WELLESLEY

Programming Languages

CS 251

Fall 2021

Carolyn Anderson

Types

What do we do about errors?

In our big step semantics, we can describe situations where errors arise. But we won’t track errors, since that requires representing the program context (hard 😢).

When we hit an error, we just abandon the derivation.

What if we could catch errors early?

This program contains an error:

```
( racket … )
( more racket … )
( lots of racket … )
…

(+ 1 #t)
```

But we won’t know about it until we get here!

What if we could catch the error before the program runs?

Types

Idea: give each expression in our language a type label.

(racket …)
(more racket …)
(lots of racket …)
…
(+ 1 #t)
(+:number → number → number **1**:number **#t**:bool**)**

clash!

Types

A type is…

- a set of values that **share some property**
- a **promise to produce** a member of a certain set of values
- a **prediction about the value** an expression will yield

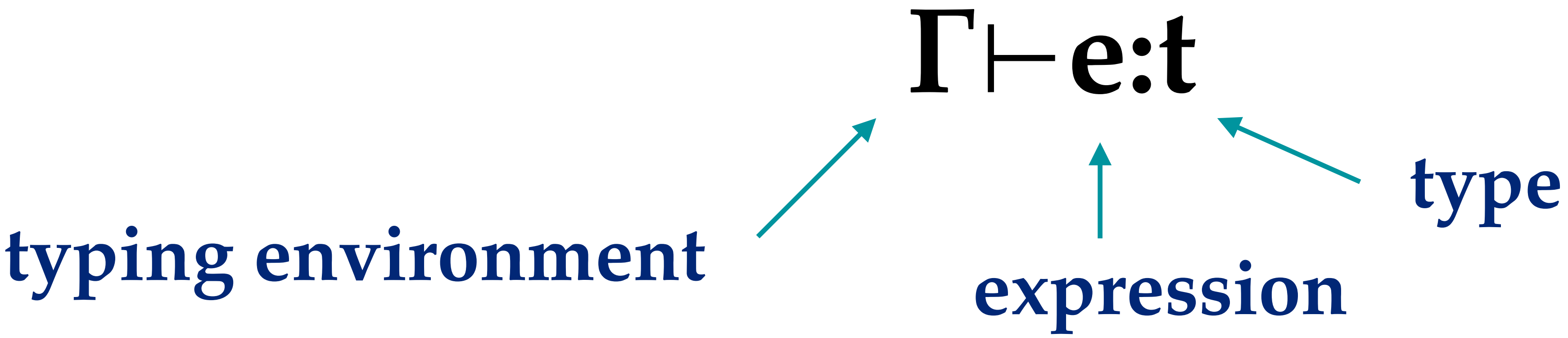
Why types?

- Help catch certain kinds of errors
- Help with documentation
- Compilers can exploit them to make code go faster

Type Judgements

A type judgment is a rule for determining the type of an expression.

We use the following notation for a type judgment:



$$\boldsymbol{\Gamma \vdash e : t}$$

typing environment

expression

type

This rule is pronounced "**Γ** proves that **e** has type **t**"
or "**e** types to **t** in environment **Γ**"

Typing numbers, Booleans, and addition

$\Gamma \vdash e : \text{number}$　　　　$\Gamma \vdash e : \text{bool}$

$$\frac{\Gamma \vdash e_1 : \text{number} \qquad \Gamma \vdash e_2 : \text{number}}{\Gamma \vdash (+\ e_1\ e_2) : \text{number}}$$

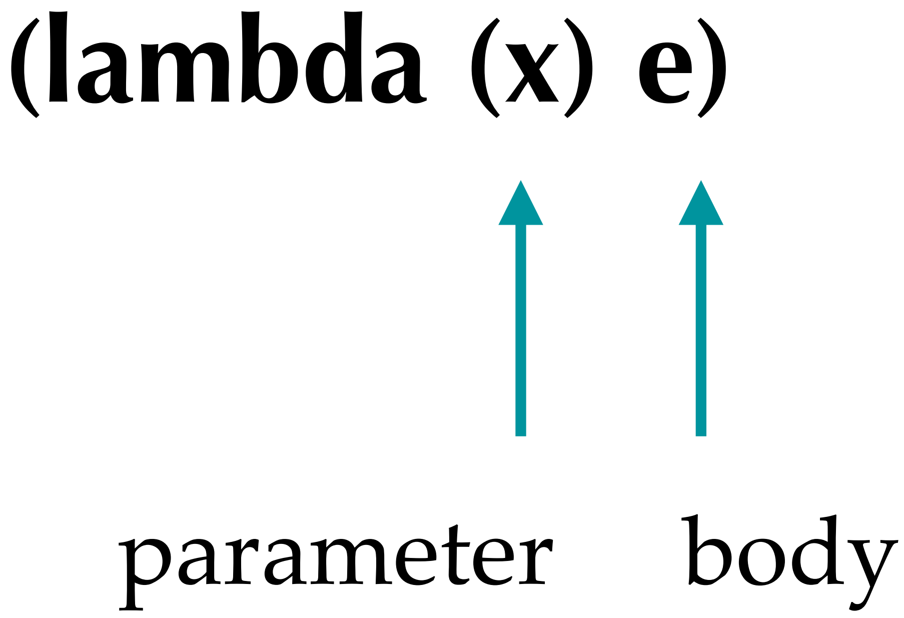
Credit: presentation of type-checking follows Krishnamurthi (2007)

Typing functions

Functions are a little harder.

We'll consider only anonymous functions, since we don't have **define** or any local binding constructs.

Also, we'll only consider 1-parameter functions.



(lambda (x) e)

parameter   body

Typing functions

The body is the easy part. We can just recursively type-check it.

(lambda (x) e)

type variable

$$\frac{\Gamma \vdash x : \mathbf{??} \qquad\qquad \Gamma \vdash e : \mathbf{tau}}{\Gamma \vdash (\text{lambda } (x)\ e) : \mathbf{??} \longrightarrow \mathbf{tau}}$$

functions have arrow types

Typing functions

But what can we do about the parameter?

We don't know anything about it!

(lambda (x) e)

$$\frac{\Gamma \vdash x : ?? \qquad\qquad \Gamma \vdash e : \mathbf{tau}}{\Gamma \vdash (\text{lambda } (x)\ e) : ?? \longrightarrow \mathbf{tau}}$$

Typing functions

Solution: we have to assume that **the program comes with type annotations on all function parameters**.

(lambda (x:tau$_1$) e)

$$\frac{\Gamma(x \vdash \mathbf{tau1}) \qquad \Gamma \vdash e\mathbf{:tau2}}{\Gamma \vdash (\mathrm{lambda}\ (x{:}\mathrm{tau1})\ e)\mathbf{:tau1 \longrightarrow tau2}}$$

Typing variables

Our last kind of **value** is variables.

Typing variables is easy. We will assume that the environment records their type.

$$\frac{\mathbf{\Gamma(e) = tau}}{\Gamma \vdash e{:}\mathbf{tau}}$$

Typing function application

What about function application?

We have two expressions: the function and its argument.

((lambda (x : number) (+ x 5)) 10)

Typing function application

What about function application?

We have two expressions: the function and its argument.

((lambda (x : number) (+ x 5)) 10)

We can recursively type check those:

$$\frac{\Gamma \vdash e_1 : \mathbf{tau1} \longrightarrow \mathbf{tau2} \qquad \Gamma \vdash e_2 : \mathbf{tau3}}{\Gamma \vdash (e_1\ e_2) : ???}$$

Typing function application

We also need to make sure that the function and its argument are type-compatible!

$$\frac{\Gamma \vdash e_1 : \mathbf{tau1} \longrightarrow \mathbf{tau2} \qquad \Gamma \vdash e_2 : \mathbf{tau3}}{\Gamma \vdash (e_1\ e_2) : \mathbf{tau2}\ \textit{if tau1 = tau3}}$$

Practice:

Work on type judgment rules for number comparison:

1) =
2) equal?

Evaluating our type judgment rules

How do we evaluate our type-checking system?

Evaluating our type judgment rules

How do we evaluate our type-checking system?

Idea #1: Try out some programs

Let’s check some programs!

```
((lambda (x : number) (+ x 5)) 10)

((lambda (x : number) (+ x 5)) #t)

((lambda (x : number) (+ #f 5)) 10)
```