

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.

1

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.

2

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.

3

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

4

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.

5

Abstract Syntax for $\mathcal{L}_{\{\text{num str}\}}$

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

Type $\tau ::= \text{num} \mid \text{str}$

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

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

- τ type defining category of types
- e exp defining category of expressions

6

Abstract and Concrete Syntax for $\mathcal{L}_{\{\text{num str}\}}$

Category	Item	Abstract	Concrete
Type	τ	$::= \text{num}$	num
		$\mid \text{str}$	str
Expr	e	$::= x$	x
		$\mid \text{num}[n]$	n
		$\mid \text{str}[s]$	$"s"$
		$\mid \text{plus}(e_1; e_2)$	$e_1 + e_2$
		$\mid \text{times}(e_1; e_2)$	$e_1 * e_2$
		$\mid \text{cat}(e_1; e_2)$	$e_1 \hat{\ } e_2$
		$\mid \text{len}(e)$	$ e $
	$\mid \text{let}(e_1; x.e_2)$	$\text{let } x \text{ be } e_1 \text{ in } e_2$	

Implicitly specifies two signatures, Ω_{type} and Ω_{expr} .

7

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 **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.

8

Typing Judgements

Static semantics is given by rules inductively defining of a family of three part (parametric) hypothetical **typing judgements**:

$$\mathcal{X} \mid \Gamma \vdash e : \tau$$

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

9

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

A variable has whatever type Γ assigns to it:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

Constants have the evident types:

$$\frac{}{\Gamma \vdash \text{num}[n] : \text{num}}$$

$$\frac{}{\Gamma \vdash \text{str}[s] : \text{str}}$$

10

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

The primitive operations have the expected typing rules:

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{plus}(e_1; e_2) : \text{num}}$$

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{times}(e_1; e_2) : \text{num}}$$

$$\frac{\Gamma \vdash e_1 : \text{str} \quad \Gamma \vdash e_2 : \text{str}}{\Gamma \vdash \text{cat}(e_1; e_2) : \text{str}}$$

$$\frac{\Gamma \vdash e : \text{str}}{\Gamma \vdash \text{len}(e) : \text{num}}$$

11

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

Type checking let expressions:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let}(e_1; x.e_2) : \tau_2}$$

12

Type-Based Approach to Programming Languages

- **Language constructs** arise as **introductory and eliminatory 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.
- Values of the type have introductory form

13

Well-Typed and Ill-Typed Expressions

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

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

14

Type Checking

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

Given Γ , e , and τ , 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 $\mathcal{L}_{\{\text{num str}\}}$ are **syntax-directed** — there is **one** rule per expression form.

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

15

Type Checking for $\mathcal{L}_{\{\text{num str}\}}$

We can say something even stronger for $\mathcal{L}_{\{\text{num str}\}}$: every expression has **at most one** type.

Lemma 1 (Unicity of Typing)

For every typing context Γ and expression e , there exists at most one τ such that $\Gamma \vdash e : \tau$.

Proof: The proof proceeds by rule induction on the typing rules for $\mathcal{L}_{\{\text{num str}\}}$.

■

16

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

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 Γ .
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 $\tau = \text{num}$ 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 $\tau = \text{num}$ 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 $\tau = \text{str}$ and $\Gamma \vdash e_1 : \text{str}$ and $\Gamma \vdash e_2 : \text{str}$.
7. If $\Gamma \vdash \text{len}(e) : \tau$, then $\tau = \text{num}$ and $\Gamma \vdash e : \text{str}$.
8. If $\Gamma \vdash \text{let}(e_1; x.e_2) : \tau$, then there exists τ_1 such that $e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau$.

17

Induction on Typing for $\mathcal{L}\{\text{num str}\}$

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

- $\mathcal{P}(\Gamma, x, \Gamma(x))$
- $\mathcal{P}(\Gamma, \text{num}[n], \text{num})$
- $\mathcal{P}(\Gamma, \text{str}[s], \text{str})$
- if $\mathcal{P}(\Gamma, e_1, \text{num})$ and $\mathcal{P}(\Gamma, e_2, \text{num})$, then $\mathcal{P}(\Gamma, \text{plus}(e_1; e_2), \text{num})$
- if $\mathcal{P}(\Gamma, e_1, \text{num})$ and $\mathcal{P}(\Gamma, e_2, \text{num})$, then $\mathcal{P}(\Gamma, \text{times}(e_1; e_2), \text{num})$
- if $\mathcal{P}(\Gamma, e_1, \text{str})$ and $\mathcal{P}(\Gamma, e_2, \text{str})$, then $\mathcal{P}(\Gamma, \text{cat}(e_1; e_2), \text{str})$
- if $\mathcal{P}(\Gamma, e, \text{str})$, then $\mathcal{P}(\Gamma, \text{len}(e), \text{num})$
- if $\mathcal{P}(\Gamma, e_1, \tau_1)$ and $\mathcal{P}(\Gamma, x : \tau_1, e_2, \tau)$, then $\mathcal{P}(\Gamma, \text{let}(e_1; x.e_2), \tau)$.

18

Structural Properties of Typing

Static semantics enjoy the structural properties of hypothetical and parametric judgements. In particular:

Lemma 3 (Weakening)

If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \# \Gamma$ and any τ type.

Proof: By induction on the derivation of $\Gamma \vdash e' : \tau'$. ■

Lemma 4 (Substitution)

If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$

Proof: By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. ■

19

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 τ such that $\Gamma \vdash e : \tau$ we have $\Gamma, x : \tau \vdash e' : \tau'$.

Proof: Follows directly from Unicity of Types. ■

20

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.

21

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.

22

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.

23

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.

24

Language-Based Models

Define execution behavior entirely at the level of the language itself.

“Computation by calculation.”

No need to specify implementation details.

Such languages are sometimes called **abstract** languages because they abstract from machine-specific details.

25

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.

26

Language-Based Models

Disadvantages:

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

27

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.

28

Dynamic Semantics - Version 1

Initially we'll define the dynamic semantics of $\mathcal{L}\{\text{num str}\}$ using a technique called **structural semantics**.

- Define a **transition relation** between states. For $\mathcal{L}\{\text{num str}\}$, 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 $\mathcal{L}\{\text{num str}\}$.

29

Values

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

$$\frac{}{\text{num}[n] \text{ val}}$$

$$\frac{}{\text{str}[s] \text{ val}}$$

30

Instruction Transitions

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

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

31

Instruction Transitions

Addition of two numbers:

$$\frac{n_1 + n_2 = n \text{ nat}}{\text{plus}(\text{num}[n_1]; \text{num}[n_2]) \mapsto \text{num}[n]}$$

and similarly for $\text{times}(\text{num}[n_1]; \text{num}[n_2])$

Concatenation of two strings:

$$\frac{s_1 \hat{\ } s_2 = s \text{ str}}{\text{cat}(\text{str}[s_1]; \text{str}[s_2]) \mapsto \text{str}[s]}$$

and similarly for $\text{len}(\text{str}[s])$

32

Instruction Transitions

Evaluation of a let expression:

$$\frac{}{\text{let}(e_1; x.e_2) \mapsto [e_1/x]e_2}$$

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 e_2 .

33

Instruction Transitions

Evaluation of a let expression:

$$\frac{e_1 \text{ val}}{\text{let}(e_1; x.e_2) \mapsto [e_1/x]e_2}$$

This is the "by value" interpretation: expression bound to variable is evaluated once and then bound variable is replaced by its corresponding value everywhere, and as many times (including zero) as, it occurs in e_2 .

34

Type-Based Approach to Programming Languages

- **Language constructs** arise as **introductory and eliminatory 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.

35

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.

36

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 $\mathcal{L}\{\text{num str}\}$ more complex rules are required.

37

Search Transitions

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

$$\frac{e_1 \mapsto e'_1}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e'_1; e_2)}$$

$$\frac{e_1 \text{ val } e_2 \mapsto e'_2}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e'_2)}$$

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

38

Search Transitions

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

$$\frac{e_1 \mapsto e'_1}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e'_1; e_2)}$$

$$\frac{e_1 \text{ val } e_2 \mapsto e'_2}{\text{cat}(e_1; e_2) \mapsto \text{cat}(e_1; e'_2)}$$

39

Search Transitions

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

$$\frac{e_1 \mapsto e'_1}{\text{let}(e_1; x.e_2) \mapsto \text{let}(e'_1; x.e_2)}$$

40

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 $e \mapsto e'$ implies $\mathcal{P}(e, e')$ for some property \mathcal{P} , it suffices to prove that \mathcal{P} is closed under the rules defining the transition judgement.

1. $\mathcal{P}(e, e')$ holds for each of the instruction axioms.
2. Assuming \mathcal{P} holds for each of the premises of a search rule, show that it holds for the conclusion as well.

41

Elementary Properties of Evaluation

Lemma 6 (Determinacy)

If $e \mapsto e'$ and $e \mapsto e''$, then e' and e'' are α -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. ■

42

Stuck States

Not every irreducible expression is a value!

`plus(num[7]; str[abc]) ↯`

Observe that this expression is ill-typed.

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

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

43

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.

44