Semantics of Functional Programming Languages

Tyng–Ruey Chuang

2008 Formosan Summer School
on Logic, Language, and Computation
June 30 – July 11, 2008

This course note . . .

- . . . is prepared for the *2008 Formosan Summer School on Logic, Language, and Computation (FLOLAC)* (held in Taipei, Taiwan),
- . . . is made available from the FLOLAC '08 web site:

  `http://flolac.iis.sinica.edu.tw/flolac08/`

-

Course outline

Unit 1. Basic domain theory.

Unit 2. Denotational semantics of functional programs.

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

1 Basic Domain Theory

1.1 Giving Meaning to Programs

Syntax and Semantics

- Syntax is about the form of the sentences in a language.
- Semantics is about the meaning of the 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 from 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.
- Operational semantics: A program means what the 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.

What's So Special about *Programming* Languages?

- A programming language is a means to convey, via programs, meaning from a programmer to machines, to other programmers, and to the programmer herself.
- Programs can be complex, even for a simple programming language.
- Programming languages themselves can be complex too.
- Programs are written by many people and executed on different machines.
- Programs may or may not terminate, or may terminate abnormally.
- *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))
```

Possible answers:

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

Which interpretation is more 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?

- $g_3(n) = \quad \uparrow$

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

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 $f = \{(d, e) \mid f(d) = e, e \text{ is defined}\}$.

$$\begin{array}{rclrcl}
g_1(n) & = & \begin{cases} \mathrm{T} & \text{if } n \text{ is even} \\ \mathrm{F} & \text{if } n \text{ is odd} \end{cases} & g_1 & = & \begin{array}{l}\{(2n, \mathrm{T}) \mid n \geq 0\} \cup \\ \{(2n+1, \mathrm{F}) \mid n \geq 0\}\end{array} \\
g_2(n) & = & \begin{cases} \mathrm{F} & \text{if } n \text{ is even} \\ \mathrm{T} & \text{if } n \text{ is odd} \end{cases} & g_2 & = & \begin{array}{l}\{(2n, \mathrm{F}) \mid n \geq 0\} \cup \\ \{(2n+1, \mathrm{T}) \mid n \geq 0\}\end{array} \\
g_3(n) & = & \uparrow & g_3 & = & \emptyset \\
g_4(n) & = & \begin{cases} \mathrm{T} & \text{if } n = 0 \\ \mathrm{F} & \text{if } n = 1 \\ \perp & \text{otherwise} \end{cases} & g_4 & = & \{(0, \mathrm{T}),\ (1, \mathrm{F})\} \\
g_5(n) & = & \begin{cases} \mathrm{F} & \text{if } n = 0 \\ \mathrm{T} & \text{if } n = 1 \\ \perp & \text{otherwise} \end{cases} & g_5 & = & \{(0, \mathrm{F}),\ (1, \mathrm{T})\}
\end{array}$$

1.2 Semantic Domains

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 $N = \{\ldots, 2, -1, 0, 1, 2, \ldots\}$ 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 executions of 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

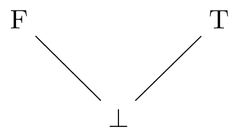
To address non-termination,

- 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.

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 $\bot \sqsubseteq \mathrm{T}$ and $\bot \sqsubseteq \mathrm{F}$. But $\mathrm{T} \not\sqsubseteq \mathrm{F}$ and $\mathrm{F} \not\sqsubseteq \mathrm{T}$.
- Domain $\mathcal{B}$ illustrated:



Partially Ordered Set (poset)

Definition 1. 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 $x \equiv y$.
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 2. 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)

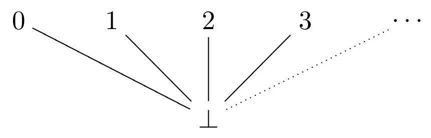
Definition 3. Let D be a poset. D is a *complete partial order* (cpo) if

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) $\bigsqcup X \in D$.

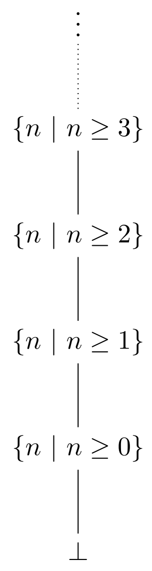
□

- That is, for a cpo D, and an approximation $X \subseteq D$, all the computations in X progress to an unique element (lub) in D (though not necessarily in X).
- We use cpo as the domain of our study on programming language semantics. The terms cpo and domains will be used interchangeably from now on.
- The subscript D in $\sqsubseteq_D$ and $\bot_D$ is often omitted if it is clear.

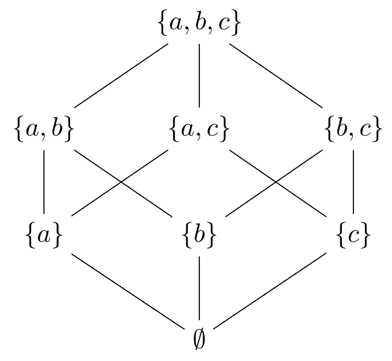
Domain $\mathcal{N}$



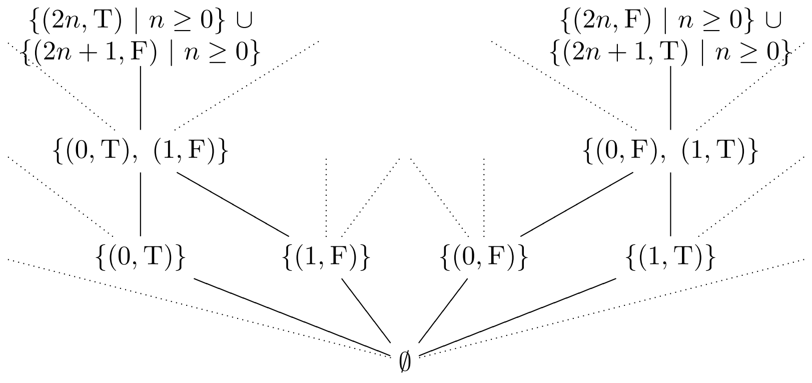
Another View of Natural Numbers



A Domain Built from Subsets



A Domain of Partial Functions


$\{(2n, \mathrm{T}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{F}) \mid n \geq 0\}$
$\{(2n, \mathrm{F}) \mid n \geq 0\} \cup \{(2n+1, \mathrm{T}) \mid n \geq 0\}$
$\{(0, \mathrm{T}),\ (1, \mathrm{F})\}$
$\{(0, \mathrm{F}),\ (1, \mathrm{T})\}$
$\{(0, \mathrm{T})\}$
$\{(1, \mathrm{F})\}$
$\{(0, \mathrm{F})\}$
$\{(1, \mathrm{T})\}$
$\emptyset$

Domain Lifting

Definition 4. Let D be a domain. Define poset lift(D) by

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

□

- If D is a domain then lift(D) forms a domain.
- lift(D) is called the *lifted* domain of D.

Domain *1* and Domain *2*

Example 5. Let *1* be the poset $\{\perp\}$ where $\perp \sqsubseteq_{\mathit{1}} \perp$. *1* is a domain. □

Example 6. Let $\mathit{2} = \mathrm{lift}(\mathit{1})$. Then *2* is a domain. *2* has only two elements, $\perp$ and $\top$, and $\perp \sqsubseteq_{\mathit{2}} \top$. □

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

$\sqcup$	$\perp$	$\top$
$\perp$	$\perp$	$\top$
$\top$	$\top$	$\top$

$\sqcap$	$\perp$	$\top$
$\perp$	$\perp$	$\perp$
$\top$	$\perp$	$\top$

(Look familiar?)

The Sum of Two Domains

Definition 7. Let D and D' be domains. Define poset $D + D'$ by

1. $D + D' \ = \ D \ \cup \ D' \ \cup \ \{\perp_{D+D'}\}$, where the elements in D are made distinct from the elements in D'. $\perp_{D+D'} \ \notin \ D \ \cup \ D'$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x \ = \ \perp_{D+D'}$ 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 8. Let D and D' be domains. Define poset $D \oplus D'$ by

1. $D \oplus D' \;=\; D \,\cup\, D' \,\cup\, \{\perp_{D\oplus D'}\}$, where the elements in D are made distinct from the elements in D', except $\perp_D$ and $\perp_{D'}$. $\perp_{D\oplus D'} \;=\; \perp_D \;=\; \perp_{D'}$.
2. $x \sqsubseteq_{D+D'} y$ if and only if $x \;=\; \perp_{D\oplus D'}$ 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 9. Let D and D' be domains. Define $D \times D'$ by

1. $D \times D' \;=\; \{\; \langle d,\, d' \rangle \mid d \in D,\; d' \in D' \;\}$, $\perp_{D\times D'} \;=\; \langle \perp_D,\, \perp_{D'} \rangle$.
2. $\langle d_1,\, d_1' \rangle \sqsubseteq_{D\times D'} \langle d_2,\, d_2' \rangle$ iff $d_1 \sqsubseteq_D d_2$ 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 10. Let D and D' be domains. Define $D \otimes D'$ by

1. $D \otimes D' \;=\; \{\; \langle d,\, d' \rangle \mid d \in D,\; d' \in D' \;\}$. $\perp_{D\otimes D'} \;=\; \langle \perp_D,\, d' \rangle \;=\; \langle d,\, \perp_{D'} \rangle$ for any $d \in D$ and $d' \in D'$.
2. $\langle d_1,\, d_1' \rangle \sqsubseteq_{D\otimes D'} \langle d_2,\, d_2' \rangle$ iff $\langle d_1,\, d_1' \rangle \;=\; \perp_{D\otimes D'}$, 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 11. 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 $f(\perp_D) \;=\; \perp_{D'}$.

□

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

1. $D \to D' \ = \ \{f \mid f$ is a continuous function from D 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

Theorem 13 (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. For F, define function $F^{\sqcup}$ by

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

Since F is a directed set, we know that, for any $d \in D$, $\{f(d) \mid f \in F\} \subseteq D'$ is a directed set 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

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

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

□

From now on, we write $\bigsqcup F$ to denote 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?

- 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 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 14. 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: $\perp$, F, and T.
- $\perp$ is the least fixed point of f.

The Least Fixed Point Theorem

Theorem 15 (Kleene). *Let D be a domain.*

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

□

Proof. 1. For a function f, define a directed set $X_f \subseteq D$ by

$$X_f = \{\perp_D,\ \ f(\perp_D),\ \ f(f(\perp_D)),\ \ \ldots,\ \ f^{(n)}(\perp_D),\ \ \ldots\}$$

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 above. (To be continued) □

The Least Fixed Point Theorem, Continued

Proof (Continued). Moreover, suppose that d too is a fixed point of f. Then

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

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

2. Define function *fix* by

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

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

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

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.

- 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, $\mathit{fix}(f) = f(\mathit{fix}(f))$. This matches the recursive definition of `g`.

- We then assign $g = \mathit{fix}(f) \in \mathcal{N} \to \mathcal{B}$ as the meaning of `g`.

2 Denotational Semantics of Functional Programs

2.1 Denotational Semantics

Compose Meaning for Programs

- For every data type, find a domain whose elements correspond to values of the type, computationally.
  - Type `unit` is domain $\mathit{2}$, 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 $\top \in \mathit{2}$, `true` is mapped to $\mathrm{T} \in \mathcal{B}$, `not` is mapped to a function $\mathit{not} \in B \to B$ such that $\mathit{not}(\bot) = \bot$, $\mathit{not}(\mathrm{F}) = \mathrm{T}$, and $\mathit{not}(\mathrm{T}) = \mathrm{F}$, etc.
  - We write $[\![\texttt{()}]\!]_{\mathit{2}} = \top$, $[\![\texttt{true}]\!]_{\mathcal{B}} = \mathrm{T}$, and $[\![\texttt{not}]\!]_{\mathcal{B}\to\mathcal{B}} = \mathit{not}$, etc.

- For a user-defined term, construct a semantic equation based on its definition, and from 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))
```

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

$$\begin{aligned} &f\ g\ n = \textit{if-then-else} \\ &\qquad (\textit{eq}\ (\textit{mod}\ n\ 2)\ 0)\ (\textit{not}\ (g\ (\textit{plus}\ n\ 1)))\ (\textit{not}\ (g\ (\textit{minus}\ n\ 1))) \end{aligned}$$

where functions

$$\begin{array}{rcl} \textit{if-then-else} & \in & \mathcal{B} \to \mathcal{N} \to \mathcal{N} \\ \textit{eq} & \in & \mathcal{N} \to \mathcal{N} \to \mathcal{B} \\ \textit{not} & \in & \mathcal{B} \to \mathcal{B} \\ \textit{mod}, \textit{plus}, \textit{minus} & \in & \mathcal{N} \to \mathcal{N} \to \mathcal{N} \end{array}$$

are defined by the following equations (with pseudo pattern-matching):

$$\begin{array}{llllll} \textit{if-then-else} & \bot & x & y & = & \bot \\ \textit{if-then-else} & \mathrm{T} & x & y & = & x \\ \textit{if-then-else} & \mathrm{F} & x & y & = & y \\ & & & & & \\ \textit{eq} & \bot & y & & = & \bot \\ \textit{eq} & x & \bot & & = & \bot \\ \textit{eq} & x & y & & = & \mathrm{T} \quad \text{where } x = y \neq \bot \\ \textit{eq} & x & y & & = & \mathrm{F} \quad \text{otherwise} \\ & & & & & \\ \textit{not} & \bot & & & = & \bot \\ \textit{not} & \mathrm{F} & & & = & \mathrm{T} \\ \textit{not} & \mathrm{T} & & & = & \mathrm{F} \\ \ldots & & & & & \\ \textit{minus} & \bot & y & & = & \bot \\ \textit{minus} & x & \bot & & = & \bot \\ \textit{minus} & x & y & & = & \bot \quad \text{where } x < y \\ \textit{minus} & x & y & & = & x - y \quad \text{otherwise} \end{array}$$

The Least Fixed Point Iteration

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

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

However, $f(\bot_{\mathcal{N}\to\mathcal{B}})$ computes to

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

This is simplified to

$$g\ (n)\ =\ \bot$$

That is, we have reached the least fixed point of f. We conclude that

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

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))
```

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

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

Start with $\perp_{\mathcal{N}\to\mathcal{B}} = \{(n, \perp) \mid n \in \mathcal{N}\}$, the least fixed point iteration will be

$$\begin{array}{rcl}
f^{(0)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(n,\perp) \mid n \in \mathcal{N}\} \\
f^{(1)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(0,1)\} \cup \{(n,\perp) \mid n > 0\} \\
f^{(2)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(0,1),\ (1,1)\} \cup \{(n,\perp) \mid n > 1\} \\
f^{(3)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(0,1),\ (1,1),\ (2,2)\} \cup \{(n,\perp) \mid n > 2\} \\
f^{(4)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(0,1),\ (1,1),\ (2,2),\ (3,6)\} \cup \{(n,\perp) \mid n > 3\} \\
\dots & & \\
f^{(k+1)}(\perp_{\mathcal{N}\to\mathcal{B}}) & = & \{(n,n!) \mid n \le k\} \cup \{(n,\perp) \mid n > k\} \\
\dots & &
\end{array}$$

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

$$\begin{array}{l}
f\ (even,\ odd) = \\
\quad (\{(n, \textit{if-then-else}\ (eq\ n\ 0)\ \mathrm{T}\ (odd\ (minus\ n\ 1))) \mid n \in \mathcal{N}\}, \\
\quad\ \ \{(n, \textit{if-then-else}\ (eq\ n\ 0)\ \mathrm{F}\ (even\ (minus\ n\ 1))) \mid n \in \mathcal{N}\})
\end{array}$$

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

$$\begin{array}{rcl}
even\ (n) & = & \textit{if-then-else}\ (eq\ n\ 0)\ \mathrm{T}\ (odd\ (minus\ n\ 1)) \\
odd\ (n) & = & \textit{if-then-else}\ (eq\ n\ 0)\ \mathrm{F}\ (even\ (minus\ n\ 1))
\end{array}$$

Dealing With Mutual Recursion, Continued

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

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

2.2 Non-standard Semantics

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 value 0?
  - Will `(f x)` always terminate?
- 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 precise about these non-standard domains too!

Scott-closed Set

Definition 16. 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 17. 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

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

Mapping between A Domain and Its Hoare Power Domain

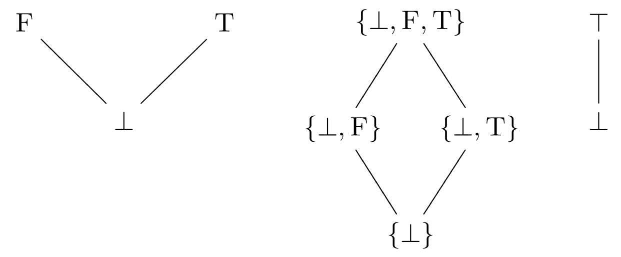
- The function $\boldsymbol{\{}\cdot\boldsymbol{\}} \in D \to \mathrm{P}(D)$ is defined by

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

- For the function from $\mathrm{P}(D)$ to D, we use the least upper bound function $\bigsqcup X$, where $X \in \mathrm{P}(D)$.
- Note that both $\boldsymbol{\{}\cdot\boldsymbol{\}}$ and $\bigsqcup X$ are well-defined and continuous.

Domain, Hoare Power Domain, and Abstract Domain

Take $\mathcal{B}$ as an example. From $\mathcal{B}$ we can build a Hoare power domain $\mathrm{P}(\mathcal{B})$. We too can relate $\mathcal{B}$ to a two-element abstract domain $\bar{\mathcal{B}} = \mathit{2}$.



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 $f(\perp_D) \ = \ \perp_{D'}$.
- 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 Can Be Non-strict

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

$$\begin{array}{lllll} \textit{if-then-else} & \bot & x & y & = & \bot \\ \textit{if-then-else} & \mathrm{T} & x & y & = & x \\ \textit{if-then-else} & \mathrm{F} & x & y & = & y \end{array}$$

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 all flat domains, such as $\mathcal{B}$ and $\mathcal{N}$, we now use $\mathit{2}$ 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 $\mathit{2} \to \mathit{2}$ below

$$\begin{tikzcd} \{(\bot,\top),\ (\top,\top)\} \arrow[d, dash] \\ \{(\bot,\bot),\ (\top,\top)\} \arrow[d, dash] \\ \{(\bot,\bot),\ (\top,\bot)\} \end{tikzcd}$$

Abstract Interpretation for Strictness Analysis, Continued

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

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

- 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.

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 $f \in (\mathit{2} \to \mathit{2}) \to (\mathit{2} \to \mathit{2})$:

$$f\ \textit{fac}\ n \quad = \quad (n \sqcap \top) \sqcap (\top \sqcup (n \sqcap (\textit{fac}\ (n \sqcap \top))))$$

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

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

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 other arguments, it may or may not diverge.