Denotational Semantics

CS 6371: Advanced Programming Languages

Kevin W. Hamlen

February 13, 2024

Operational Semantics

- Operational Semantics: Two Styles
  1. Large-step: Programs "converge" to answers (otherwise "diverge")
  2. Small-step: Programs are state transformers (computational steps)
- Philosophy: Either way, programs are defined by the behavior of an abstract machine.
- Math connection: Operational semantics define an inference logic for computations (new inference rules and axioms of logic)

Denotational Semantics

- Denotational Semantics
  - Philosophy: Programs denote mathematical objects (functions, numbers, etc.)
  - Goal: Define a function that converts each program into the mathematical object it denotes
- Advantages:
  - No extensions to the foundations of logic
  - Excellent for verifying code that performs scientific computations
- Disadvantages:
  - Hard to reason about non-termination (like large-step operational)
  - Mathematical object is often huge and complex

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

$$\mathcal{A} : a \to ?$$

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

$$\mathcal{A} : a \to \mathbb{Z}$$

This is fine for expressions like $3 + 2 * 10$ (denotes 23), but what about expressions containing variables (e.g., $\mathtt{x} + 1$)?

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \to \mathbb{Z})$$

Idea: Arithmetic expressions in our language denote functions from variable values (i.e., stores) to integers.

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \to \mathbb{Z})$$

New problem: What if a contains a variable that is undefined in $\sigma \in \Sigma$ (uninitialized variable)?

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \rightharpoonup \mathbb{Z})$$

Solution: Arithmetic expressions in our language actually denote *partial functions* from stores to integers.

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \rightharpoonup \mathbb{Z})$$

Important: Valuation functions are **total** in their first arguments. (Every program must have a denotation, even if its denotation is a partial function.)

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \rightharpoonup \mathbb{Z})$$
$$\mathcal{B} : b \to (\Sigma \rightharpoonup \{T, F\})$$

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \rightharpoonup \mathbb{Z})$$
$$\mathcal{B} : b \to (\Sigma \rightharpoonup \{T, F\})$$
$$\mathcal{C} : c \to ?$$

Semantic Domains

First step: Define the language's **semantic domain**.

Definition (valuation function): A function that accepts a program (or expression or any other sub-language in the syntax of programs) and returns the mathematical object it denotes.

Definition (semantic domain): The type signatures (domains and ranges) of the valuation functions.

Semantic Domain of SIMPL

$$\Sigma = v \rightharpoonup \mathbb{Z}$$
$$\mathcal{A} : a \to (\Sigma \rightharpoonup \mathbb{Z})$$
$$\mathcal{B} : b \to (\Sigma \rightharpoonup \{T, F\})$$
$$\mathcal{C} : c \to (\Sigma \rightharpoonup \Sigma)$$

Valuation Functions

Notation: Customary to write the code argument to a valuation function enclosed within **semantic brackets** rather than parentheses:

$$\mathcal{A}[\![\texttt{x + 1}]\!] = \cdots$$

(In LaTeX, write "`\mathcal{A}[\![ ··· ]\!]`".)

$\mathcal{A}[\![a]\!]$ and σ are partial functions, so we can use set notation

$$\mathcal{A}[\![a]\!] = \{(\sigma, n+1) \mid (\texttt{x}, n) \in \sigma\}$$

or we can use function notation

$$\mathcal{A}[\![a]\!](\sigma) = \sigma(\texttt{x}) + 1$$

Need to be able to read and write both, and convert between them as needed!

Since we are functional programmers, it's customary to drop paretheses around arguments unless needed for grouping.

$$\mathcal{A}[\![a]\!]\sigma \qquad \text{instead of} \qquad \mathcal{A}[\![a]\!](\sigma)$$

Arithmetic Expression Valuation

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$

Arithmetic Expression Valuation

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![v]\!] = \{(\sigma, n) \mid (v, n) \in \sigma\}$$
$$\mathcal{A}[\![a_1 + a_2]\!] = \{(\sigma, n_1 + n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\}$$

Arithmetic Expression Valuation

$$
\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![v]\!] &= \{(\sigma, n) \mid (v, n) \in \sigma\} \\
\mathcal{A}[\![a_1 \texttt{+} a_2]\!] &= \{(\sigma, n_1 + n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\} \\
\mathcal{A}[\![a_1 \texttt{-} a_2]\!] &= \{(\sigma, n_1 - n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\} \\
\mathcal{A}[\![a_1 \texttt{*} a_2]\!] &= \{(\sigma, n_1 n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\}
\end{aligned}
$$

Arithmetic Expression Valuation

$$
\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![v]\!] &= \{(\sigma, n) \mid (v, n) \in \sigma\} \\
\mathcal{A}[\![a_1 \texttt{+} a_2]\!] &= \{(\sigma, n_1 + n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\} \\
\mathcal{A}[\![a_1 \texttt{-} a_2]\!] &= \{(\sigma, n_1 - n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\} \\
\mathcal{A}[\![a_1 \texttt{*} a_2]\!] &= \{(\sigma, n_1 n_2) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], (\sigma, n_2) \in \mathcal{A}[\![a_2]\!]\}
\end{aligned}
$$

Or (equivalently),

$$
\begin{aligned}
\mathcal{A}[\![n]\!]\sigma &= n \\
\mathcal{A}[\![v]\!]\sigma &= \sigma\, v \\
\mathcal{A}[\![a_1 \texttt{+} a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{A}[\![a_1 \texttt{-} a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{A}[\![a_1 \texttt{*} a_2]\!]\sigma &= (\mathcal{A}[\![a_1]\!]\sigma)(\mathcal{A}[\![a_2]\!]\sigma)
\end{aligned}
$$

Boolean Expression Valuation

Valuation function $\mathcal{B}$ for boolean expressions is similar.
See online notes for its full definition.

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$
$$\mathcal{C}[\![c_1 \,;\, c_2]\!] = ?$$

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$
$$\mathcal{C}[\![c_1\,;c_2]\!] = \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\}$$

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$
$$\mathcal{C}[\![c_1 \,\texttt{;}\, c_2]\!] = \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\}$$
$$\mathcal{C}[\![v \,\texttt{:=}\, a]\!] = ?$$

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$
$$\mathcal{C}[\![c_1 \,\texttt{;}\, c_2]\!] = \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\}$$
$$\mathcal{C}[\![v \,\texttt{:=}\, a]\!] = \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\}$$

Command Valuation

$$\mathcal{C}[\![\texttt{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$
$$\mathcal{C}[\![c_1 \,\texttt{;}\, c_2]\!] = \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\}$$
$$\mathcal{C}[\![v \,\texttt{:=}\, a]\!] = \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\}$$
$$\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] = ?$$

Command Valuation

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\ \texttt{:=}\ a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\}
\end{aligned}
$$

Command Valuation

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\,\texttt{:=}\,a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\}
\end{aligned}
$$

Command Valuation

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\ \texttt{:=}\ a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= ?
\end{aligned}
$$

Command Valuation

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\,\texttt{:=}\,a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= \mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ (c\texttt{;while}\ b\ \texttt{do}\ c)\ \texttt{else skip}]\!]
\end{aligned}
$$

Command Valuation

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma,\sigma') \mid (\sigma,\sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2,\sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\,\texttt{:=}\,a]\!] &= \{(\sigma,\sigma[v \mapsto n]) \mid (\sigma,n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma,\sigma') \mid (\sigma,F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= \mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ (c\texttt{;while}\ b\ \texttt{do}\ c)\ \texttt{else skip}]\!]
\end{aligned}
$$

Problem: This is not a well-founded recursive definition. Why?

Well-founded Relations

Even though in computer programming we may have gotten used to writing non-terminating functions, such as

```
let f () = f ();;
```

in mathematics if I ask you to define a function f satisfying some property P, it is not acceptable to define

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

This is not a definition. It is a property of f (trivially satisfied by all functions f), but it does not define f.

Well-founded Relations

Let's expand our "definition" of $\mathcal{C}$ for while-loops:

$$\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]$$

$$= \mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ (c;\texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}]\!]$$

$$= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c;\texttt{while}\ b\ \texttt{do}\ c]\!]\}$$
$$\cup \{(\sigma,\sigma') \mid (\sigma,F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![\texttt{skip}]\!]\}$$

$$= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in \mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\}$$
$$\cup \{(\sigma,\sigma) \mid (\sigma,F) \in \mathcal{B}[\![b]\!]\}$$

Well-founded Relations

Let's expand our "definition" of $\mathcal{C}$ for while-loops:

$$\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]$$

$$= \mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ (c;\texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}]\!]$$

$$= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c;\texttt{while}\ b\ \texttt{do}\ c]\!]\}$$
$$\cup \{(\sigma,\sigma') \mid (\sigma,F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![\texttt{skip}]\!]\}$$

$$= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in \mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\}$$
$$\cup \{(\sigma,\sigma) \mid (\sigma,F) \in \mathcal{B}[\![b]\!]\}$$

Well-founded Relations

Let's expand our "definition" of $\mathcal{C}$ for while-loops:

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!]$$

$$= \mathcal{C}[\![\texttt{if } b \texttt{ then } (c\texttt{;while } b \texttt{ do } c) \texttt{ else skip}]\!]$$

$$= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c\texttt{;while } b \texttt{ do } c]\!]\}$$
$$\cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![\texttt{skip}]\!]\}$$

$$= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!]\}$$
$$\cup \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\}$$

$\mathcal{C}$ is defined in terms of itself. Not a valid recursive definition.

The Problem of Loops

How do we fix this?

Looks like our days of side-stepping the problem through trickery are over. We actually have to solve the problem of what loops really mean mathematically. This is going to require some real work. (Bear with me.)

The Problem of Loops

High-level goal: A while-loop should denote a set of input-output store pairs. Each output store is the final state of the program when the loop terminates.

If a loop doesn't terminate, the function should have no output state associated with that input state (i.e., undefined for that input).

(Remember, every program must be a denotation, but its denotation is permitted to be a partial function that is undefined for some inputs.)

Iterative Approach

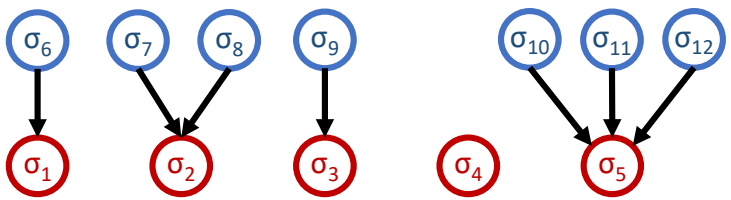
Idea: Maybe we can build this set of input-output pairs iteratively.

Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \{(\sigma_1, \sigma_1), (\sigma_2, \sigma_2), (\sigma_3, \sigma_3), (\sigma_4, \sigma_4), (\sigma_5, \sigma_5)\} \cup \cdots$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.

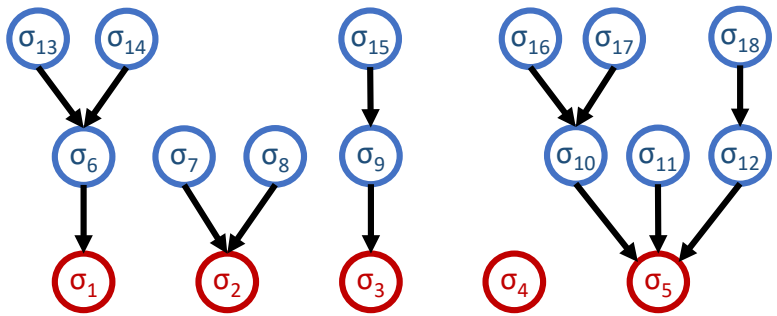


Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = &\{(\sigma_1,\sigma_1),(\sigma_2,\sigma_2),(\sigma_3,\sigma_3),(\sigma_4,\sigma_4),(\sigma_5,\sigma_5)\} \cup \\ &\{(\sigma_6,\sigma_1),(\sigma_7,\sigma_2),(\sigma_8,\sigma_2),(\sigma_9,\sigma_3),(\sigma_{10},\sigma_5),(\sigma_{11},\sigma_5),(\sigma_{12},\sigma_5)\} \cup \cdots\end{aligned}$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.

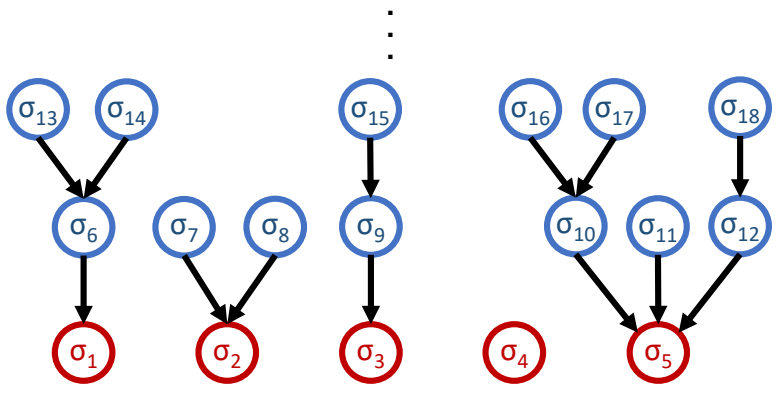


Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = {} & \{(\sigma_1,\sigma_1),(\sigma_2,\sigma_2),(\sigma_3,\sigma_3),(\sigma_4,\sigma_4),(\sigma_5,\sigma_5)\} \cup \\ & \{(\sigma_6,\sigma_1),(\sigma_7,\sigma_2),(\sigma_8,\sigma_2),(\sigma_9,\sigma_3),(\sigma_{10},\sigma_5),(\sigma_{11},\sigma_5),(\sigma_{12},\sigma_5)\} \cup \\ & \{(\sigma_{13},\sigma_1),(\sigma_{14},\sigma_1),(\sigma_{15},\sigma_3),(\sigma_{16},\sigma_5),(\sigma_{17},\sigma_5),(\sigma_{18},\sigma_5)\} \cup \cdots\end{aligned}$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.

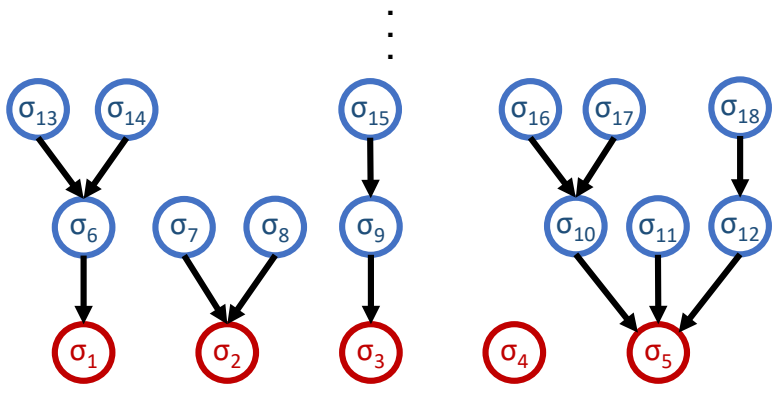


Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = {} & \{(\sigma_1,\sigma_1),(\sigma_2,\sigma_2),(\sigma_3,\sigma_3),(\sigma_4,\sigma_4),(\sigma_5,\sigma_5)\} \cup \\ & \{(\sigma_6,\sigma_1),(\sigma_7,\sigma_2),(\sigma_8,\sigma_2),(\sigma_9,\sigma_3),(\sigma_{10},\sigma_5),(\sigma_{11},\sigma_5),(\sigma_{12},\sigma_5)\} \cup \\ & \{(\sigma_{13},\sigma_1),(\sigma_{15},\sigma_3),(\sigma_{16},\sigma_5),(\sigma_{17},\sigma_5),(\sigma_{18},\sigma_5)\} \cup \\ & \vdots\end{aligned}$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.

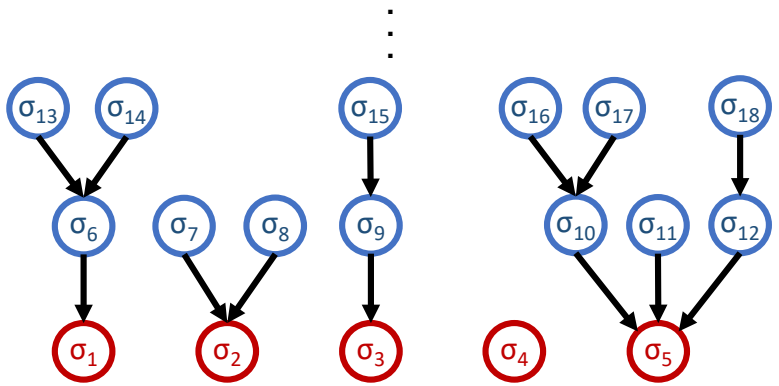


Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = {} & \{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\ & \{(\sigma_6,\sigma_1),(\sigma_7,\sigma_2),(\sigma_8,\sigma_2),(\sigma_9,\sigma_3),(\sigma_{10},\sigma_5),(\sigma_{11},\sigma_5),(\sigma_{12},\sigma_5)\} \cup \\ & \{(\sigma_{13},\sigma_1),(\sigma_{15},\sigma_3),(\sigma_{16},\sigma_5),(\sigma_{17},\sigma_5),(\sigma_{18},\sigma_5)\} \cup \\ & \vdots\end{aligned}$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.

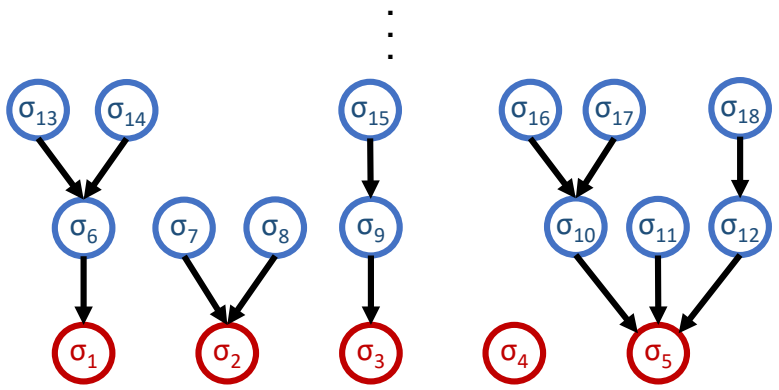


Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = &\{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\ &\{(\sigma,\sigma') \mid (\sigma, T), (\sigma', F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c]\!]\} \cup \\ &\{(\sigma_{13},\sigma_1), (\sigma_{15},\sigma_3), (\sigma_{16},\sigma_5), (\sigma_{17},\sigma_5), (\sigma_{18},\sigma_5)\} \cup \\ &\vdots\end{aligned}$$

Iterative Approach

Idea: Maybe we can build this set of input-output pairs iteratively.



Red states σ are those in which $\mathcal{B}[\![b]\!]\sigma = F$.
Blue states σ are those in which $\mathcal{B}[\![b]\!]\sigma = T$.
Edges $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ show how loop body c changes the state.

$$\begin{aligned}\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = &\{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\ &\{(\sigma,\sigma') \mid (\sigma, T), (\sigma', F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c]\!]\} \cup \\ &\{(\sigma,\sigma') \mid (\sigma, T), (\sigma_2, T), (\sigma', F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2), (\sigma_2,\sigma') \in \mathcal{C}[\![c]\!]\} \cup \\ &\vdots\end{aligned}$$

Making it a Recursion

$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\}$$

Making it a Recursion

$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\}$$
$$s_2 = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_1\}$$

Making it a Recursion

$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\}$$
$$s_2 = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_1\}$$
$$s_3 = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_2\}$$

Making it a Recursion

$$
\begin{aligned}
s_1 &= \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \\
s_2 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_1\} \\
s_3 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_2\} \\
s_n &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \qquad (\forall n > 1)
\end{aligned}
$$

Making it a Recursion

$$
\begin{aligned}
s_1 &= \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \\
s_2 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_1\} \\
s_3 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_2\} \\
s_n &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \qquad (\forall n > 1)
\end{aligned}
$$

$$
\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 1} s_n
$$

Making it a Recursion

$$
\begin{aligned}
s_0 &= \{\,\} \\
s_1 &= \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \\
s_2 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_1\} \\
s_3 &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_2\} \\
s_n &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \qquad (\forall n > 1)
\end{aligned}
$$

$$
\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} s_n
$$

Making it a Recursion

$$
\begin{aligned}
s_0 &= \{\,\} \\
s_1 &= \{(\sigma,\sigma) \mid (\sigma,F) \in \mathcal{B}[\![b]\!]\} \cup s_0 \\
s_2 &= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in s_1\} \cup s_1 \\
s_3 &= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in s_2\} \cup s_2 \\
s_n &= \{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in s_{n-1}\} \cup s_{n-1} \qquad (\forall n > 1)
\end{aligned}
$$

$$
\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} s_n
$$

This gives us the useful property that s is now a family of nested subsets $s_0 \subseteq s_1 \subseteq s_2 \subseteq \cdots$

Making it a Recursion

$$s_0 = \{\,\}$$
$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup s_0$$
$$s_n = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \cup s_{n-1} \qquad (\forall n > 1)$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} s_n$$

Idea: Is it possible to define a function Γ that, when given s_i as input returns the next s_{i+1}?

$$\Gamma(s) = ?$$

Making it a Recursion

$$s_0 = \{\,\}$$
$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup s_0$$
$$s_n = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \cup s_{n-1} \qquad (\forall n > 1)$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} s_n$$

Idea: Is it possible to define a function Γ that, when given s_i as input returns the next s_{i+1}?

$$\Gamma(s) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup ?$$

Making it a Recursion

$$s_0 = \{\,\}$$
$$s_1 = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup s_0$$
$$s_n = \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s_{n-1}\} \cup s_{n-1} \qquad (\forall n > 1)$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} s_n$$

Idea: Is it possible to define a function Γ that, when given s_i as input returns the next s_{i+1}?

$$\Gamma(s) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in s\}$$

Making it a Recursion

Idea: Now we can simplify away all the s sets and just use Γ!

$$
\begin{aligned}
s_0 &= \{\,\} \\
s_1 &= \Gamma(\{\}) \\
s_2 &= \Gamma(\Gamma(\{\})) \\
s_3 &= \Gamma(\Gamma(\Gamma(\{\}))) \\
s_n &= \Gamma^n(\{\})
\end{aligned}
$$

$$
\begin{aligned}
\Gamma(s) = \{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\
\{(\sigma,\sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in s\}
\end{aligned}
$$

$$
\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n \geq 0} \Gamma^n(\{\})
$$

Making it a Recursion

Notation: In lattice theory, the partial function that is undefined for all inputs (i.e., $\{\}$) is often written $\bot$.

$$\begin{aligned}\Gamma(s) = &\{(\sigma,\sigma) \mid (\sigma,F) \in \mathcal{B}[\![b]\!]\} \cup \\ &\{(\sigma,\sigma') \mid (\sigma,T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in s\}\end{aligned}$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n\geq 0} \Gamma^n(\bot)$$

Deeper Mathematical Connections

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v\ \texttt{:=}\ a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= \bigcup_{n \geq 0} \Gamma^n(\bot)
\end{aligned}
$$

where

$$
\begin{aligned}
\Gamma(f) = &\{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\
&\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}
\end{aligned}
$$

It turns out many of these functions have names you might recognize!

Deeper Mathematical Connections

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \iota_\Sigma \quad \text{(identity function)} \\
\mathcal{C}[\![c_1\,;c_2]\!] &= \{(\sigma,\sigma') \mid (\sigma,\sigma_2) \in \mathcal{C}[\![c_1]\!], (\sigma_2,\sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![v := a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] &= \{(\sigma,\sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma,\sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma,\sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] &= \bigcup_{n \geq 0} \Gamma^n(\bot)
\end{aligned}
$$

where

$$
\begin{aligned}
\Gamma(f) &= \{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma,\sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in f\}
\end{aligned}
$$

It turns out many of these functions have names you might recognize!

Deeper Mathematical Connections

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \iota_\Sigma \quad \text{(identity function)} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \quad \text{(function composition)} \\
\mathcal{C}[\![v\ \texttt{:=}\ a]\!] &= \{(\sigma, \sigma[v \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= \bigcup_{n \geq 0} \Gamma^n(\bot)
\end{aligned}
$$

where

$$
\begin{aligned}
\Gamma(f) = &\{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\
&\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}
\end{aligned}
$$

It turns out many of these functions have names you might recognize!

Deeper Mathematical Connections

$$
\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!] &= \iota_\Sigma \quad \text{(identity function)} \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \quad \text{(function composition)} \\
\mathcal{C}[\![v\ \texttt{:=}\ a]\!] &= [v \mapsto \cdot] \circ \mathcal{A}[\![a]\!] \quad \text{(functional update)} \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, F) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] &= \bigcup_{n \geq 0} \Gamma^n(\bot)
\end{aligned}
$$

where

$$
\begin{aligned}
\Gamma(f) &= \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}
\end{aligned}
$$

It turns out many of these functions have names you might recognize!

Deeper Mathematical Connections

$$\mathcal{C}[\![\texttt{skip}]\!] = \iota_\Sigma \quad \text{(identity function)}$$

$$\mathcal{C}[\![c_1\,;c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \quad \text{(function composition)}$$

$$\mathcal{C}[\![v := a]\!] = [v \mapsto \cdot] \circ \mathcal{A}[\![a]\!] \quad \text{(functional update)}$$

$$\mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] = \mathcal{C}[\![c_1]\!]\big|_{\mathcal{B}[\![b]\!]} \sqcup \mathcal{C}[\![c_2]\!]\big|_{\neg\mathcal{B}[\![b]\!]} \quad \text{(lattice join)}$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \bigcup_{n\geq 0} \Gamma^n(\bot)$$

where

$$\Gamma(f) = \{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma,\sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in f\}$$

It turns out many of these functions have names you might recognize!

Deeper Mathematical Connections

$$\mathcal{C}[\![\texttt{skip}]\!] = \iota_\Sigma \quad \text{(identity function)}$$

$$\mathcal{C}[\![c_1\,;c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \quad \text{(function composition)}$$

$$\mathcal{C}[\![v := a]\!] = [v \mapsto \cdot] \circ \mathcal{A}[\![a]\!] \quad \text{(functional update)}$$

$$\mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] = \mathcal{C}[\![c_1]\!]\big|_{\mathcal{B}[\![b]\!]} \sqcup \mathcal{C}[\![c_2]\!]\big|_{\neg\mathcal{B}[\![b]\!]} \quad \text{(lattice join)}$$

$$\mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] = \mathit{fix}(\Gamma) \quad \text{(least fixed point)}$$

where

$$\Gamma(f) = \{(\sigma,\sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma,\sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma,\sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2,\sigma') \in f\}$$

It turns out many of these functions have names you might recognize!

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?
Answer: $x = 0$ and $x = 1$

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?
Answer: $x = 0$ and $x = 1$

Exercise: What is a fixed point of $g(x) = x + 1$?

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?
Answer: $x = 0$ and $x = 1$

Exercise: What is a fixed point of $g(x) = x + 1$?
Answer: g has no fixed points.

Takeaway: Functions can have one fixed point, many fixed points, or none.

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?
Answer: $x = 0$ and $x = 1$

Exercise: What is a fixed point of $h(S) = \{x^2 \mid x \in S\}$?

Introduction to Fixed Points

Definition (fixed point): A fixed point of a function $f : A \rightharpoonup A$ is any value $x \in A$ satisfying $f(x) = x$.

Exercise: What is a fixed point of $f(x) = x^2$?
Answer: $x = 0$ and $x = 1$

Exercise: What is a fixed point of $h(S) = \{x^2 \mid x \in S\}$?
Answer: $\{\}$, $\{0\}$, $\{1\}$, and $\{0, 1\}$ are all fixed points.

Introduction to Fixed Points

Definition (least fixed point): A least fixed point of a function $f : A \rightharpoonup A$ is a fixed point $x \in A$ such that all fixed points $y \in A$ satisfy $x \leq y$.

Exercise: What is the *least* fixed point of $f(x) = x^2$?

Introduction to Fixed Points

Definition (least fixed point): A least fixed point of a function $f : A \rightharpoonup A$ is a fixed point $x \in A$ such that all fixed points $y \in A$ satisfy $x \leq y$.

Exercise: What is the *least* fixed point of $f(x) = x^2$?
Answer: 0

In General: All functions have at most one least fixed point.

Introduction to Fixed Points

Definition (least fixed point): A least fixed point of a function $f : A \rightharpoonup A$ is a fixed point $x \in A$ such that all fixed points $y \in A$ satisfy $x \leq y$.

Exercise: What is the *least* fixed point of $f(x) = x^2$?
Answer: 0

Exercise: What is the least fixed point of $h(S) = \{x^2 \mid x \in S\}$?

What does "least" even mean when domain A consists of sets, not numbers?

Introduction to Fixed Points

Definition (least fixed point): A least fixed point of a function $f : A \rightharpoonup A$ is a fixed point $x \in A$ such that all fixed points $y \in A$ satisfy $x \leq y$.

Exercise: What is the *least* fixed point of $f(x) = x^2$?
Answer: 0

Exercise: What is the least fixed point of $h(S) = \{x^2 \mid x \in S\}$?

What does "least" even mean when domain A consists of sets, not numbers?

In general, $\leq$ can be any specified relation over domain A. In the case of sets, we typially use $\subseteq$.

Introduction to Fixed Points

Definition (least fixed point): A least fixed point of a function $f : A \rightharpoonup A$ is a fixed point $x \in A$ such that all fixed points $y \in A$ satisfy $x \subseteq y$.

Exercise: What is the *least* fixed point of $f(x) = x^2$?
Answer: 0

Exercise: What is the least fixed point of $h(S) = \{x^2 \mid x \in S\}$?
Answer: $\{\}$ is the least fixed point of h.

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$

Does Γ have a least fixed point? If so, what is it?

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$

Does Γ have a least fixed point? If so, what is it?

First, what is Γ's type (domain and range)?

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$

Does $\Gamma : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ have a least fixed point? If so, what is it?

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$

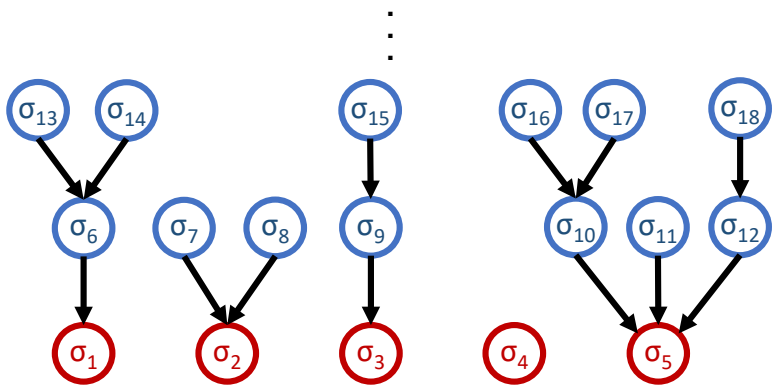
Does $\Gamma : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ have a least fixed point? If so, what is it?

Amazing fact: $\mathit{fix}(\Gamma) = \bigcup_{n \geq 0} \Gamma^n(\bot) = \mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]$

Why?

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$

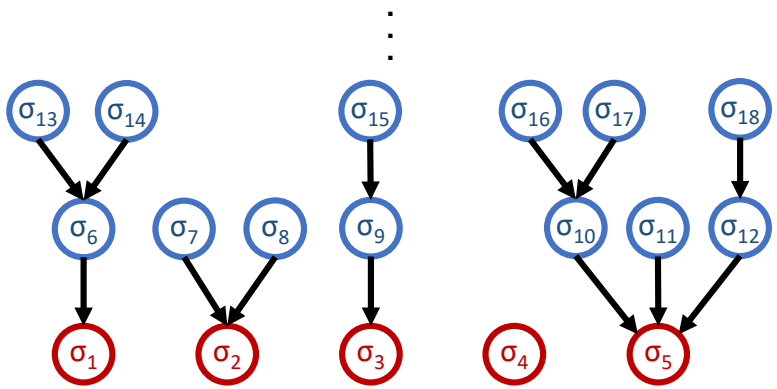


Intuition #1: $\bigcup_{n \geq 0} \Gamma^n(\bot)$ includes all (infinity) of the input-output pairs in the state diagram above. What does $\Gamma\left(\bigcup_{n \geq 0} \Gamma^n(\bot)\right)$ compute?

It adds the "next generation" of input-output pairs to the set, but there's no more to add. So it returns the same set. Therefore it's a fixed point.

Fixed Point Theory

$$\Gamma(f) = \{(\sigma, \sigma) \mid (\sigma, F) \in \mathcal{B}[\![b]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid (\sigma, T) \in \mathcal{B}[\![b]\!], (\sigma, \sigma_2) \in \mathcal{C}[\![c]\!], (\sigma_2, \sigma') \in f\}$$



Intuition #2: Any strict subset of $S \subsetneq \bigcup_{n \geq 0} \Gamma^n(\bot)$ cannot be a fixed point because $\Gamma(S)$ would add more input-output pairs.

Intuitive conclusion: $\bigcup_{n \geq 0} \Gamma^n(\bot)$ is the least fixed point of Γ.

(Warning: This is not a proof, just guiding intuition.)

Knaster-Tarski Fixed-point Theorem

Theorem: Knaster-Tarski Fixed-point Theorem

Let A be any complete partial order with bottom $\bot$, and let $f : A \to A$ be any continuous function. The least fixed point of f is $\bigsqcup_{i \geq 0} f^i(\bot)$.

Partial Order

A partial order is a set A on which there is a binary relation $\sqsubseteq$ that is:

- reflexive: $\forall a \in A, a \sqsubseteq a$
- transitive: $\forall a, b, c \in A, a \sqsubseteq b \sqsubseteq c \Longrightarrow a \sqsubseteq c$
- antisymmetric: $\forall a, b \in A, a \sqsubseteq b \sqsubseteq a \Longrightarrow a = b$

Knaster-Tarski Fixed-point Theorem

ω-chain

An ω-chain of a partial order $(A, \sqsubseteq)$ is an infinite sequence $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$.

CPO Completeness

A partial order $(A, \sqsubseteq)$ is **complete** if every ω-chain has a least upper bound.

Monotonicity

A function $f : A \to A$ is **monotonic** if $a \sqsubseteq b \Longrightarrow f(a) \sqsubseteq f(b)$ $(\forall a, b \in A)$.

Continuity

A function f is **continuous** if it is monotonic and for every ω-chain $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$,

$$\bigsqcup_{i \geq 0} f(a_i) = f\left(\bigsqcup_{i \geq 0} a_i\right)$$

Knaster-Tarski Fixed-point Theorem

Theorem: Knaster-Tarski Fixed-point Theorem

Let A be any complete partial order with bottom $\bot$, and let $f : A \to A$ be any continuous function. The least fixed point of f is $\bigsqcup_{i \geq 0} f^i(\bot)$.

Turns out $\Sigma \rightharpoonup \Sigma$ is a complete partial order with bottom $\bot$, and Γ is a continuous function over that CPO. (See online notes for complete proof.)

Therefore, Knaster-Tarski proves that $\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!]$ is its least fixed point.

Why do we care? Proving things about code boils down to proving things about loops. Proving things about infinite subsets is hard, but proving things about least fixed points is greatly facilitated by a powerful tool: **fixed-point induction**.

... which we will learn about next time!