

Meeting 17 - Static Typing

Dakota Bryan

Tuesday, October 22, 2024

Meeting 17 - Static Typing






© 1986 Universal Press Syndicate

ARE YOU BRINGING ME
HOME ANY PRESENTS
TONIGHT? ... NO? WELL,
JUST THOUGHT I'D ASK...



Links

-  In-Class Slides
-  In-Class Jupyter
-  Book Chapter

Announcements

- Prof. Chang is delivering Thursday's (10/24) lecture remotely, still come to class as it will be interactive
- Lab 4 due Mon 10/28

Today

- Triage your questions
 - Will do my best to answer all that I can
- **Static Typing**
 - Using a subset of JavaScripty, we will review abstract syntax, small-step semantics, and dynamic typing
 - Using the same language, we will discuss static typing, typing rules, and implement a new language with a static type checker

Static Typing

Motivation

So far, we have handled mismatched types with coercions and dynamic type errors

- Both of these implementations have painpoints, what are some for each?

Static typing checking alleviates these issues by handling type errors prior to running the program

JavaScripty with Numbers and Functions - Review

Syntax

Syntax for JavaScripty with only anonymous functions, function calls, and numbers:

values	v	$::=$	$n \mid (x) \Rightarrow e_1$
expressions	e	$::=$	$n \mid (x) \Rightarrow e_1 \mid x \mid e_1(e_2)$
variables	x		

Semantics

Small-step semantics:

$$\boxed{e \longrightarrow e'}$$

DoCall

$$\frac{}{((x) \Rightarrow e_1)(v_2) \longrightarrow [v_2/x]e_1}$$

SearchCall1

$$\frac{e_1 \longrightarrow e'_1}{e_1(e_2) \longrightarrow e'_1(e_2)}$$

SearchCall2

$$\frac{e_2 \longrightarrow e'_2}{v_1(e_2) \longrightarrow v_1(e'_2)}$$

Why do we need the substitute?

Implementation

e_{illtyped} : 3(4)

How does our semantics handle the above?

Dynamic Type Errors - Syntax and Semantics

Recall, we fixed the above issue by implementing dynamic type errors

step-results $r ::= \text{typeerror } e \mid e'$

$e \longrightarrow r$

TypeErrorCall

$$\frac{v_1 \neq x^?(y) \Rightarrow e_1}{v_1(e_2) \longrightarrow \text{typeerror}(v_1(e_2))}$$

PropagateCall1

$$\frac{e_1 \longrightarrow \text{typeerror } e}{e_1(e_2) \longrightarrow \text{typeerror } e}$$

PropagateCall2

$$\frac{e_2 \longrightarrow \text{typeerror } e}{v_1(e_2) \longrightarrow \text{typeerror } e}$$

Implementation

Static Typing

- Some expressions are “well-typed” and some expressions are “ill-typed”

$e_1: ((i) \Rightarrow i) (4)$

$e_2: 3(4)$

- Which one is ill-typed? Why?
- What is a type?
- A *type system* is a language of types and typing judgments that define when an expression has a particular type

$e : \tau$

TypeScripty

Let's modify our language to allow us to type check before interpretation, and implement typing rules that can infer the types of expressions

- What are the types of this language?

TypeScripty Syntax

types $\tau, t ::= \text{number} \mid (x : \tau) \Rightarrow \tau'$
values $v ::= n \mid (x : \tau) \Rightarrow e_1$
expressions $e ::= n \mid (x : \tau) \Rightarrow e_1 \mid x \mid e_1(e_2)$

- This is very similar to the abstract syntax of JavaScripty
- All possible types of values have their own types

TypeScripty Small-Step Operational Semantics

$$\boxed{e \longrightarrow e'}$$

DoCall

$$((x : \tau) \Rightarrow e_1)(v_2) \longrightarrow [v_2/x]e_1$$

SearchCall1

$$\frac{e_1 \longrightarrow e'_1}{e_1(e_2) \longrightarrow e'_1(e_2)}$$

SearchCall2

$$\frac{e_2 \longrightarrow e'_2}{v_1(e_2) \longrightarrow v_1(e'_2)}$$

- Remains largely unchanged
- We use the simpler version (without dynamic typing or coercions) because we will be type checking before interpretation

Typing Judgment

Recall our judgment form:

$$e : \tau$$

Let's inductively define this judgment form with a set of typing rules

- What is our rule for a number?
- What about a variable?

Typing rules - Type Environment

First, we need a type environment

type environments $\Gamma, \text{tenv} ::= \cdot \mid \Gamma, x : \tau$

Type environment Γ is either empty, or an existing environment with the additional mapping from variable x to type τ .

We now modify our judgment form:

$$\Gamma \vdash e : \tau$$

Typing rules

$$\boxed{\Gamma \vdash e : \tau}$$

TypeNumber

$$n : \text{number}$$

TypeFunction

$$\Gamma, x : \tau \vdash e_1 : \tau'$$

$$\Gamma \vdash (x : \tau) \Rightarrow e_1 : (y : \tau) \Rightarrow \tau'$$

TypeVar

$$\Gamma \vdash x : \Gamma(x)$$

TypeCall

$$\frac{\Gamma \vdash e_1 : (x : \tau) \Rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(e_2) : \tau'}$$

- What is TypeFunction stating?
- We will implement these rules as a static type checker which will run before our interpreter

Implementation

$e_{\text{welltyped}}$: $((i : \text{number}) \Rightarrow i) (4)$

e_{illtyped} : $3(4)$

Soundness

We would like to show that our type system guarantees the following property:

- If an expression e is well-typed, then it can never get stuck in a step - assuming a correctly-implemented interpreter.

This is *soundness*, if we claim that e is well-typed, then it will not exhibit runtime errors related to typing. In other words, we are not making incorrect assumptions.

If our type system is sound if and only if the progress and preservation properties can be shown.

Progress

If $e : \tau$, then $e \longrightarrow e'$ for some expression e' .

Preservation

If $e \longrightarrow e'$ and $e : \tau$, then $e' : \tau$.

Showing These Properties

We want to show that these properties hold for test expressions.

Let's define the following judgment form:

$$e \hookrightarrow_{\tau} v$$

Read as: Expression e reduces to a value v using some number of steps while checking the preservation of type τ at each step.

Showing These Properties cont.

We can now inductively define this judgment form with the following inference rules:

$$\boxed{e \hookrightarrow_{\tau} v}$$

Reduces Value

$$\frac{e \text{ value}}{e \hookrightarrow_{\tau} e}$$

Reduces Progress And Preservation

$$\frac{e \longrightarrow e' \quad \cdot \vdash e' : \tau \quad e' \hookrightarrow_{\tau} e''}{e \hookrightarrow_{\tau} e''}$$

- What is ReducesProgressAndPreservation stating?
- What aspects of the rule cover the three properties?
- Why do we need three versions of e ?

Implementation

Let's Add Binary Plus

How does our abstract syntax change?

Expanded TypeScript Syntax

types $\tau, t ::= \text{number} \mid (x : \tau) \Rightarrow \tau'$

values $v ::= n \mid (x : \tau) \Rightarrow e_1$

expressions $e ::= n \mid (x : \tau) \Rightarrow e_1 \mid x \mid e_1(e_2) \mid e_1 + e_2$

What about our typing rules?

BinaryPlus Typing Rule

TypeBinaryPlus

$$\frac{\Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number}}{\Gamma \vdash e_1 + e_2 : \mathbf{number}}$$

We must also update our small-step operational semantics

BinaryPlus Small-Step Semantics

$$\boxed{e \longrightarrow e'}$$

DoBinPlus

$$\frac{n' = n_1 + n_2}{n_1 + n_2 \longrightarrow n'}$$

SearchBinary1

$$\frac{e_1 \longrightarrow e'_1}{e_1 + e_2 \longrightarrow e'_1 + e_2}$$

SearchBinary2

$$\frac{e_2 \longrightarrow e'_2}{v_1 + e_2 \longrightarrow v_1 + e'_2}$$

Implementation