Introduction

Formal definitions of programming languages have three parts:

1. **Abstract syntax**: deep structure and binding.
2. **Static semantics**: typing rules.
3. **Dynamic semantics**: execution rules.

Latter two parts reflect the **phase distinction** between static and dynamic phases of processing found in most languages.

Phase Distinction

**Static phase**:
- Parsing and type checking
- Ensuring program is well-formed

**Dynamic phase**:
- Execution of well-formed programs

A language is **safe** exactly when well-formed programs are well-behaved when executed.

Formalizing Safety

Central theme of book and course is a rigorous treatment of type safety.

1. Formal definition of type safety.
2. Proving a language safe.
3. Relation to informal notions of safety.

The Consensus View

Pattern for book and modern study of programming languages

1. Programming languages organized as **collections of types**
2. Language “features” as operations associated with particular types
3. Types given meaning by static and dynamic semantics
4. Tied together, and shown to be well defined, by type safety proof

Static Semantics

The static phase is specified by **static semantics**:

- Rules for deriving **typing judgements**, determining when expressions of given types are well-formed

Types mediate interaction between parts of a program:

- By “predicting” aspects of parts’ execution behavior, giving assurance that parts will fit properly at run-time

Type safety means predictions are accurate; otherwise semantics is deemed ill-defined and language deemed unsafe for execution.
Abstract Syntax for $\mathcal{L}\{\text{num str}\}$

Two syntactic categories, augmented by two other inductively defined categories of objects (natural numbers and strings):

- $\tau \ ::= \text{num} \mid \text{str}$
- $e \ ::= \, x \mid \text{num}[n] \mid \text{str}[s] \mid \text{plus}(e_1; e_2) \mid \text{times}(e_1; e_2) \mid \text{cat}(e_1; e_2) \mid \text{len}(e) \mid \text{let}(e_1; x; e_2)$

This defines two classes of abstract binding trees specified by two judgement forms:

- $\tau$ type defining category of types
- $e$ expr defining category of expressions

Static Semantics

The static semantics, or type system, imposes context-sensitive restrictions on the formation of expressions.

- For example, $\text{plus}(x; \text{num}[n])$ is sensible exactly if $x$ has type $\text{num}$ in the surrounding context; in fact, this is the only relevant kind of contextual information for static semantics.
- Distinguishes well-typed from ill-typed expressions.
- Type constraints eliminate prima facie non-sensical programs.

Typing Judgements

Static semantics is given by rules inductively defining of a family of three part (parametric) hypothetical typing judgements: $\chi \vdash e : \tau$

1. A type assignment, or type context, $\chi$ that consists of hypotheses of the form $x : \tau$, one for each $x \in X$, where $X$ is a finite set of variables (usually not explicitly mentioned). A variable $x$ is fresh for $\chi$, $x \not\in \chi$, if no hypothesis $x : \tau$ is in $\chi$.
2. An expression $e$ whose free variables are given types by $\chi$.
3. A type $\tau$ for the expression $e$.

Typing Rules for $\mathcal{L}\{\text{num str}\}$

A variable has whatever type $\chi$ assigns to it:

$$\chi, x : \tau \vdash x : \tau$$

Constants have the evident types:

$$\begin{align*}
\chi & \vdash \text{num}[n] : \text{num} \\
\chi & \vdash \text{str}[s] : \text{str}
\end{align*}$$

Typing Rules for $\mathcal{L}\{\text{num str}\}$

The primitive operations have the expected typing rules:

$$\begin{align*}
\chi & \vdash e_1 : \text{num} \quad \chi & \vdash e_2 : \text{num} \\
\chi & \vdash \text{plus}(e_1; e_2) : \text{num} \\
\chi & \vdash \text{times}(e_1; e_2) : \text{num} \\
\chi & \vdash \text{cat}(e_1; e_2) : \text{str} \\
\chi & \vdash \text{len}(e) : \text{num} \\
\chi & \vdash \text{let}(e_1; x; e_2) : \text{str}
\end{align*}$$
Typing Rules for $L\{\text{num str}\}$

Type checking let expressions:

$\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2$

$\Gamma \vdash \text{let}(e_1, x, e_2) : \tau_2$

Well-Typed and Ill-Typed Expressions

An expression $e$ is well-typed, or typable, in a context $\Gamma$ iff there exists a type $\tau$ such that $\Gamma \vdash e : \tau$.

If there is no $\tau$ such that $\Gamma \vdash e : \tau$, then $e$ is ill-typed, or untypable, in context $\Gamma$.

Type Checking

In practice we use computers to find typing proofs. This is the job of a type checker:

- Given $\Gamma$, $e$, and $\tau$, is there a derivation of $\Gamma \vdash e : \tau$ according to the typing rules?

- How does the type checker find typing proofs?

Important fact: the typing rules for $L\{\text{num str}\}$ are syntax-directed — there is one rule per expression form.

Therefore the checker can invert the typing rules and work backwards towards the proof.

Properties of Typing for $L\{\text{num str}\}$

Lemma 2 (Inversion for Typing)
The typing rules are necessary, as well as sufficient. That is:

1. If $\Gamma \vdash x : \tau$, then $x : \tau$ occurs in $\Gamma$.
2. If $\Gamma \vdash \text{num}(n) : \tau$, then $\tau = \text{num}$.
3. If $\Gamma \vdash \text{str}(s) : \tau$, then $\tau = \text{str}$.
4. If $\Gamma \vdash \text{plus}(e_1, e_2) : \tau$, then $\tau$ is $\text{num}$ and $\Gamma \vdash e_1 : \text{num}$ and $\Gamma \vdash e_2 : \text{num}$.
5. If $\Gamma \vdash \text{times}(e_1, e_2) : \tau$, then $\tau$ is $\text{num}$ and $\Gamma \vdash e_1 : \text{num}$ and $\Gamma \vdash e_2 : \text{num}$.
6. If $\Gamma \vdash \text{cat}(e_1, e_2) : \tau$, then $\tau$ is $\text{str}$ and $\Gamma \vdash e_1 : \text{str}$ and $\Gamma \vdash e_2 : \text{str}$.
7. If $\Gamma \vdash \text{len}(e) : \tau$, then $\tau$ is $\text{num}$ and $\Gamma \vdash e : \text{str}$.
8. If $\Gamma \vdash \text{let}(e_1, x, e_2) : \tau$, then there exist $\tau_1$ such that $e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau$.
Induction on Typing for $\mathcal{L}\{\text{num}, \text{str}\}$

To show that some property $P(\Gamma, e, \tau)$ holds whenever $\Gamma \vdash e : \tau$, as for example in the preceding lemmas, it is enough to show

- $P(\Gamma, e, \tau)$
- $P(\Gamma, \text{num}[n], \text{num})$
- $P(\Gamma, \text{str}[s], \text{str})$
- if $P(\Gamma, e, \text{num})$ and $P(\Gamma, e', \text{num})$, then $P(\Gamma, \text{plus}(e, e'), \text{num})$
- if $P(\Gamma, e, \text{str})$ and $P(\Gamma, e', \text{str})$, then $P(\Gamma, \text{cat}(e, e'), \text{str})$
- if $P(\Gamma, e, \tau)$ and $P(\Gamma, e', \tau)$, then $P(\Gamma, \text{let}(e, e'), \tau)$

Structural Properties of Typing

The following decomposition property is the converse of the substitution property.

**Lemma 5 (Decomposition)**

If $\Gamma \vdash [e/x]e' : \tau'$, then for every type $\tau$ such that $\Gamma \vdash e : \tau$ we have $\Gamma, x : \tau \vdash e' : \tau'$.

**Proof:** Follows directly from Unicity of Types.

Dynamic Semantics

The dynamic semantics of a language specifies how to execute programs written in that language.

Two general approaches:

1. **Machine-based:** describe execution in terms of a mapping of the language onto a (concrete or abstract) machine.

2. **Language-based:** describe execution entirely in terms of the language itself.

Machine-Based Models

Historically, machine-based approaches have dominated.

- Assembly languages.
- Systems languages such as C and its derivatives.

Such languages are sometimes called **concrete** languages because of their close association with the machine.

Machine-Based Models

Advantages:

- Specifies meanings of data types in terms of machine-level concepts.
- Facilitates low-level programming, e.g., writing device drivers.
- Supports low-level "hacks" based on the quirks of the target machine.
Machine-Based Models

Disadvantages:

• Requires you to understand how a language is compiled.
• Inhibits portability.
• Run-time errors (such as “bus error”) cannot be understood in terms of the program, only in terms of how it is compiled and executed.

Language-Based Models

Advantages:

• Inherently portable across platforms.
• Semantics is defined entirely in terms of concepts within the language.
• No mysterious (implementation-specific) errors to track down.

Disadvantages:

• Cannot take advantage of machine-specific details.
• Can be more difficult to understand complexity (time and space usage).

Machine- vs. Language-Based Models

Language-based models will dominate in the future.

• Low-level programming is a vanishingly small percentage of the mix.
• Emphasis on bit-level efficiency is almost always misplaced.
• Portability matters much more than efficiency.

Dynamic Semantics - Version 1

Initially we’ll define the dynamic semantics of $L(\text{num,str})$ using a technique called structural semantics.

• Define a transition relation between states. For $L(\text{num,str})$, states are closed expressions, all of which are initial states.
• A transition consists of execution of a single instruction.
• Rules determine which instruction to execute next.
• There are no transitions from closed values, which are the final states for $L(\text{num,str})$. 
Values

The set of closed values is inductively defined by the following rules:

\[
\begin{align*}
\text{num}[n] & \rightarrow \\
\text{str}[s] & \rightarrow 
\end{align*}
\]

Instruction Transitions

First, we inductively define the instruction transitions of \( L(\text{num}, \text{str}) \). These are the atomic transition steps.

- Primitive operations on numbers.
- Primitive operations on strings.
- Evaluation of a let expression.

Instruction Transitions

Addition of two numbers:

\[
\begin{align*}
\text{num}[n_1] + \text{num}[n_2] & \rightarrow \text{num}[n] \\
\text{num}[n_1] \times \text{num}[n_2] & \rightarrow \text{num}[n] \\
\text{str}[s_1] + \text{str}[s_2] & \rightarrow \text{str}[s] \\
\text{str}[s_1] \times \text{str}[s_2] & \rightarrow \text{str}[s] \\
\end{align*}
\]

Instruction Transitions

Concatenation of two strings:

\[
\begin{align*}
\text{cat}[\text{str}[s_1], \text{str}[s_2]] & \rightarrow \text{str}[s] \\
\text{let}[c_1, x, c_2] & \rightarrow [c_1, x] c_2
\end{align*}
\]

Instruction Transitions

Evaluation of a let expression:

\[
\begin{align*}
\text{let}[c_1, x, c_2] & \rightarrow [c_1, x] c_2
\end{align*}
\]

This is the "by value" interpretation: bound variable stands for the expression itself, which is evaluated as many times (including zero) as it occurs in \( c_2 \).

Instruction Transitions

Evaluation of a let expression:

\[
\begin{align*}
\text{let}[c_1, x, c_2] & \rightarrow [c_1, x] c_2
\end{align*}
\]

This is the "by name" interpretation: bound variable stands for the expression itself, which is evaluated as many times (including zero) as it occurs in \( c_2 \).

Instruction Transitions

Evaluation of a let expression:

\[
\begin{align*}
\text{let}[c_1, x, c_2] & \rightarrow [c_1, x] c_2
\end{align*}
\]

This is the "by name" interpretation: bound variable stands for the expression itself, which is evaluated as many times (including zero) as it occurs in \( c_2 \).

Type-Based Approach to Programming Languages

- Language constructs arise as introductory and eliminator forms associated with types.
- Elimination forms must be inverse to introduction forms. Elimination forms associated with a type have a principal argument, which must be of the type, to which the elimination form is the inverse.
Type-Based Approach to Programming Languages

• The principal argument of an elimination form is necessarily evaluated to an introduction form, exposing an opportunity for cancellation according to the conservation principle.

• It is more or less arbitrary whether non-principal arguments to an elimination form are evaluated prior to cancellation.

Search Transitions

Second, we specify the next instruction to execute by a set of rules that inductively define search transitions. These rules specify the order of evaluation of expressions: which instruction is to be executed next?

Assembly language programs are linear sequences of instructions; for these languages a simple counter (the PC) determines the next instruction.

For more structured languages such as \( L\{\text{asm \ axt} \} \) more complex rules are required.

The arguments of the primitive operations are evaluated left-to-right:

\[
\begin{align*}
\text{plus}(e_1; e_2) & \Rightarrow \text{plus}(e_1'; e_2) \\
\text{val}(e_1; e_2) & \Rightarrow \text{val}(e_1'; e_2) \\
\text{plus}(e_1; e_2) & \Rightarrow \text{plus}(e_1; e_2') \\
\text{cat}(e_1; e_2) & \Rightarrow \text{cat}(e_1; e_2')
\end{align*}
\]

and similarly for \( \text{times}(e_1; e_2) \)

Search Transitions

The arguments of the primitive operations are evaluated left-to-right:

\[
\begin{align*}
\text{let}(e_1; x; e_2) & \Rightarrow \text{let}(e_1'; x; e_2) \\
\text{cat}(e_1; e_2) & \Rightarrow \text{cat}(e_1; e_2')
\end{align*}
\]

Search Transitions

In the “by value” interpretation, let expressions are evaluated left-to-right: first the expression to be substituted, then the substitution.

\[
\begin{align*}
\text{let}(e_1; x; e_2) & \Rightarrow \text{let}(e_1'; x; e_2)
\end{align*}
\]

Induction on Evaluation

Since the transition judgement for structural semantics is inductively defined, there is an associated principle of induction, called induction on evaluation.

To prove that \( e \Rightarrow e' \) implies \( P(e, e') \) for some property \( P \), it suffices to prove that \( P \) is closed under the rules defining the transition judgement.

1. \( P(e, e') \) holds for each of the instruction axioms.

2. Assuming \( P \) holds for each of the premises of a search rule, show that it holds for the conclusion as well.
Elementary Properties of Evaluation

Lemma 6 (Determinacy)

If $e \rightarrow e'$ and $e \rightarrow e''$, then $e'$ and $e''$ are $\alpha$-equivalent.

Proof: By simultaneous induction of the two premises. The key observation is that only one rule applies for a given $e$, from which the result follows easily by induction in each case. ■

Stuck States

Not every irreducible expression is a value!

Observe that this expression is ill-typed:

\[
\text{plus}([^\text{sw}[?]}; \text{str}[^\text{ok}]) \not\rightarrow
\]

An expression $e$ that is not a value, but for which there exists no $e'$ such that $e \rightarrow e'$ is said to be stuck.

Safety: all stuck expressions are ill-typed. Equivalently, well-typed expressions do not get stuck.

Summary

1. The static semantics of the simple expression language is specified by an inductive definition of the typing judgement $\Gamma \vdash e : \tau$.

2. Properties of the type system may be proved by induction on typing derivations.

3. Structural semantics is a language-based model of computation: no mention of a mapping onto a machine.