Static Semantics
CS 4301/6371: Advanced Programming Languages

Kevin W. Hamlen

February 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)
Let’s add support for boolean variables to SIMPL:

arithmetic expressions $a ::= n \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$

boolean expressions $b ::= true \mid false \mid v \mid a_1 \leq a_2 \mid b_1 \& \& b_2 \mid b_1 || b_2 \mid !b$

commands $c ::= skip \mid c_1;c_2 \mid v := a \mid v := b \mid if b then c_1 else c_2 \mid while b do c$

variable names $v$

integer constants $n$

Q: Unfortunately there’s a problem with this new grammar. What?
Let’s add support for boolean variables to SIMPL:

**arithmetic expressions**

\[ a ::= n \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \]

**boolean expressions**

\[ b ::= \text{true} \mid \text{false} \mid v \mid a_1 \leq a_2 \mid b_1 \& b_2 \mid b_1 \mid b_2 \mid !b \]

**commands**

\[ c ::= \text{skip} \mid c_1 ; c_2 \mid v := a \mid v := b \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \]

**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: \( x := y \) (Is \( y \) a \( b \) or an \( a \)?)

Or even worse: \( y := \text{true}; x := y + 1 \)
Disambiguating the Syntax

How to fix? Three typical options:

1. Add extra syntax (e.g., $\text{Arith}(v)$ and $\text{Bool}(v)$ instead of $v$)
   - really annoying; programmers hate it!

2. Find an interpretation for everything (e.g., $\text{true} + 1 = 2$)
   - 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

\[
e ::= n \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ast e_2 \\
\quad \mid \text{true} \mid \text{false} \mid e_1 \leq e_2 \mid e_1 \&\& e_2 \mid e_1 \mid\mid e_2 \mid !e
\]

commands

\[
c ::= \text{skip} \mid c_1 ; c_2 \mid v := e \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
\]

variable names

\[v\]

integer constants

\[n\]
Add Type Declarations

expressions
\[ e ::= n \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ast e_2 \]
\[ \mid \text{true} \mid \text{false} \mid e_1 \leq e_2 \mid e_1 \&\& e_2 \mid e_1 \| e_2 \mid !e \]

commands
\[ c ::= \text{skip} \mid c_1; c_2 \mid v ::= e \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \mid \text{int } v \mid \text{bool } v \]

variable names
\[ v \]

integer constants
\[ n \]

Declarations have no effect at runtime:

\[ \langle \text{int } v, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle \]
\[ \langle \text{bool } v, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle \]
Many Stuck States

expressions \[ e ::= n \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ast e_2 \]
\[ \mid \text{true} \mid \text{false} \mid e_1 \leq e_2 \mid e_1 \&\& e_2 \mid e_1 || e_2 \mid !e \]

commands \[ c ::= \text{skip} \mid c_1;c_2 \mid v := e \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \mid \text{int } v \mid \text{bool } v \]

variable names \[ v \]
integer constants \[ n \]

Declarations have no effect at runtime:

\[ \langle \text{int } v, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle \quad \quad \langle \text{bool } v, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle \]

We disambiguated the grammar, but now there are many stuck states!
Example: \[ \langle \text{true} + 3, \sigma \rangle \] (and of course we still have \[ \langle x, \bot \rangle \)
**Static Semantics:** Deductive rules that, when combined with syntax restrictions, define the set of legal programs by precluding stuck states.

**Types:**

\[ \tau ::= \text{int} \mid \text{bool} \]

**Typing Contexts:**

\[ \Gamma : v \rightarrow \tau \]

**Typing Judgments:**

\[ \Gamma \vdash e : \tau \]

“\( \Gamma \) proves that \( e \) has type \( \tau \)”
**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

\[ \tau ::= \text{int} \mid \text{bool} \]

Typing contexts

\[ \Gamma : v \rightarrow (\tau \times \{T, F\}) \]

Typing judgments

\[ \Gamma \vdash e : \tau \]

"\( \Gamma \) proves that \( e \) has type \( \tau \)"

Intuition: \( \Gamma(v) = (\text{int}, T) \) means \( v \) is an integer and is definitely initialized.
Primitive Typing Judgments

Define derivation rules that prove typing judgments. Easy ones:

\[
\Gamma \vdash n : int^{(28)}
\]

\[
\Gamma \vdash true : bool^{(29)}
\]

\[
\Gamma \vdash false : bool^{(30)}
\]
Typing Arithmetic Operations

\[ \Gamma \vdash e_1 + e_2 : ? \]
Typing Arithmetic Operations

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]

\[ \Gamma \vdash e_1 + e_2 : \text{int} \]
Typing Arithmetic Operations

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash e_1 + e_2 : \text{int}
\]
Typing Arithmetic Operations

\[
\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \\
\hline
\Gamma \vdash e_1 + e_2 : int
\]

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

\[ \Gamma \vdash e_1 \&\& e_2 : ? \]
Typing Boolean Operations

\[
\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool
\]

\[
\Gamma \vdash e_1 \&\& e_2 : bool
\]
Typing Comparisons

\[ \Gamma \vdash e_1 \leq e_2 : ? \]
Typing Comparisons

\[ \Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \]

\[ \Gamma \vdash e_1 \leq e_2 : bool \]
Typing Variable Reads

\[ ? \]

\[ \Gamma \vdash v : ? \]
Typing Variable Reads

\[
\Gamma(v) = (\tau, p) \quad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \tau}
\]
Typing Variable Reads

\[ \Gamma(v) = (\tau, T) \]

\[ \Gamma \vdash v : \tau \]
Typing Commands

Other rules for expressions are similar (see assignment).

Q: How do we type-check commands?

\[ \Gamma \vdash c : ? \]
Other rules for expressions are similar (see assignment).

Q: How do we type-check commands?

\[ \Gamma \vdash c : \Gamma' \]
Typing Skip

\[ \Gamma \vdash \text{skip} : \Gamma \]
Typing Sequence

\[ \Gamma \vdash c_1 ; c_2 : ? \]
Typing Sequence

\[ \Gamma \vdash c_1 : \Gamma_2 \quad \Gamma_2 \vdash c_2 : \Gamma' \]

\[ \Gamma \vdash c_1 ; c_2 : \Gamma' \quad \text{(24)} \]
Typing Declarations

\[ \Gamma \vdash \text{int } v : ? \]
Typing Declarations

\[ \Gamma \vdash \text{int } v : \Gamma[v \mapsto (\text{int}, F)] \]
Typing Declarations

\[
\frac{v \not\in \Gamma^{-}}{\Gamma \vdash \text{int } v : \Gamma[v \mapsto (\text{int}, F)]}^{(22)}
\]
Typing Declarations

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

\[
\frac{v \notin \Gamma}{\Gamma \vdash \text{bool } v : \Gamma[v \mapsto (\text{bool}, F)\]}^{(23)}
\]
Typing Assignments

\[ \Gamma \vdash v := e : ? \]
Typing Assignments

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash v := e : ? \]
Typing Assignments

\[
\Gamma \vdash e : \tau \quad \Gamma(v) = (\tau, T) \\
\hline 
\Gamma \vdash v := e : \Gamma
\]
Typing Assignments

\[
\frac{\Gamma \vdash e : \tau \quad \Gamma(v) = (\tau, p)}{\Gamma \vdash v := e : \Gamma[v \mapsto (\tau, T)]}
\tag{25}
\]
Typing Conditionals

\[ \Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : ? \]
Typing Conditionals

\[
\Gamma \vdash e : bool \quad \Gamma \vdash c_1 : ? \quad ? \vdash c_2 : ? \\
\overline{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : ?}
\]
Typing Conditionals

\[ \Gamma \vdash e : bool \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2 \]

\[ \Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : ? \]
Typing Conditionals

\[ \Gamma \vdash e : \text{bool} \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2 \]

\[ \Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \Gamma \]
Typing Conditionals

\[ \Gamma \vdash e : bool \quad \Gamma \vdash c_1 : \Gamma_1 \quad \Gamma \vdash c_2 : \Gamma_2 \]

\[ \Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \Gamma \quad (26) \]

Optional Exercise: See if you can come up with a better choice than \( \Gamma \).

- 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:

\[
\Gamma \vdash e : bool \quad \Gamma \vdash c_1 : \Gamma_1 \quad \text{(27)}
\]

\[
\Gamma \vdash \text{while } e \text{ do } c_1 : \Gamma
\]

(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 $\Gamma'$.

**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.