = 2$. Relative to its Hoare power domain $\mathrm{P}(\mathcal{B})$ and its abstract domain $\bar{\mathcal{B}}$, we call $\mathcal{B}$ the concrete domain, or the standard domain.



Collecting Interpretation and Abstract Interpretation

- Instead of using the standard domains, we can map data types to Hoare power domains, and map programs to functions between the Hoare power domains. When so doing, we are performing collecting interpretation of functional programs.

Collecting Interpretation and Abstract Interpretation

- Instead of using the standard domains, we can map data types to Hoare power domains, and map programs to functions between the Hoare power domains. When so doing, we are performing collecting interpretation of functional programs.
- Instead of using the standard domains, we can map data types to abstract domains, and map programs to functions between the abstract domains. When so doing, we are performing abstract interpretation of functional programs.

Collecting Interpretation and Abstract Interpretation

- Instead of using the standard domains, we can map data types to Hoare power domains, and map programs to functions between the Hoare power domains. When so doing, we are performing collecting interpretation of functional programs.
- Instead of using the standard domains, we can map data types to abstract domains, and map programs to functions between the abstract domains. When so doing, we are performing abstract interpretation of functional programs.
- Abstract interpretation is a useful technique for program analysis, but we need to relate the three semantics: standard interpretation, collecting interpretation, and abstract interpretation.

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.
- In call-by-value functional languages, function application is strict: computation always diverges if an argument diverges.

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.
- In call-by-value functional languages, function application is strict: computation always diverges if an argument diverges.
- In call-by-name/need functional languages, function application is non-strict: computation may terminate even if all arguments diverge.

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.
- In call-by-value functional languages, function application is strict: computation always diverges if an argument diverges.
- In call-by-name/need functional languages, function application is non-strict: computation may terminate even if all arguments diverge.
- In O'Caml, the evaluation for `zero` will diverge.

```
let rec loop  x = loop x
let     const y = 0
let     zero    = const (loop true)
```

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.
- In call-by-value functional languages, function application is strict: computation always diverges if an argument diverges.
- In call-by-name/need functional languages, function application is non-strict: computation may terminate even if all arguments diverge.
- In O'Caml, the evaluation for `zero` will diverge.

```
let rec loop  x = loop x
let     const y = 0
let     zero    = const (loop true)
```

- In Haskell, `zero` evaluates to 0.

```
loop  x = loop x
const y = 0
zero    = const (loop True)
```

Strictness Analysis

- A function $f \in D \to D'$ is *strict* if and only if $f(\bot_D) = \bot_{D'}$.
- In call-by-value functional languages, function application is strict: computation always diverges if an argument diverges.
- In call-by-name/need functional languages, function application is non-strict: computation may terminate even if all arguments diverge.
- In O'Caml, the evaluation for `zero` will diverge.

```
let rec loop  x = loop x
let     const y = 0
let     zero    = const (loop true)
```

- In Haskell, `zero` evaluates to 0.

```
loop  x = loop x
const y = 0
zero    = const (loop True)
```

- For call-by-name/need languages, strictness analysis is used to determine if functions in a program are strict or not.

Language Constructs May Be Non-strict

For the `if ... then ... else ...` language construct (in both call-by-value and call-by-name/need languages), we define function $\mathit{if\text{-}then\text{-}else} \in \mathcal{B} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$ below as its semantics:

$$\begin{array}{llllll} \mathit{if\text{-}then\text{-}else} & \bot & x & y & = & \bot \\ \mathit{if\text{-}then\text{-}else} & \mathrm{T} & x & y & = & x \\ \mathit{if\text{-}then\text{-}else} & \mathrm{F} & x & y & = & y \end{array}$$

Language Constructs May Be Non-strict

For the `if ... then ... else ...` language construct (in both call-by-value and call-by-name/need languages), we define function $\mathit{if\text{-}then\text{-}else} \in \mathcal{B} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$ below as its semantics:

$$\begin{array}{llllll} \mathit{if\text{-}then\text{-}else} & \bot & x & y & = & \bot \\ \mathit{if\text{-}then\text{-}else} & \mathrm{T} & x & y & = & x \\ \mathit{if\text{-}then\text{-}else} & \mathrm{F} & x & y & = & y \end{array}$$

Note that *if-then-else* is strict in its first argument, non-strict in its third argument if its first argument is T, and non-strict in its second argument if its first argument is F.

Abstract Interpretation for Strictness Analysis

- For domains such as $\mathcal{B}$ and $\mathcal{N}$, we now use 2 as the abstract domain with the intention that $\bot$ denotes non-termination while $\top$ denotes values that may or may not terminate.

Abstract Interpretation for Strictness Analysis

- For domains such as $\mathcal{B}$ and $\mathcal{N}$, we now use 2 as the abstract domain with the intention that $\bot$ denotes non-termination while $\top$ denotes values that may or may not terminate.
- For domains such as $\mathcal{N} \to \mathcal{N}$, we now use the abstract domain $2 \to 2$ below to denote all possible strictness properties for all elements in $\mathcal{N} \to \mathcal{N}$ (which are continuous functions):

$$\{(\bot, \top),\ (\top, \top)\}$$
$$|$$
$$\{(\bot, \bot),\ (\top, \top)\}$$
$$|$$
$$\{(\bot, \bot),\ (\top, \bot)\}$$

Abstract Interpretation for Strictness Analysis, Continued

- Constant like *if-then-else* is now a function in the domain $2 \to 2 \to 2 \to 2$. It is defined by

$$\textit{if-then-else}\ b\ x\ y \quad = \quad b \sqcap (x \sqcup y)$$

  Note: An *if-then-else* expression will not terminate if the conditional part b will not terminate, or if both branches x and y will not terminate.
- For a user-defined term, construct an abstract semantic equation based on its definition, and from the abstract semantics of existing terms and constants.
- If the definition is recursive, compute the least fixed point.

Abstract Semantics for Strictness Analysis

$$\begin{aligned} \mathit{if\text{-}then\text{-}else}\ b\ x\ y &= b \sqcap (x \sqcup y) \\ \mathit{eq}\ x\ y &= x \sqcap y \\ \mathit{not}\ x &= x \\ \mathit{minus}\ x\ y &= x \sqcap y \\ \mathit{times}\ x\ y &= x \sqcap y \end{aligned}$$

The Factorial Program, Revisited

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

The Factorial Program, Revisited

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

We now compose the following function $f \in (2 \to 2) \to (2 \to 2)$:

$$\begin{aligned} f\ fac\ n &= (n \sqcap \top) \sqcap (\top \sqcup (n \sqcap (fac\ (n \sqcap \top)))) \\ &= n \end{aligned}$$

The Factorial Program, Revisited

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

We now compose the following function $f \in (2 \to 2) \to (2 \to 2)$:

$$\begin{aligned} f\ fac\ n &= (n \sqcap \top) \sqcap (\top \sqcup (n \sqcap (fac\ (n \sqcap \top)))) \\ &= n \end{aligned}$$

Start with $\bot_{2\to 2} = \{(\bot, \bot),\ (\top, \bot)\}$, the least fixed point iteration becomes

$$\begin{aligned} f^{(0)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \bot)\} \\ f^{(1)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \top)\} \\ f^{(2)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \top)\} \end{aligned}$$

The Factorial Program, Revisited

For the following program `fac`:

```
let rec fac n = if n = 0 then 1 else n * (fac (n - 1))
```

We now compose the following function $f \in (2 \to 2) \to (2 \to 2)$:

$$\begin{aligned} f\ fac\ n &= (n \sqcap \top) \sqcap (\top \sqcup (n \sqcap (fac\ (n \sqcap \top)))) \\ &= n \end{aligned}$$

Start with $\bot_{2\to 2} = \{(\bot, \bot),\ (\top, \bot)\}$, the least fixed point iteration becomes

$$\begin{aligned} f^{(0)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \bot)\} \\ f^{(1)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \top)\} \\ f^{(2)}(\bot_{2\to 2}) &= \{(\bot, \bot),\ (\top, \top)\} \end{aligned}$$

We reach the least fixed point at $\{(\bot, \bot),\ (\top, \top)\}$. That is, *fac* is strict. When *fac* is applied to a non-terminating argument, it will diverge. When it is applied to others, it may or may not diverge.

Formalizing Abstract Interpretation

Let $abs \in D \to \bar{D}$ be an (intuitive) abstraction function that maps from a concrete domain D to an abstract domain $\bar{D}$. We can define on the Hoare power domain the abstraction and corresponding concretization functions

$$\begin{aligned} Abs &\in \mathrm{P}(D) \to \mathrm{P}(\bar{D}) \\ Conc &\in \mathrm{P}(\bar{D}) \to \mathrm{P}(D) \end{aligned}$$

Formalizing Abstract Interpretation

Let $abs \in D \to \bar{D}$ be an (intuitive) abstraction function that maps from a concrete domain D to an abstract domain $\bar{D}$. We can define on the Hoare power domain the abstraction and corresponding concretization functions

$$\begin{aligned} Abs &\in \mathrm{P}(D) \to \mathrm{P}(\bar{D}) \\ Conc &\in \mathrm{P}(\bar{D}) \to \mathrm{P}(D) \end{aligned}$$

by

$$\begin{aligned} Abs\,(S) &= \mathrm{P}(abs)\,(S), \text{ where } \mathrm{P}(f)\,(X) = \{f(x) \mid x \in X\}^* \\ Conc\,(S) &= \bigcup \{T \mid Abs\,(T) \sqsubseteq_{\mathrm{P}(\bar{D})} S, T \in \mathrm{P}(D)\} \end{aligned}$$

Strictness Analysis, Revisited

For strictness analysis, it is straightforward to define $abs \in D \to 2$ for a concrete (basis) domain D as

$$abs\,(d) = \begin{cases} \bot_2 & \text{if } d = \bot_D \\ \top_2 & \text{if } d \neq \bot_D \end{cases}$$

Strictness Analysis, Revisited

For strictness analysis, it is straightforward to define $abs \in D \to 2$ for a concrete (basis) domain D as

$$abs\,(d) = \begin{cases} \bot_2 & \text{if } d = \bot_D \\ \top_2 & \text{if } d \neq \bot_D \end{cases}$$

Then we have

$$Abs\,(X) = \begin{cases} \{\bot_2\} & \text{if } X = \{\bot_D\} \\ 2 & \text{otherwise} \end{cases}$$

$$Conc\,(X) = \begin{cases} \{\bot_D\} & \text{if } X = \{\bot_2\} \\ D & \text{if } X = 2 \end{cases}$$

Formalizing Abstract Interpretation, Continued

Let $f \in C \to D$ be a function, we define the abstraction function $abs \in (C \to D) \to (\bar{C} \to \bar{D})$ as

$$abs\,(f) \;=\; \bigsqcup \circ\; Abs \circ \mathrm{P}(f) \circ Conc \circ \{\cdot\}$$

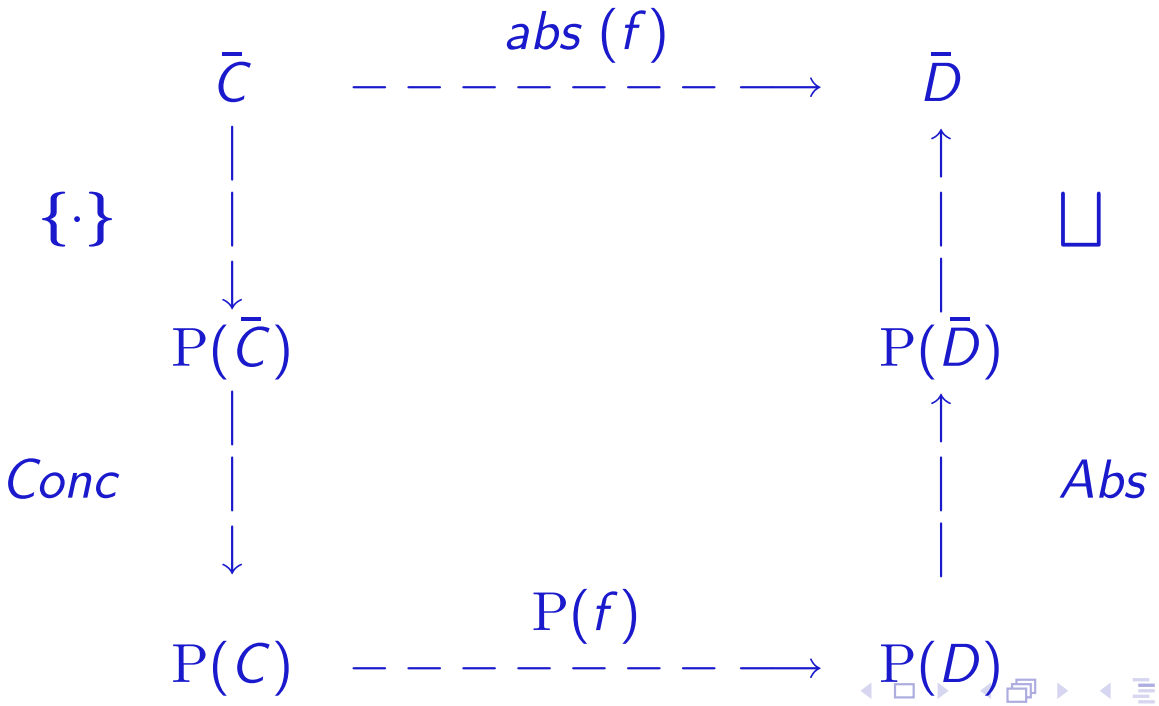
so that $abs\,(f) \in \bar{C} \to \bar{D}$ is an abstraction of f.

Formalizing Abstract Interpretation, Continued

Let $f \in C \to D$ be a function, we define the abstraction function $abs \in (C \to D) \to (\bar{C} \to \bar{D})$ as

$$abs\,(f) \;=\; \bigsqcup \circ Abs \circ \mathrm{P}(f) \circ Conc \circ \{\cdot\}$$

so that $abs\,(f) \in \bar{C} \to \bar{D}$ is an abstraction of f. This is illustrated by the following diagram:



Formalizing Abstract Interpretation, Continued

This definition of abstraction for function is safe.

Theorem (Burn, Hankin, Abramsky)

Let function $f \in C \to D$. Then

$$\mathrm{P}(f) \sqsubseteq_{\mathrm{P}(C)\to\mathrm{P}(D)} \mathsf{Conc} \circ \mathrm{P}(\mathsf{abs}\,(f)) \circ \mathsf{Abs}$$

□

Formalizing Abstract Interpretation, Continued

This definition of abstraction for function is safe.

Theorem (Burn, Hankin, Abramsky)

Let function $f \in C \to D$. Then

$$\mathrm{P}(f) \sqsubseteq_{\mathrm{P}(C)\to\mathrm{P}(D)} \mathsf{Conc} \circ \mathrm{P}(\mathsf{abs}\,(f)) \circ \mathsf{Abs}$$

□

This is illustrated by the following diagram:

$$\begin{array}{ccc} \mathrm{P}(C) & \xrightarrow{\quad \mathrm{P}(f) \quad} & \mathrm{P}(D) \\ \Big\downarrow \mathsf{Abs} & & \Big\uparrow \mathsf{Conc} \\ \mathrm{P}(\bar{C}) & \xrightarrow{\quad \mathrm{P}(\mathit{abs}\,(f)) \quad} & \mathrm{P}(\bar{D}) \end{array}$$

Strictness Analysis (Burn, Hankin, and Abramsky)

The following abstractions for built-in functions are safe.

1. If f is strict in all of its n arguments, then define

$$(abs\,(f))\ x_1\ x_2\ \ldots\ x_n\ =\ x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n$$

2. Let $\mathit{if\text{-}then\text{-}else} \in \mathcal{B} \to D \to D \to D$ be the standard semantics of the "if then else" construct. Then define

$$(abs\,(\mathit{if\text{-}then\text{-}else}))\ x\ y\ z\ =\ x\ \mathbf{and}\ (y \sqcup z),$$

where $\mathbf{and} \in 2 \to \bar{D} \to \bar{D}$ is defined by

$$\begin{aligned} \bot\ \mathbf{and}\ e &= \bot_{\bar{D}} \\ \top\ \mathbf{and}\ e &= e \end{aligned}$$

3. If $f \in D \to D$, then define

$$abs\,(fix\,(f)) = fix\,(abs\,(f))$$