Static Semantics

CS 4301/6371: Advanced Programming Languages

Kevin W. Hamlen

Febrary 29, 2024

Introduction

Steps for designing a new programming language:

1. Formally define the syntax using BNF
2. Formally define operational or denotational semantics (or both)
3. Prove semantic equivalence if you have multiple semantics
4. Today: Formally define a *static semantics* (type theory)

Extending the Syntax

Let's add support for boolean variables to SIMPL:

arithmetic expressions	$a ::= n \mid v \mid a_1 \texttt{+} a_2 \mid a_1 \texttt{-} a_2 \mid a_1 \texttt{*} a_2$
boolean expressions	$b ::= \texttt{true} \mid \texttt{false} \mid v \mid a_1 \texttt{<=} a_2 \mid b_1 \texttt{\&\&} b_2 \mid b_1 \texttt{\|\|} b_2 \mid \texttt{!}b$
commands	$c ::= \texttt{skip} \mid c_1 \texttt{;} c_2 \mid v \texttt{:=} a \mid v \texttt{:=} b \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c$
variable names	v
integer constants	n

Q: Unfortunately there's a problem with this new grammar. What?

Extending the Syntax

Let's add support for boolean variables to SIMPL:

arithmetic expressions	$a ::= n \mid v \mid a_1 \texttt{+} a_2 \mid a_1 \texttt{-} a_2 \mid a_1 \texttt{*} a_2$
boolean expressions	$b ::= \texttt{true} \mid \texttt{false} \mid v \mid a_1 \texttt{<=} a_2 \mid b_1 \texttt{\&\&} b_2 \mid b_1 \texttt{||} b_2 \mid \texttt{!}b$
commands	$c ::= \texttt{skip} \mid c_1 \texttt{;} c_2 \mid v \texttt{:=} a \mid v \texttt{:=} b \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$
variable names	v
integer constants	n

Q: Unfortunately there's a problem with this new grammar. What?

A: It's ambiguous (recall definition of ambiguity).

Example: $\texttt{x := y}$ (Is $\texttt{y}$ a b or an a?)
Or even worse: $\texttt{y := true; x := y + }1$

Disambiguating the Syntax

How to fix? Three typical options:

1. Add extra syntax (e.g., $\texttt{Arith}(v)$ and $\texttt{Bool}(v)$ instead of v)
   - really annoying; programmers hate it!
2. Find an interpretation for everything (e.g., $\texttt{true} + 1 = 2$)
   - results in a chaotic language
   - bad for debugging, readability, maintainability, security, ...
3. The right solution: Coalesce the syntax and introduce a static semantics!

Coalesce the Syntax

expressions	$e ::= n \mid v \mid e_1 \texttt{+} e_2 \mid e_1 \texttt{-} e_2 \mid e_1 \texttt{*} e_2$
	$\mid \texttt{true} \mid \texttt{false} \mid e_1 \texttt{<=} e_2 \mid e_1 \texttt{\&\&} e_2 \mid e_1 \texttt{\|\|} e_2 \mid \texttt{!}e$
commands	$c ::= \texttt{skip} \mid c_1 \texttt{;} c_2 \mid v \texttt{:=} e \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c$
variable names	v
integer constants	n

Add Type Declarations

expressions	$e ::= n \mid v \mid e_1 \texttt{+} e_2 \mid e_1 \texttt{-} e_2 \mid e_1 \texttt{*} e_2$
	$\mid \texttt{true} \mid \texttt{false} \mid e_1 \texttt{<=} e_2 \mid e_1 \texttt{\&\&} e_2 \mid e_1 \texttt{||} e_2 \mid \texttt{!}e$
commands	$c ::= \texttt{skip} \mid c_1 \texttt{;} c_2 \mid v \texttt{:=} e \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{int}\ v \mid \texttt{bool}\ v$
variable names	v
integer constants	n

Declarations have *no effect* at runtime:

$$\frac{}{\langle \texttt{int}\ v, \sigma\rangle \to_1 \langle \texttt{skip}, \sigma\rangle} \qquad \frac{}{\langle \texttt{bool}\ v, \sigma\rangle \to_1 \langle \texttt{skip}, \sigma\rangle}$$

Many Stuck States

expressions	$e ::= n \mid v \mid e_1 \texttt{+} e_2 \mid e_1 \texttt{-} e_2 \mid e_1 \texttt{*} e_2$
	$\mid \texttt{true} \mid \texttt{false} \mid e_1 \texttt{<=} e_2 \mid e_1 \texttt{\&\&} e_2 \mid e_1 \texttt{\|\|} e_2 \mid \texttt{!}e$
commands	$c ::= \texttt{skip} \mid c_1 \texttt{;} c_2 \mid v \texttt{:=} e \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c \mid \texttt{int } v \mid \texttt{bool } v$
variable names	v
integer constants	n

Declarations have *no effect* at runtime:

$$\frac{}{\langle \texttt{int } v, \sigma\rangle \to_1 \langle \texttt{skip}, \sigma\rangle} \qquad \frac{}{\langle \texttt{bool } v, \sigma\rangle \to_1 \langle \texttt{skip}, \sigma\rangle}$$

We disambiguated the grammar, but now there are many stuck states!
Example: $\langle \texttt{true} \texttt{+} 3, \sigma\rangle$ (and of course we still have $\langle \texttt{x}, \bot\rangle$)

Intro to Static Semantics

Static Semantics: Deductive rules that, when combined with syntax restrictions, define the set of legal programs by precluding stuck states.

types	$\tau ::= int \mid bool$	
typing contexts	$\Gamma : v \rightharpoonup \tau$	
typing judgments	$\Gamma \vdash e : \tau$	"Γ proves that e has type τ"

Intro to Static Semantics

Static Semantics: Deductive rules that, when combined with syntax restrictions, define the set of legal programs by precluding stuck states.

types	$\tau ::= int \mid bool$	
typing contexts	$\Gamma : v \rightharpoonup (\tau \times \{T, F\})$	
typing judgments	$\Gamma \vdash e : \tau$	"Γ proves that e has type τ"

Intuition: $\Gamma(v) = (int, T)$ means v is an integer and is definitely initialized.

Primitive Typing Judgments

Define derivation rules that prove typing judgments. Easy ones:

$$\frac{}{\Gamma \vdash n : \mathit{int}} \text{(28)}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathit{bool}} \text{(29)}$$

$$\frac{}{\Gamma \vdash \mathtt{false} : \mathit{bool}} \text{(30)}$$

Typing Arithmetic Operations

$$\frac{?}{\Gamma \vdash e_1 \texttt{+} e_2 : ?}$$

Typing Arithmetic Operations

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \,\texttt{+}\, e_2 : int}$$

Typing Arithmetic Operations

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \, \texttt{+} \, e_2 : int}$$

Typing Arithmetic Operations

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

We need these premises!

Remember: The goal of a static semantics is to *preclude stuck states*, not infer a type for as many expressions as possible!

Rejecting bad programs *helps the programmer!*

Typing Boolean Operations

$$\frac{?}{\Gamma \vdash e_1 \;\texttt{\&\&}\; e_2 : ?}$$

Typing Boolean Operations

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \mathbin{\texttt{\&\&}} e_2 : bool}$$

Typing Comparisons

$$\frac{?}{\Gamma \vdash e_1 \; \texttt{<=} \; e_2 : ?}$$

Typing Comparisons

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \texttt{<=} e_2 : bool}$$

Typing Variable Reads

$$\frac{?}{\Gamma \vdash v : ?}$$

Typing Variable Reads

$$\frac{\Gamma(v) = (\tau, p)}{\Gamma \vdash v : \tau}$$

Typing Variable Reads

$$\frac{\Gamma(v) = (\tau, T)}{\Gamma \vdash v : \tau} \quad (31)$$

Typing Commands

Other rules for expressions are similar (see assignment).

Q: How do we type-check commands?

$$\Gamma \vdash c : ?$$

Typing Commands

Other rules for expressions are similar (see assignment).

Q: How do we type-check commands?

$$\Gamma \vdash c : \Gamma'$$

Typing Skip

$$\frac{}{\Gamma \vdash \texttt{skip} : \Gamma} (21)$$

Typing Sequence

$$\frac{?}{\Gamma \vdash c_1\,;c_2 : ?}$$

Typing Sequence

$$\frac{\Gamma \vdash c_1 : \Gamma_2 \qquad \Gamma_2 \vdash c_2 : \Gamma'}{\Gamma \vdash c_1 ; c_2 : \Gamma'} (24)$$

Typing Declarations

$$\frac{?}{\Gamma \vdash \texttt{int}\ v : ?}$$

Typing Declarations

$$\frac{}{\Gamma \vdash \texttt{int}\ v : \Gamma[v \mapsto (int, F)]}$$

Typing Declarations

$$\frac{v \notin \Gamma^{\leftarrow}}{\Gamma \vdash \texttt{int}\ v : \Gamma[v \mapsto (int, F)]} (22)$$

Typing Declarations

$$\frac{v \notin \Gamma^{\leftarrow}}{\Gamma \vdash \texttt{int}\ v : \Gamma[v \mapsto (int, F)]} \quad (22)$$

$$\frac{v \notin \Gamma^{\leftarrow}}{\Gamma \vdash \texttt{bool}\ v : \Gamma[v \mapsto (bool, F)]} \quad (23)$$

Typing Assignments

$$\frac{?}{\Gamma \vdash v \; \mathtt{:=} \; e : ?}$$

Typing Assignments

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash v \; \texttt{:=} \; e : ?}$$

Typing Assignments

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = (\tau, T)}{\Gamma \vdash v \mathrel{\texttt{:=}} e : \Gamma}$$

Typing Assignments

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = (\tau, p)}{\Gamma \vdash v \texttt{ := } e : \Gamma[v \mapsto (\tau, T)]} \quad (25)$$

Typing Conditionals

$$\frac{?}{\Gamma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : ?}$$

Typing Conditionals

$$\frac{\Gamma \vdash e : \mathit{bool} \qquad \Gamma \vdash c_1 : ? \qquad ? \vdash c_2 : ?}{\Gamma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : ?}$$

Typing Conditionals

$$\frac{\Gamma \vdash e : \mathit{bool} \qquad \Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : ?}$$

Typing Conditionals

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : \Gamma}$$

Typing Conditionals

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma} \text{(26)}$$

Optional Exercise: See if you can come up with a better choice than Γ.

- Your choice must not permit stuck states!
- But it should admit as many non-stuck programs as possible.

(But for the assignment, just implement the given rule.)

Typing Loops

Same strategy for loops:

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash c_1 : \Gamma_1}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c_1 : \Gamma} \quad (27)$$

(Not many better choices this time. Why?)

Devising Static Semantics

Definition (well-typed): A command c is *well-typed* if $\bot \vdash c : \Gamma'$ is derivable for some Γ'.

Definition (type-checker): A decision procedure for $\bot \vdash c : \Gamma'$ is called a *type-checker*.

Recall two possible interpretations of derivation rules:

- The rules form an implementation recipe for a type-checker.
- The rules extend propositional logic, allowing us to prove things about code (e.g., assuming a program is well-typed gives us extra reasoning power).

A good static semantics:

- Catches all (or most) stuck states before runtime (type-safety)
- Is deterministic!
    - Don't put operational/denotational semantics inside static semantics!
    - *"In order to find out whether the program is safe, first run the program ..."*
- Isn't so restrictive that it rules out important functionalities.