LUND UNIVERSITY

EDA045F: Program Analysis

LECTURE 10: TYPES 1

Christoph Reichenbach

In the last lecture...

- Performance Counters
- Challenges in Dynamic Performance Analysis
- Taint Analysis
- Binary Instrumentation

Types

Java

```
int v;
```

Haskell

```
v :: Int
```

ML

```
val v : int
```

- Framework for classifying parts of programs by:
  - Which set they may be drawn from, and/or
  - What behaviour they exhibit
- *Dynamic type analysis* part of dynamic program analysis
- We focus on *static type analysis*
- Key properties of *static type analysis*:
  - Explicit user annotations widely accepted
  - Analysis is (usually) flow and context-insensitive
  - Modular checking with annotations possible

Sets vs. Types

- Some types correspond to sets ($\mathbb{B}$, $\mathbb{N}$)
- Some types become very complex when viewed as sets:
  - *Computable* functions from $\mathbb{N} \to \mathbb{N}$
  - `java.util.Map`: specification of expected behaviour of 'map' datatypes
- Most sets are not useful types:
  - Set of all prime numbers
  - Set of irrational numbers
  - Set of all infinite subsets of $\mathbb{N}$

  ...
- Type theory focuses on:
  - Decidable concepts (for typical type systems)
  - Provable concepts (for interactive theorem proving)

Sets and types intersect in parts, but have evolved into different schools of research

Types and Semantics

- Research on types and program semantics tightly intertwined
  - **Semantics**: What precisely does a given program mean / do / describe?
  - **Types**: Approximation of semantics (decidable or provable)
- Many formal connections
- We will look into:
  - Formal description of program semantics
  - Types and type systems
  - How to compute types

Syntax of a simple toy language

Syntax of language STOL-B:

$$
\begin{array}{lcl}
\mathit{val} & ::= & \mathit{nat} \\
 & | & \textsf{true} \;|\; \textsf{false} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle \;\textsf{plus}\; \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{expr} \rangle >= \langle \mathit{expr} \rangle \\
 & | & \textsf{if}\; \langle \mathit{expr} \rangle \;\textsf{then}\; \langle \mathit{expr} \rangle \;\textsf{else}\; \langle \mathit{expr} \rangle
\end{array}
$$

Examples:

- `5`
- `5 plus 27`
- `if 5 >= 2 then 0 else 1`

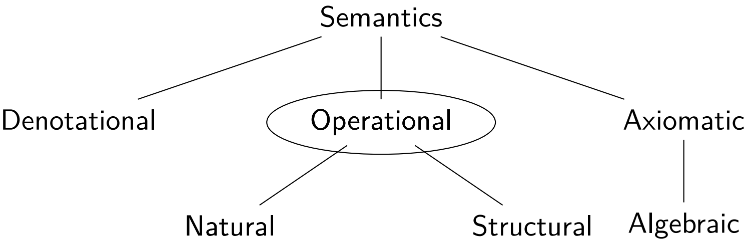
Meaning of our toy language: examples

What we want the meaning to be:

`5`	5
`5 plus 27`	32
`if 5 >= 2 then 1 else 0`	1

Giving Meaning to Programs

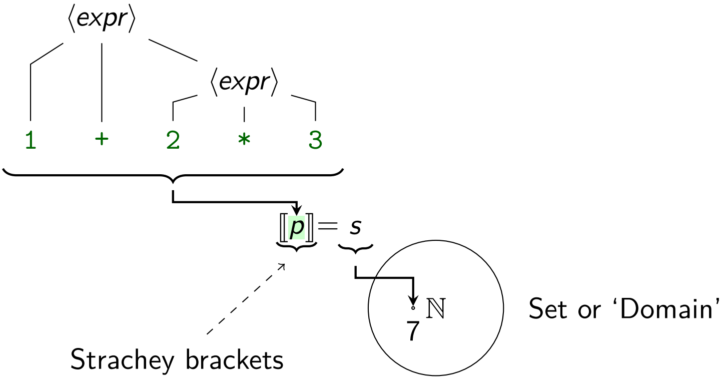
The principal schools of semantics:



Object and Meta-Language

- To describe the semantics of a language, we need another language
  - Informal descriptions: e.g., English
  - Formal descriptions: Some other formal language
- This 'description language' is called the *Meta-language*
- The language whose semantics we describe is called the *Object language*
- To abstract over concepts in the object language, we use Metavariables.

Denotational Semantics



- Maps program to mathematical object
- Equational theory to reason about programs

Directly maps program to its mathematical 'meaning'

Operational Semantics: The two branches

- Natural Semantics (Big-Step Semantics)
  - $p \Downarrow v$: p evaluates to v
  - Describes *complete* evaluation
  - Compact, useful to describe interpreters
- Structural Operational Semantics (Small-Step Semantics)
  - $p_1 \longrightarrow p_2$: p_1 evaluates one step to p_2
  - Captures individual *evaluation steps*
  - Verbose/detailed, useful for formal proofs

Axiomatic Semantics

Describe *statements*– not good fit for our current language

$$\{P\}\textit{statement}\{Q\}$$

- P: Precondition
- Q: Postcondition
- if P holds, then *statement* ensures that Q holds

Example:

$$\{x \geq 0\}\texttt{x := x + 1;}\{x > 0\}$$

Frequently used for "design-by-contract" software development

Comparison

- *Denotational Semantics*
  Equational theory, also describes nontermination
- *Natural Semantics*
  Compact, describes interpreter, doesn't give semantics to nonterminating programs
- *Structural Operational Semantics*
  Describes fully detailed evaluation strategy
- *Axiomatic Semantics*
  Describes effect of *statements* (before/after), no nontermination
- *Algebraic Semantics*
  Describes effect of *operations* on opaque data structures, no nontermination

Natural (Operational) Semantics

We relate expressions e to their results v:

$$e \Downarrow v$$

- $e \in \langle expr \rangle$
- $v \in \langle val \rangle$
- "e evaluates to n"

Semantics of STOL-B

$$n_1, n_2 \in \texttt{num}$$
$$v, v_1, v_2, v_3 \in \texttt{val}$$
$$e, e_1, e_2, e_3 \in \texttt{expr}$$

Informal use of meta-language '+' on numbers in object language (for brevity)

$$\frac{}{v \Downarrow v}\ (\mathit{val})$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad v = n_1 + n_2}{e_1\ \mathsf{plus}\ e_2 \Downarrow ?v}\ (\mathit{plus})$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \geq v_2}{e_1 >= e_2 \Downarrow \mathsf{true}}\ (\mathit{ge\text{-}true})$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 < v_2}{e_1 >= e_2 \Downarrow \mathsf{false}}\ (\mathit{ge\text{-}false})$$

$$\frac{e_1 \Downarrow \mathsf{true} \quad e_2 \Downarrow v_2}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_2}\ (\mathit{if\text{-}true})$$

$$\frac{e_1 \Downarrow \mathsf{false} \quad e_3 \Downarrow v_3}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_3}\ (\mathit{if\text{-}false})$$

Conditional Natural Semantics

$$\frac{\overbrace{P_1 \quad \cdots \quad P_n}^{\text{Preconditions}}}{\underbrace{e}_{\text{Program/Expression}} \Downarrow \underbrace{v}_{\text{Value}}}$$

If $P_1, \ldots, P_n$ all hold, then e evaluates to v.

- e: Program
- v: Irreducible result

Semantics of STOL-B

$$n_1, n_2 \in \texttt{num}$$
$$v, v_1, v_2, v_3 \in \texttt{val}$$
$$e, e_1, e_2, e_3 \in \texttt{expr}$$

Informal use of meta-language '+' on numbers in object language (for brevity)

$$\frac{}{v \Downarrow v}\ (\mathit{val})$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad v = n_1 + n_2}{e_1 \text{ plus } e_2 \Downarrow ?v}\ (\mathit{plus})$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \geq v_2}{e_1 >= e_2 \Downarrow \text{true}}\ (\mathit{ge\text{-}true})$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 < v_2}{e_1 >= e_2 \Downarrow \text{false}}\ (\mathit{ge\text{-}false})$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_2}\ (\mathit{if\text{-}true})$$

$$\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_3}\ (\mathit{if\text{-}false})$$

Evaluating a Program

$$\frac{}{v \Downarrow v}\ (val) \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad v = n_1 + n_2}{e_1 \text{ plus } e_2 \Downarrow v}\ (plus)$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \geq v_2}{e_1 >= e_2 \Downarrow \text{true}}\ (ge\text{-}true) \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 < v_2}{e_1 >= e_2 \Downarrow \text{false}}\ (ge\text{-}false)$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_2}\ (if\text{-}true) \qquad \frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_3}\ (if\text{-}false)$$

$$\frac{\dfrac{\dfrac{\overline{4 \Downarrow 4} \quad \overline{3 \Downarrow 3} \quad 7 = 4 + 3}{4 \text{ plus } 3 \Downarrow 7}\ (plus) \quad \dfrac{}{5 \Downarrow 5}\ (val) \quad 7 \geq 5}{4 \text{ plus } 3 >= 5 \Downarrow \text{true}}\ (ge\text{-}true) \quad \dfrac{}{1 \Downarrow 1}\ (val)}{\textbf{if } 4 \text{ plus } 3 >= 5 \textbf{ then } 1 \textbf{ else } 0 \Downarrow 1}\ (if\text{-}true)$$

Derivation (tree) / Proof (tree)

Connection to Datalog

$$\frac{P_1 \quad \ldots \quad P_k}{e \Downarrow v}$$

$$\textsc{EvalTo}(e, v) \text{ :- } P_1, \ldots, P_k$$

$\textsc{EvalTo} \subseteq \mathit{expr} \times \mathit{val}$

Summary

- Natural (big-step) operational semantics relates programs to their result
- Computational results: *values* (cannot be evaluted further)
- Simplest case:

$$\Downarrow \subseteq \mathit{expr} \times \mathit{val}$$

- Use evaluation rules expressed as Gentzen-style natural deduction *axioms* to describe semantics:

$$\frac{P_1 \quad \ldots \quad P_k}{e \Downarrow v}$$

- Computation of results recursively combines (natural deduction-style)

‘Bad Programs’ in STOL-B

$$\frac{}{v \Downarrow v}\ (val) \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad v = n_1 + n_2}{e_1 \text{ plus } e_2 \Downarrow v}\ (plus)$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \geq v_2}{e_1 >= e_2 \Downarrow \text{true}}\ (ge\text{-}true) \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 < v_2}{e_1 >= e_2 \Downarrow \text{false}}\ (ge\text{-}false)$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_2}\ (if\text{-}true) \qquad \frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v_3}\ (if\text{-}false)$$

$$\frac{}{\text{true plus } 1 \Downarrow ?}$$

- If no rule matches, there is no v such that $e \Downarrow v$
- Program is ‘stuck’
- Type systems should catch this problem

The Typing Relation

- We the set of types of STOL-B, $\mathbb{T}_{\textit{stol-b}} = \{\textsc{Bool}, \textsc{Nat}\}$:
  - $\textsc{Bool}$: Type of booleans (true, false)
  - $\textsc{Nat}$: Type of natural numbers (0, 1, , . . .)
- We can now type values:

$$\begin{array}{lcl} \text{true} & : & \textsc{Bool} \\ 23 & : & \textsc{Nat} \end{array}$$

- In other words, (:) is a binary relation:

$$(:) \subseteq \textit{val} \times \mathbb{T}_{\textit{stol-b}}$$

Types for Expressions (1/2)

- We extend (:) to expressions:

$$(:) \subseteq \mathit{expr} \times \mathbb{T}_{\mathit{stol\text{-}b}}$$

- Now we can type e.g.:

$$\text{39 plus 3} \quad : \quad \textsc{Nat}$$

- Motivation: build decidable type system
  - We say that e is *well-formed* whenever $e : \tau$ for some τ
  - Design goal: well-formed programs don't get stuck
  - Must build 'good' type system to guarantee this

Types for Expressions (2/2)

$$\frac{}{\mathsf{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\mathsf{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in \mathit{nat}}{v : \textsc{Nat}}\ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Nat} \quad e_2 : \textsc{Nat}}{e_1\ \mathsf{plus}\ e_2 : \textsc{Nat}}\ (\textit{t-plus}) \qquad \frac{e_1 : \textsc{Nat} \quad e_2 : \textsc{Nat}}{e_1 >= e_2 : \textsc{Bool}}\ (\textit{t-ge})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ (\textit{t-if})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \textsc{Nat} \quad e_3 : \textsc{Nat}}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \textsc{Nat}}\ (\textit{t-if-nat})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \textsc{Bool} \quad e_3 : \textsc{Bool}}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \textsc{Bool}}\ (\textit{t-if-bool})$$

(*if*) rule summarises (*if-nat*) and (*if-bool*) via *type variable*

Using the Typing Relation

- With $e : \tau$, we can have:
  1. Exactly one τ fits (we've computed a type):

     $$2 \text{ plus } 3 : \textsc{Nat}$$

  2. No τ fits (type error):

     Type error in true plus 0

- Some languages allow multiple τ that match a program fragment

Summary

- Type system relates expressions to types:

$$(:) \subseteq \mathit{expr} \times \mathbb{T}_{\mathit{stol\text{-}b}}$$

- Again uses axioms to describe rules for computing types:

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \; (\mathit{t\text{-}if})$$

- No type matches $\Rightarrow$ type error
- Goal: Well-typed program should not get 'stuck'

The ATL Syntax

$$
\begin{array}{lcl}
\textit{name} & ::= & \textit{id} \\
 & \mid & \textit{id}\ .\ \textit{id} \\
 & & \\
\textit{val} & ::= & \textit{num} \\
 & \mid & \langle \textit{name} \rangle \\
 & & \\
\textit{expr} & ::= & \langle \textit{val} \rangle \\
 & \mid & \langle \textit{expr} \rangle + \langle \textit{expr} \rangle \\
 & \mid & \mathsf{null} \\
 & \mid & \mathsf{print}\ \langle \textit{val} \rangle \\
 & \mid & \mathsf{new}(\langle \textit{init} \rangle \star) \\
 & \mid & \textit{id}\ (\ \langle \textit{val} \rangle \star\ ) \\
 & & \\
\textit{init} & ::= & \textit{id} : \langle \textit{expr} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\textit{stmt} & ::= & \langle \textit{name} \rangle\ \mathsf{=}\ \langle \textit{expr} \rangle \\
 & \mid & \{\ \langle \textit{stmt} \rangle \star\ \} \\
 & \mid & \mathsf{if}\ \langle \textit{expr} \rangle\ \langle \textit{stmt} \rangle\ \mathsf{else}\ \langle \textit{stmt} \rangle \\
 & \mid & \mathsf{while}\ \langle \textit{expr} \rangle\ \langle \textit{stmt} \rangle \\
 & \mid & \mathsf{skip} \\
 & \mid & \mathsf{return}\ \langle \textit{expr} \rangle \\
 & & \\
\textit{decl} & ::= & \mathsf{proc}\ \textit{id}\ (\ \textit{id} \star\ )\ \langle \textit{stmt} \rangle \\
 & \mid & \mathsf{global}\ \textit{id} = \textit{num} \\
 & & \\
\textit{prog} & ::= & \mathsf{decl} \star
\end{array}
$$

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & & \\
\mathit{val} & ::= & \mathit{num} \\
 & \mid & \langle \mathit{name} \rangle \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & \mid & \langle \mathit{expr} \rangle \mathsf{+} \langle \mathit{expr} \rangle \\
 & \mid & \mathsf{null}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle \ \mathsf{=} \ \langle \mathit{expr} \rangle \\
 & \mid & \{ \ \langle \mathit{stmt} \rangle \star \ \} \\
 & \mid & \mathsf{if} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \ \mathsf{else} \ \langle \mathit{stmt} \rangle \\
 & \mid & \mathsf{while} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \\
 & \mid & \mathsf{skip} \\
 & \mid & \mathsf{return} \ \langle \mathit{expr} \rangle
\end{array}
$$

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \texttt{null} \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle \texttt{+} \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle
\end{array}
\qquad
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle \texttt{ = } \langle \mathit{expr} \rangle \\
 & | & \{ \ \langle \mathit{stmt} \rangle \ \} \\
 & | & \texttt{if } \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \texttt{ else } \langle \mathit{stmt} \rangle \\
 & | & \texttt{while } \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \\
 & | & \texttt{skip} \\
 & | & \texttt{return } \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle \texttt{;} \ \langle \mathit{stmt} \rangle
\end{array}
$$

Operational Semantics for Variables

```
x = 1;
return x + 2
```

- To describe operational semantics, we want to decompose the above *stmt*
- Describe semantics of all *stmt*s, *expr*s, *val*s, *name*s within

$$\texttt{x + 2} \Downarrow ?$$

Environments

Evaluating variables: must remember their most recent assignment

Environment: $E : \textbf{\textit{id}} \nrightarrow \textbf{\textit{val}}$

- Environments are *partial functions* from names to values
- Also called *Stores*
- For convenience: $\mathbb{E} = id \nrightarrow val$, so

$$E : \mathbb{E}$$

Notation:

let $E' = [n \mapsto v]E$
then:

$$E'(x) = \begin{cases} v & \iff x = n \\ E(x) & \text{otherwise} \end{cases}$$

The Environment: Expressions

- We incorporate the environment into evaluation:

$$\frac{E(x) = v}{\langle E, x \rangle \Downarrow v}$$

- Our evaluation relation now incorporates an environment:

$$- \Downarrow - \subseteq (\mathbb{E} \times \mathit{expr}) \times \mathit{val}$$

- Environment access can be written in many ways:
  - $E(x) = v$
  - $E = \{x \mapsto v, \ldots\}$
  - $E = \{x : v\} \cup E'$
  - $E = E' + \{x := v\}$

Environments in Natural Semantics

$$\begin{array}{rcl} id & \in & name \\ v, v_i & \in & val \\ e, e_i & \in & expr \end{array}$$

$$\frac{}{\langle E, v \rangle \Downarrow v} \ (val) \qquad \frac{E(id) = v}{\langle E, id \rangle \Downarrow v} \ (id)$$

$$\frac{\langle E, e_1 \rangle \Downarrow v_1 \quad \langle E, e_2 \rangle \Downarrow v_2 \quad v = v_1 + v_2}{\langle E, e_1 \texttt{+} e_2 \rangle \Downarrow v} \ (plus)$$

$$\frac{\langle E, e \rangle \Downarrow v}{\langle E, id \texttt{ = } e \rangle \Downarrow \textbf{?}} \ (stmt\text{-}assign)$$

The Environment: Statements

- Statements don't normally produce an output value
  - Exception: **return**
- Must be able to update environment
- Second evaluation relation:

$$\Downarrow_s \subseteq (\mathbb{E} \times \mathit{stmt}) \times (\mathit{val} \uplus \mathbb{E})$$

$$\frac{\langle E, e \rangle \Downarrow v}{\langle E, id = e \rangle \Downarrow_s [id \mapsto v]E} \ (\mathit{assign})$$

$$\frac{\langle E, e \rangle \Downarrow v}{\langle E, \textbf{return}\ e \rangle \Downarrow_s v} \ (\mathit{return})$$

Environments in Natural Semantics

$id \in name \qquad v, v_i \in val \qquad e, e_i \in expr \qquad s, s_i \in stmt \qquad E, E', E'' \in \mathbb{E} \qquad Ev \in val \uplus \mathbb{E}$

$$\frac{\langle E, s\rangle \Downarrow_s E'}{\langle E, \{\ s\ \}\rangle \Downarrow_s E'}\ (block) \qquad \frac{\langle E, s_1\rangle \Downarrow_s E' \quad \langle E', s_2\rangle \Downarrow_s Ev}{\langle E, s_1;\ s_2\ \rangle \Downarrow_s Ev}\ (seq)$$

$$\frac{}{\langle E, \mathbf{skip}\rangle \Downarrow_s E}\ (skip) \qquad \frac{\langle E, e\rangle \Downarrow v}{\langle E, id = e\rangle \Downarrow_s [id \mapsto v]E}\ (assign)$$

$$\frac{\langle E, e\rangle \Downarrow v \quad v \in num \quad v \neq 0 \quad \langle E, s_1\rangle \Downarrow_s Ev}{\langle E, \mathbf{if}\ e\ s_1\ \mathbf{else}\ s_2\rangle \Downarrow_s Ev}\ (if\text{-}then)$$

$$\frac{\langle E, e\rangle \Downarrow 0 \quad \langle E, s_2\rangle \Downarrow_s Ev}{\langle E, \mathbf{if}\ e\ s_1\ \mathbf{else}\ s_2\rangle \Downarrow_s Ev}\ (if\text{-}else) \qquad \frac{\langle E, e\rangle \Downarrow 0}{\langle E, \mathbf{while}\ e\ s\rangle \Downarrow_s E}\ (while\text{-}done)$$

$$\frac{\langle E, e\rangle \Downarrow v \quad v \in num \quad v \neq 0 \quad \langle E, s\rangle \Downarrow_s E' \quad \langle E', \mathbf{while}\ e\ s\rangle \Downarrow_s E''}{\langle E, \mathbf{while}\ e\ s\rangle \Downarrow_s E''}\ (while\text{-}step)$$

$$\frac{\langle E, e\rangle \Downarrow v}{\langle E, \mathbf{return}\ e\rangle \Downarrow_s v}\ (return)$$

Summary

- Supporting variables (and updates) requires maintaining a mapping from names to values
- Introduce Environment: $E : \mathbb{E}$
  where $\mathbb{E} = id \rightarrow val$
- Extend $\Downarrow$ relation to be able to read from $\mathbb{E}$
- Introduce evaluation relation for statements:

$$\Downarrow_s \subseteq (\mathbb{E} \times stmt) \times (val \uplus \mathbb{E})$$

- Allows updating environment *or* returning value

The `while` loop

$$\textbf{while } 1 \; \{ \ldots \textbf{print } 0 \ldots \}$$

$$\frac{\langle E, e\rangle \Downarrow v \quad v \in num \quad v \neq 0 \quad \langle E, s\rangle \Downarrow_s E' \quad \langle E', \textbf{while } e \; s\rangle \Downarrow_s E''}{\langle E, \textbf{while } e \; s\rangle \Downarrow_s E''} \; (\textit{while-step})$$

- Infinite loop: no matching E'
- Natural semantics description insufficient for reasoning about nontermination

We sometimes need to reason about programs that don't terminate

Semantics and Nonterminating Code

- Natural semantics computes

$$\textit{stmt} \Downarrow \textit{final result}$$

- Idea: instead compute

$$\textit{stmt} \longrightarrow \textit{next step}$$

- One-step evaluation relation, overloaded:

$$\begin{array}{rcl} (\longrightarrow) & \subseteq & (\mathbb{E} \times \textit{stmt}) \times (\mathbb{E} \times \textit{stmt}) \\ (\longrightarrow) & \subseteq & (\mathbb{E} \times \textit{expr}) \times (\mathbb{E} \times \textit{expr}) \end{array}$$

Structural Operational Semantics

Structural Operational Semantics (1/3)

$id \in name \qquad v, v_i \in val \qquad e, e_i, e'_i \in expr$

$$\frac{E(id) = v}{\langle E, id \rangle \longrightarrow \langle E, v \rangle}\ (var)$$

$$\frac{\langle E, e_1 \rangle \longrightarrow \langle E, e'_1 \rangle}{\langle E, e_1 + e_2 \rangle \longrightarrow \langle E, e'_1 + e_2 \rangle}\ (plus\text{-}l) \qquad \frac{\langle E, e_2 \rangle \longrightarrow \langle E, e'_2 \rangle}{\langle E, e_1 + e_2 \rangle \longrightarrow \langle E, e_1 + e'_2 \rangle}\ (plus\text{-}r)$$

$$\frac{v = v_1 + v_2}{\langle E, v_1 + v_2 \rangle \longrightarrow \langle E, v \rangle}\ (plus\text{-}e)$$

$$\frac{\langle E, e \rangle \longrightarrow \langle E, e' \rangle}{\langle E, id = e \rangle \longrightarrow \langle E, id = e' \rangle}\ (assign\text{-}p)$$

$$\frac{}{\langle E, id = v \rangle \longrightarrow \langle [id \mapsto v]E, \mathbf{skip} \rangle}\ (assign\text{-}e)$$

Structural Operational Semantics (2/3)

$$\frac{}{\langle E, id = v \rangle \longrightarrow \langle [id \mapsto v]E, \textbf{skip} \rangle} \; (\textit{assign-e})$$

- Subtleties in the above:
  - v is implicitly required to be in *val*
  - Analogously to natural semantics, we assume that v is *irreducible*, i.e., there is no v' such that

$$v \longrightarrow v'$$

  - We transform the assignment into **skip**, for later clean-up:

$$\frac{\langle E, s_1 \rangle \longrightarrow \langle E', s_1' \rangle}{\langle E, s_1 \; ; \; s_2 \rangle \longrightarrow \langle E', s_1' \; ; \; s_2 \rangle} \; (\textit{seq-1})$$

$$\frac{}{\langle E, \textbf{skip} \; ; \; s \rangle \longrightarrow \langle E', s \rangle} \; (\textit{seq-skip})$$

Structural Operational Semantics (3/3)

Infinite loop with initial environment $E = \{x \mapsto 1\}$:

$$
\begin{array}{ll}
& \langle \{x \mapsto 1\}, \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \textbf{if}\ \mathsf{x}\ \{\ \{\ \mathsf{x = x + 1}\ \}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \}\ \}\ \textbf{else skip} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \textbf{if}\ \mathsf{1}\ \{\ \{\ \mathsf{x = x + 1}\ \}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \}\ \}\ \textbf{else skip} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \{\ \{\ \mathsf{x = x + 1}\ \}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \{\ \mathsf{x = x + 1}\ \}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \mathsf{x = x + 1}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \mathsf{x = 1 + 1}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 1\}, \mathsf{x = 2}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 2\}, \textbf{skip}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 2\}, \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \} \rangle \\
\longrightarrow & \langle \{x \mapsto 2\}, \textbf{if}\ \mathsf{x}\ \{\ \{\ \mathsf{x = x + 1}\ \}\ ;\ \textbf{while}\ \mathsf{x}\ \{\ \mathsf{x = x + 1}\ \}\ \}\ \textbf{else skip} \rangle \\
\longrightarrow & \cdots
\end{array}
$$

Can describe evaluation of non-terminating program

Comparison

- Structural Operational Semantics;
  - Also *Small-step semantics*
  - Model step-by-step evaluation
  - Closer to compiler / machine view of program
  - Not continuing at s (no s' with $s \longrightarrow s'$) means either *error* ('getting stuck') or *program finished*
  - Useful for formal proofs, reasoning about nontermination
- Natural Semantics;
  - Also *Big-step semantics*
  - Model full execution of language parts
  - Matches typical interpreter implementation
  - Not continuing at s (no s' with $s \Downarrow s'$) means *error* ('getting stuck')
  - Usually more compact than small-step semantics

If you describe both, remember to prove equivalence!

Review

- Basic Natural Semantics
- Basic Structural Operational Semantics
- Basic Type Systems

To be continued. . .

- More types and semantics