# Program Analysis and Verification

0368-4479

http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html

Noam Rinetzky

## Lecture 3: Program Semantics

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

# Good manners

- Mobiles

# Admin

- Grades
  - First home assignment will be published on Tuesday.
    - Due lesson 5

✓ Scribes (this week)

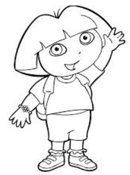
? Scribes (next week)

# Today

- Operational semantics
  - Advanced features
- Traces semantics
- Denotational Semantics

# What do we mean?

⟦שלום⟧ =

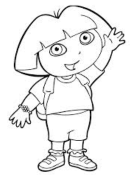


*syntax*

*semantics*

# What do we mean?

$$[\![ \; P \; ]\!] =$$



P: x→7

*syntax*

*semantics*

# Why formal semantics?

- Implementation-independent definition of a programming language

- Automatically generating interpreters (and some day maybe full fledged compilers)

- **Verification and debugging**
  - if you don't know what it does, how do you know its incorrect?

# Programming Languages

- Syntax
    - "how do I write a program?"
  - BNF
  - "Parsing"

- Semantics
    - "What does my program mean?"
  - …

# Program semantics

- State-transformer
  - Set-of-states transformer
  - Trace transformer
- Predicate-transformer
- Functions
- …

# What semantics do we want?

- Captures the aspects of computations we care about
  - “adequate”
- Hides irrelevant details
  - “fully abstract”
- Compositional

# A simple imperative language: **While**

Abstract syntax:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

$$b ::= \texttt{true} \mid \texttt{false}$$
$$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$S ::= x \texttt{ := } a \mid \texttt{skip} \mid S_1 \texttt{;}\ S_2$$
$$\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$
$$\mid \texttt{while}\ b\ \texttt{do}\ S$$

# Syntactic categories

| | |
|---|---|
| $n \in$ **Num** | numerals |
| $x \in$ **Var** | program variables |
| $a \in$ **Aexp** | arithmetic expressions |
| $b \in$ **Bexp** | boolean expressions |
| $S \in$ **Stm** | statements |

# Semantic categories

| | |
|---|---|
| **Z** | Integers {0, 1, -1, 2, -2, …} |
| **T** | Truth values {**ff**, **tt**} |
| **State** | **Var** → **Z** |

| | |
|---|---|
| Example state: | s=[x↦5, y↦7, z↦0] |
| Lookup: | s x = 5 |
| Update: | s[x↦6] = [x↦6, y↦7, z↦0] |

# Semantics of expressions

- Arithmetic expressions are side-effect free
  - Semantic function $\mathcal{A}[\![\,\textbf{Aexp}\,]\!] : \textbf{State} \to \textbf{Z}$
  - Defined by induction on the syntax tree

    $\mathcal{A}[\![\, n \,]\!]\, s = n$

    $\mathcal{A}[\![\, x \,]\!]\, s = s\ x$

    $\mathcal{A}[\![\, a_1 + a_2 \,]\!]\, s = \mathcal{A}[\![\, a_1 \,]\!]\, s + \mathcal{A}[\![\, a_2 \,]\!]\, s$

    $\mathcal{A}[\![\, a_1 - a_2 \,]\!]\, s = \mathcal{A}[\![\, a_1 \,]\!]\, s - \mathcal{A}[\![\, a_2 \,]\!]\, s$

    $\mathcal{A}[\![\, a_1 * a_2 \,]\!]\, s = \mathcal{A}[\![\, a_1 \,]\!]\, s \times \mathcal{A}[\![\, a_2 \,]\!]\, s$

    $\mathcal{A}[\![\, (a_1) \,]\!]\, s = \mathcal{A}[\![\, a_1 \,]\!]\, s$ --- not needed

    $\mathcal{A}[\![\, - a \,]\!]\, s = 0 - \mathcal{A}[\![\, a_1 \,]\!]\, s$
  - Compositional
  - Properties can be proved by structural induction
- Similarly for Boolean expressions

# Operational Semantics

(Recap)

# Operational Semantics

*The **meaning** of a program **in the language** is explained in terms of **a hypothetical compute**r which performs the set of actions which constitute the **elaboration** of that program.*

[Algol68, Section 2]

# Operational Semantics

It is all very well to aim for a more 'abstract' and a 'cleaner' approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored.

[Scott70]

# Operational semantics

- Concerned with **how** to execute programs
  - How statements modify **state**
  - Define transition relation between configurations
- Two flavors
  - Structural operational semantics: describes how the **individual steps** of a computations take place
    - So-called “small-step” semantics
  - Natural semantics: describes how the **overall** results of executions are obtained
    - So-called “big-step” semantics

# Structural operating semantics (SOS)

- aka "Small-step semantics"

$$\langle S, s \rangle \Rightarrow \langle S', s' \rangle$$

first step

# Structural operational semantics

- Developed by Gordon Plotkin [TR 1981]
- Configurations: $\gamma$ has one of two forms:

  $\langle S, s \rangle$ Statement *S* is about to execute on state *s*

  *s* Terminal (final) state

- Transitions $\langle S, s \rangle \Rightarrow \gamma$ (first step)

  $\gamma = \langle S', s' \rangle$ Execution of *S* from *s* is **not** completed and remaining computation proceeds from intermediate configuration $\gamma$

  $\gamma = s'$ Execution of *S* from *s* has **terminated** and the final state is s'

- $\langle S, s \rangle$ is stuck if there is no $\gamma$ such that $\langle S, s \rangle \Rightarrow \gamma$

# Structural semantics for **While**

[ass$_{sos}$] $\langle x{:=}a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

[skip$_{sos}$] $\langle \texttt{skip}, s\rangle \Rightarrow s$

[comp$^1_{sos}$] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[comp$^2_{sos}$] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[if$^{tt}_{sos}$] $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle$ if $\mathcal{B}[\![b]\!]\,s = \mathbf{tt}$

[if$^{ff}_{sos}$] $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle$ if $\mathcal{B}[\![b]\!]\,s = \mathbf{ff}$

# Derivation sequences

- A derivation sequence of a statement S starting in state s is either
- A **finite** sequence $\gamma_0, \gamma_1, \gamma_2 \ldots, \gamma_k$ such that
  1. $\gamma_0 = \langle S, s \rangle$
  2. $\gamma_i \Rightarrow \gamma_{i+1}$
  3. $\gamma_k$ is either stuck configuration or a final state
- An **infinite** sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ such that
  1. $\gamma_0 = \langle S, s \rangle$
  2. $\gamma_i \Rightarrow \gamma_{i+1}$
- Notations:
  – $\gamma_0 \Rightarrow^k \gamma_k$ $\gamma_0$ derives $\gamma_k$ in k steps
  – $\gamma_0 \Rightarrow^* \gamma$ $\gamma_0$ derives $\gamma$ in a finite number of steps

# Natural operating semantics (NS)

- aka "Large-step semantics"

$$\langle S, s \rangle \longrightarrow s'$$

all steps

# Natural operating semantics

- Developed by Gilles Kahn [STACS 1987]
- Configurations

| | |
|---|---|
| $\langle S, s\rangle$ | Statement *S* is about to execute on state *s* |
| *s* | Terminal (final) state |

- Transitions

| | |
|---|---|
| $\langle S, s\rangle \rightarrow s'$ | Execution of *S* from *s* will terminate with the result state *s*' |

  – Ignores non-terminating computations

# Natural operating semantics

- $\longrightarrow$ defined by rules of the form

premise

side condition

$$\frac{\langle S_1, s_1\rangle \longrightarrow s_1', \ldots, \langle S_n, s_n\rangle \longrightarrow s_n'}{\langle S, s\rangle \longrightarrow s'} \quad \text{if} \ldots$$

conclusion

- The meaning of compound statements is defined using the meaning immediate constituent statements

# Natural semantics for **While**

[ass$_{ns}$] $\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

[skip$_{ns}$] $\langle \texttt{skip}, s\rangle \to s$

axioms

[comp$_{ns}$] $$\frac{\langle S_1, s\rangle \to s',\ \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

[if$^{tt}{}_{ns}$] $$\frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'}$$ if $\mathcal{B}[\![b]\!]\, s = \textbf{tt}$

[if$^{ff}{}_{ns}$] $$\frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'}$$ if $\mathcal{B}[\![b]\!]\, s = \textbf{ff}$

# Derivation trees

- Using axioms and rules to derive a transition $\langle S, s\rangle \rightarrow$ s' gives a derivation tree
  - Root: $\langle S, s\rangle \rightarrow$ s'
  - Leaves: axioms
  - Internal nodes: conclusions of rules
    - Immediate children: matching rule premises

# Evaluation via derivation sequences

- For any **While** statement $S$ and state $s$ it is always possible to find at least one derivation sequence from $\langle S, s \rangle$
  - Apply axioms and rules forever or until a terminal or stuck configuration is reached
- ***Proposition:*** *there are no stuck configurations in* ***While***

# The semantic function $S_{sos}$

- The meaning of a statement $S$ is defined as a partial function from **State** to **State**

  $S_{sos}$: **Stm** → (**State** ↪ **State**)

$$S_{sos}[\![S]\!]\, s = \begin{cases} s' & \text{if } \langle S, s\rangle \Rightarrow^* s' \\ \text{undefined} & \textit{otherwise} \end{cases}$$

$[\![S]\!]$: What a statement S means to the context: P[S]
- $S_1$ = `x:=x+2`
- $S_2$ = `x:=x+1 ; x:=x+1`
- P[$S$] = `z:=x; S; y:=x;`

- Examples:

  $S_{sos}[\![$`skip`$]\!]\, s = s$

  $S_{sos}[\![$`x:=1`$]\!]\, s = s\,[x \mapsto 1]$

  $S_{sos}[\![$`while true do skip`$]\!]\, s$ = undefined

# The semantic function $S_{ns}$

- The meaning of a statement $S$ is defined as a partial function from **State** to **State**

  $S_{ns}$: **Stm** → (**State** ↪ **State**)

$$S_{ns}\,[\![S]\!]\,s = \begin{cases} s' & \text{if } \langle S, s\rangle \longrightarrow s' \\ \text{undefined} & \textit{otherwise} \end{cases}$$

- Examples:

  $S_{ns}\,[\![\texttt{skip}]\!]s = s$

  $S_{ns}\,[\![\texttt{x:=1}]\!]s = s\,[x \mapsto 1]$

  $S_{ns}\,[\![\texttt{while true do skip}]\!]s = \text{undefined}$

# An equivalence result

- $S_1$ and $S_2$ are semantically equivalent if for all $s$ and $s'$ $\langle S_1, s\rangle \to s'$ if and only if $\langle S_2, s\rangle \to s'$
  - Same semantics

- SOS and NS
  - For every statement in **While** $S_{ns} [\![S]\!] = S_{sos} [\![S]\!]$
  - Proof in pages 40-43

# While in **WHILE**

# Semantic equivalence

- $S_1$ and $S_2$ are semantically equivalent if for all $s$ and $s'$
  $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$
- Simple example
  `while` $b$ `do` $S$
  is semantically equivalent to:
  `if` $b$ `then` ($S$; `while` $b$ `do` $S$) `else skip`
  – Read proof in pages 26-27

# Structural semantics for while

[while$_{sos}$]

$$\langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow \langle \texttt{if } b \texttt{ then } S; \texttt{while } b \texttt{ do } S) \texttt{ else } \texttt{skip}, s\rangle$$

# Natural semantics for **While**

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\langle \texttt{while } b \texttt{ do } S, s\rangle \longrightarrow s$ if $\mathcal{B}[\![b]\!]\, s = \mathbf{ff}$

Non-compositional

$[\text{while}^{\text{tt}}_{\text{ns}}]$ $\dfrac{\langle S, s\rangle \longrightarrow s',\ \langle \texttt{while } b \texttt{ do } S, s'\rangle \longrightarrow s''}{\langle \texttt{while } b \texttt{ do } S, s\rangle \longrightarrow s''}$ if $\mathcal{B}[\![b]\!]\, s = \mathbf{tt}$

# Comparing semantics

| Statement | Natural semantics | Structural semantics |
|---|---|---|
| `abort` | | |
| `abort;` *S* | | |
| `skip;` *S* | | |
| `if x = 0 then abort else y := y / x` | | |

- The natural semantics cannot describe *looping* executions
  - Every execution is represented by a *finite* derivation tree
- The structural operational semantics can describe both
  - Looping executions have *infinite* derivation sequences
    - Every step in the derivation sequence is justified by a *finite* derivation tree
  - Terminating executions have a finite one

# What is a semantics good for?

- Allows to "evaluate" a program

- Properties of programming language semantics holds for all programs …

# What is a semantics good for?

- Allows to “evaluate” a program
- Properties of programming language semantics holds for all programs …
- NS: more abstract
  - Fewer rules, top-down interpreter, simpler proofs
- SOS: more “accurate”
  - Order of evaluation,  non-termination

# Operational Semantics

(paceR)

# Operational Semantics

(Extended language)

# Language Extensions

- `abort` statement (like C's exit w/o return value)
- Non-determinism
- Parallelism
- Local Variables
- Procedures
  - Static Scope
  - Dynamic scope

# While + `abort`

- Abstract syntax

$$S ::= x \mathbf{:=} a \mid \mathbf{skip} \mid S_1; S_2$$
$$\mid \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$$
$$\mid \mathbf{while}\ b\ \mathbf{do}\ S$$
$$\mid \mathbf{abort}$$

- Abort terminates the execution
  - In "**`skip;`** *S*" the statement *S* executes
  - In "**`abort;`** *S*" the statement *S* should never execute
- Natural semantics rules: …?
- Structural semantics rules: …?

# Comparing semantics

| Statement | Natural semantics | Structural semantics |
|---|---|---|
| `abort` | | |
| `abort;` *S* | | |
| `skip;` *S* | | |
| `while true do skip` | | |
| `if x = 0 then abort else y := y / x` | | |

- The natural semantics cannot distinguish between *looping* and *abnormal termination*
  - Unless we add a special *error state*
- The structural operational semantics can distinguish
  - looping is reflected by infinite derivations and abnormal termination is reflected by stuck configuration

# **While** + non-determinism

- Abstract syntax

$$S ::= x \mathbf{\texttt{:=}}\ a \mid \mathbf{\texttt{skip}} \mid S_1;\ S_2$$
$$\mid \mathbf{\texttt{if}}\ b\ \mathbf{\texttt{then}}\ S_1\ \mathbf{\texttt{else}}\ S_2$$
$$\mid \mathbf{\texttt{while}}\ b\ \mathbf{\texttt{do}}\ S$$
$$\mid S_1\ \mathbf{\texttt{or}}\ S_2$$

- Either $S_1$ is executed or $S_2$ is executed
- Example: `x:=1 or (x:=2; x:=x+2)`
  - Possible outcomes for `x`: 1 and 4

# **While** + non-determinism: natural semantics

[or$^1_{ns}$] $$\frac{\langle S_1, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'}$$

[or$^2_{ns}$] $$\frac{\langle S_2, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'}$$

# **While** + non-determinism: structural semantics

$[or^1_{sos}]$ ?

$[or^2_{sos}]$ ?

# **While** + non-determinism

- What about the definitions of the semantic functions?
  - $S_{\text{ns}} [\![ S_1 \texttt{ or } S_2 ]\!] s$
  - $S_{\text{sos}} [\![ S_1 \texttt{ or } S_2 ]\!] s$

# Comparing semantics

| Statement | Natural semantics | Structural semantics |
|---|---|---|
| `x:=1 or (x:=2; x:=x+2)` | | |
| `(while true do skip) or (x:=2; x:=x+2)` | | |

- In the natural semantics non-determinism will suppress non-termination (looping) if possible
- In the structural operational semantics non-determinism does not suppress non-terminating statements

# **While** + parallelism

Abstract syntax

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2$$
$$\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$
$$\mid \texttt{while}\ b\ \texttt{do}\ S$$
$$\mid S_1 \parallel S_2$$

- All the interleaving of $S_1$ and $S_2$ are executed
- Example: `x:=1 ‖ (x:=2; x:=x+2)`
  - Possible outcomes for `x`: 1, 3, 4

# **While** + parallelism: structural semantics

[par$^1_{sos}$] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \| S_2, s\rangle \Rightarrow \langle S_1' \| S_2, s'\rangle}$$

[par$^2_{sos}$] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \| S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[par$^3_{sos}$] $$\frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \| S_2, s\rangle \Rightarrow \langle S_1 \| S_2', s'\rangle}$$

[par$^4_{sos}$] $$\frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \| S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

# **While** + parallelism: natural semantics

Challenge problem:
Give a formal proof that this is in fact impossible.

*Idea:* try to prove on a restricted version of **While** without loops/conditions

# Example: derivation sequences of a parallel statement

$\langle$`x:=1 ∥ (x:=2; x:=x+2),`$s\rangle \Rightarrow$

# **While** + Local Variables

- $S ::= \ldots \mid \texttt{\textbf{Let}}\ \ x := a\ \texttt{\textbf{in}}\ \ S$

# Conclusion

- In the structural operational semantics we concentrate on small steps so interleaving of computations can be easily expressed

- In the natural semantics immediate constituent is an atomic entity so we cannot express interleaving of computations

# **While** + memory

Abstract syntax

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2$$
$$\mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$
$$\mid \texttt{while}\ b\ \texttt{do}\ S$$
$$\mid x := \texttt{malloc}(a)$$
$$\mid x := [y]$$
$$\mid [x] := y$$

~~**State : Var**~~ → Z

**State : Stack** × **Heap**

**Stack : Var** → Z

**Heap** : Z → Z

Integers as memory addresses

# From states to traces

# Trace semantics

- Low-level (conceptual) semantics
- Add program counter (pc) with states
  - $\Sigma$ = **State** + pc
- The meaning of a program is a relation

$$\tau \subseteq \Sigma \times \textbf{Stm} \times \Sigma$$

- Execution is a finite/infinite sequence of states
- A useful concept in defining static analysis as we will see later

# Example

```
1: y := 1;
 while 2: ¬(x=1) do (
     3: y := y * x;
     4: x := x - 1
)
5:
```

# Traces

```
1: y := 1;
 while 2: ¬(x=1) do (
      3: y := y * x;
      4: x := x - 1
)
5:
```

Set of traces is infinite therefore trace semantics is incomputable in general

⟨{x↦2,y↦3},1⟩ [**y:=1**] ⟨{x↦2,y↦1},2⟩ [**¬(x=1)**] ⟨{x↦2,y↦1},3⟩ [**y:=y*x**] ⟨{x↦2,y↦2},4⟩ [**x:=x-1**] ⟨{x↦1,y↦2},2⟩ [**¬(x=1)**] ⟨{x↦1,y↦2},5⟩

⟨{x↦3,y↦3},1⟩ [**y:=1**] ⟨{x↦3,y↦1},2⟩ [**¬(x=1)**] ⟨{x↦3,y↦1},3⟩ [**y:=y*x**] ⟨{x↦3,y↦3},4⟩ [**x:=x-1**] ⟨{x↦2,y↦3},2⟩ [**¬(x=1)**] ⟨{x↦2,y↦3},3⟩ [**y:=y*x**] ⟨{x↦2,y↦6},4⟩ [**x:=x-1**] ⟨{x↦1,y↦6},2⟩ [**¬(x=1)**] ⟨{x↦1,y↦6}, 5⟩

…

# Operational semantics summary

- SOS is powerful enough to describe imperative programs
  - Can define the set of traces
  - Can represent program counter implicitly
  - Handle `goto` statements and other non-trivial control constructs (e.g., exceptions)
- Natural operational semantics is an abstraction
- Different semantics may be used to justify different behaviors
- Thinking in concrete semantics is essential for a analysis writer

# Denotational Semantics

Based on a lecture by Martin Abadi

# Denotational Semantics

- A "mathematical" semantics
  - $[\![S]\!]$ is a mathematical object
  - A fair amount of mathematics is involved
- Compositional
- More abstract and canonical than Op. Sem.
  - No notion of "execution"
    - Merely definitions
  - No small step vs. big step

# Denotational Semantics

- Denotational semantics is also called
  - Fixed point semantics
  - Mathematical semantics
  - Scott-Strachey semantics

# Plan

- Denotational semantics of While (1st attempt)
- Math
  - Complete partial orders
  - Montonicity
  - Continuity
- Denotational semantics of While

# Denotational semantics

- **A:** Aexp $\rightarrow (\Sigma \rightarrow N)$
- **B**: Bexp $\rightarrow (\Sigma \rightarrow T)$
- **S:** Stm $\rightarrow (\Sigma \rightharpoonup \Sigma)$
- Defined by structural induction

# Denotational semantics

- **A:** Aexp → (Σ→N)
- **B**: Bexp →(Σ→T)
- **S:** Stm →(Σ⇀Σ)
- Defined by structural induction

$\mathcal{A}[\![a]\!]$ , $\mathcal{B}[\![b]\!]$, $S_{ns}[\![S]\!]$, $S_{sos}[\![S]\!]$

# Denotational semantics of Aexp

- **A:** Aexp $\to$ $(\Sigma \to N)$
- **A** $[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$
- **A** $[\![X]\!] = \{(\sigma, \sigma X) \mid \sigma \in \Sigma\}$
- **A** $[\![a_0+a_1]\!] = \{(\sigma, n_0+n_1) \mid (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$
- **A** $[\![a_0-a_1]\!] = \{(\sigma, n_0-n_1) \mid (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$
- **A** $[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$

Functions represented as sets of pairs

Lemma: A $[\![a]\!]$ is a function

# Denotational semantics of Aexp with $\lambda$

- **A:** Aexp $\to$ ($\Sigma \to$N)
- **A** $[\![n]\!]$ = $\lambda\sigma\in\Sigma.n$
- **A** $[\![X]\!]$ = $\lambda\sigma\in\Sigma.\sigma(X)$
- **A** $[\![a_0+a_1]\!]$ = $\lambda\sigma\in\Sigma.(\mathbf{A}[\![a_0]\!]\sigma+\mathbf{A}[\![a_1]\!]\sigma)$
- **A** $[\![a_0-a_1]\!]$ = $\lambda\sigma\in\Sigma.(\mathbf{A}[\![a_0]\!]\sigma-\mathbf{A}[\![a_1]\!]\sigma)$
- **A** $[\![a_0\times a_1]\!]$ = $\lambda\sigma\in\Sigma.(\mathbf{A}[\![a_0]\!]\sigma\times\mathbf{A}[\![a_1]\!]\sigma)$

# Denotational semantics of Bexp

- **B:** Bexp $\rightarrow$ ($\Sigma \rightarrow$T)
- **B** $[\![\text{true}]\!] = \{(\sigma, \text{true}) \mid \sigma \in \Sigma\}$
- **B** $[\![\text{false}]\!] = \{(\sigma, \text{false}) \mid \sigma \in \Sigma\}$
- **B** $[\![a_0 = a_1]\!] = \{(\sigma, \text{true}) \mid \sigma \in \Sigma \ \& \ \mathbf{A}[\![a_0]\!]\sigma = \mathbf{A}[\![a_1]\!]\sigma\} \cup$
  $\{(\sigma, \text{false}) \mid \sigma \in \Sigma \ \& \ \mathbf{A}[\![a_0]\!]\sigma \neq \mathbf{A}[\![a_1]\!]\sigma\}$
- **B** $[\![a_0 \leq a_1]\!] = \{(\sigma, \text{true}) \mid \sigma \in \Sigma \ \& \ \mathbf{A}[\![a_0]\!]\sigma \leq \mathbf{A}[\![a_1]\!]\sigma\} \cup$
  $\{(\sigma, \text{false}) \mid \sigma \in \Sigma \ \& \ \mathbf{A}[\![a_0]\!]\sigma \not\leq \mathbf{A}[\![a_1]\!]\sigma\}$
- **B** $[\![\neg b]\!] = \{(\sigma, \neg_T t) \mid \sigma \in \Sigma, (\sigma, t) \in \mathbf{B}[\![b]\!]\}$
- **B** $[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge_T t_1) \mid \sigma \in \Sigma, (\sigma, t_0) \in \mathbf{B}[\![b_0]\!], (\sigma, t_1) \in \mathbf{B}[\![b_1]\!]\}$
- **B** $[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee_T t_1) \mid \sigma \in \Sigma, (\sigma, t_0) \in \mathbf{B}[\![b_0]\!], (\sigma, t_1) \in \mathbf{B}[\![b_1]\!]\}$

Lemma: B$[\![b]\!]$ is a function

# Denotational semantics of statements?

- Intuition:
  - Running a statement s starting from a state $\sigma$ yields another state $\sigma'$
- Can we define **S** $[\![s]\!]$ as a function that maps $\sigma$ to $\sigma'$ ?
  - **S** $[\![.]\!]$: Stm $\rightarrow (\Sigma \rightarrow \Sigma)$

# Denotational semantics of commands?

- Problem: running a statement might not yield anything if the statement does not terminate
- We introduce the special element $\perp$ to denote a special outcome that stands for non-termination
- For any set X, we write $X_\perp$ for $X \cup \{\perp\}$

- Convention:
  - whenever $f \in X \to X_\perp$ we extend f to $X_\perp \to X_\perp$ "strictly" so that $f(\perp) = \perp$

# Denotational semantics of statements?

- We try:
  - $S[\![.]\!] : Stm \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})$
- $S[\![skip]\!]\sigma = \sigma$
- $S[\![s_0 ; s_1]\!]\sigma = S[\![s_1]\!](S[\![s_0]\!]\sigma)$
- $S[\![\text{if } b \text{ then } s_0 \text{ else } s_1]\!]\sigma =$
  $\text{if } B[\![b]\!]\sigma \text{ then } S[\![s_0]\!]\sigma \text{ else } S[\![s_1]\!]\sigma$

# Examples

- $S [\![X{:=}\ 2;\ X{:=}1]\!] \sigma = \sigma[X \mapsto 1]$
- $S [\![\text{if true then } X{:=}2;\ X{:=}1 \text{ else } \ldots]\!]\ \sigma = \sigma[X \mapsto 1]$
- The semantics does not care about intermediate states
- So far, we did not explicitly need $\perp$

# Denotational semantics of loops?

- $S [\![ \text{while b do s} ]\!] \sigma = ?$

# Denotational semantics of statements?

- Abbreviation $W = S[\![\text{while b do s}]\!]$
- Idea: we rely on the equivalence

  while b do s ~ if b then (s; while b do s) else skip
- We may try using unwinding equation

  $W(\sigma) = \text{if } B[\![b]\!]\sigma \text{ then } W(S[\![s]\!]\,\sigma) \text{ else } \sigma$
- Unacceptable solution
  - Defines W in terms of itself
  - It not evident that a suitable W exists
  - It may not describe W uniquely
    (e.g., for while true do skip)

# Introduction to Domain Theory

- We will solve the unwinding equation through a general theory of recursive equations
- Think of programs as processors of streams of bits (streams of 0's and 1's, possibly terminated by $) What properties can we expect?

# Motivation

- Let "isone" be a function that must return "1$" when the input string has at least a 1 and "0$" otherwise
  - isone(00…0$)  = 0$
  - isone(xx…1…$) =1$
  - isone(0…0) =?
- **Monotonicity** : Output is never retracted
  - More information about the input is reflected in more information about the output
- How do we express monotonicity precisely?

# Montonicity

- Define a partial order
  $x \sqsubseteq y$
  – A partial order is reflexive, transitive, and anti-symmetric
  – y is a refinement of x
    - "more precise"
- For streams of bits $x \sqsubseteq y$ when x is a prefix of y
- For programs, a typical order is:
  – No output (yet) $\sqsubseteq$ some output

# Montonicity

- A set equipped with a partial order is a poset
- Definition:
  - D and E are postes
  - A function f: D →E is monotonic if
    $\forall x, y \in D: x \sqsubseteq_D y \Rightarrow f(x) \sqsubseteq_E f(y)$
  - The semantics of the program ought to be a monotonic function
    - More information about the input leads to more information about the output

# Montonicity Example

- Consider our "isone" function with the prefix ordering
- Notation:
  - $0^k$ is the stream with k consecutive 0's
  - $0^\infty$ is the infinite stream with only 0's
- Question (revisited): what is isone($0^k$ )?
  - By definition, isone($0^k$\$) = 0\$ and isone($0^k$1\$) = 1\$
  - But $0^k \sqsubseteq 0^k$\$ and $0^k \sqsubseteq 0^k$1\$
  - "isone" must be monotone, so:
    - isone( $0^k$ ) $\sqsubseteq$ isone( $0^k$\$) = 0\$
    - isone( $0^k$ ) $\sqsubseteq$ isone( $0^k$1\$) = 1\$
  - Therefore, monotonicity requires that isone($0^k$ ) is a common prefix of 0\$ and 1\$, namely $\varepsilon$

# Motivation

- Are there other constraints on "isone"?
- Define "isone" to satisfy the equations
  - isone(ε)=ε
  - isone(1s)=1$
  - isone(0s)=isone(s)
  - isone($)=0$
- What about $0^\infty$?
- Continuity: finite output depends only on finite input (no infinite lookahead)

# Chains

- A chain is a countable increasing sequence $\langle x_i \rangle = \{x_i \in X \mid x_0 \sqsubseteq x_1 \sqsubseteq \ldots\}$
- An upper bound of a set if an element "bigger" than all elements in the set
- The least upper bound is the "smallest" among upper bounds:
  - $x_i \sqsubseteq \sqcup\langle x_i \rangle$ for all $i \in N$
  - $\sqcup\langle x_i \rangle \sqsubseteq y$ for all upper bounds y of $\langle x_i \rangle$ and it is unique if it exists

# Complete Partial Orders

- Not every poset has an upper bound
  - with ⊥ ⊑ n and n⊑n for all n ∈N
  - {1, 2} does not have an upper bound
- Sometimes chains have no upper bound

0 1 2 …

⊥

⋮

2

1

0

The chain

0 ≤1≤2≤…

does not have an upper bound

# Complete Partial Orders

- It is convenient to work with posets where every chain (not necessarily every set) has a least upper bound
- A partial order P is complete if every chain in P has a least upper bound also in P
- We say that P is a complete partial order (cpo)
- A cpo with a least ("bottom") element $\perp$ is a pointed cpo (pcpo)

# Examples of cpo's

- Any set P with the order
  $x \sqsubseteq y$ if and only if $x = y$ is a cpo
  It is discrete or flat
- If we add $\perp$ so that $\perp \sqsubseteq x$ for all $x \in P$, we get a flat pointed cpo
- The set N with $\leq$ is a poset with a bottom, but not a complete one
- The set $N \cup \{ \infty \}$ with $n \leq \infty$ is a pointed cpo
- The set N with $\geq$ is a cpo without bottom
- Let S be a set and P(S) denotes the set of all subsets of S ordered by set inclusion
  - P(S) is a pointed cpo

# The End