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

What types?

In theory we could make every value its own type.

But this is equivalent to not having any types: it doesn't help us reason about errors like (+ 1 #t): + would have to accept an infinite number of individual number types.

In practice, type systems often reflect categories of values that share implementation-independent properties.

Categorizing values by their properties

Integers:

- Can be added and subtracted
- Can be multiplied and divided
- Produce an integer when 1 is added to them

Lists:

- Can be concatenated
- Can have items appended to them
- Have a first item

These properties are implementation independent.

Static type checking

In **static type checking** , a program is checked for type errors before it is run.

This catches errors as early as possible.

Type Judgements

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

We use the following notation for a type judgment:

$$\underbrace{\Gamma}_{\text{typing environment}} \vdash \underbrace{\mathbf{e}}_{\text{expression}} : \underbrace{\mathbf{t}}_{\text{type}}$$

This rule is pronounced "Γ proves that **e** has type **t**"

or "**e** types to **t** in environment Γ"

A subset of Racket

Values

- numbers
- Booleans
- functions

Operators

- +
- -
- and
- or
- function application
- if

Note: we're omitting **let, letrec,** and **define**

Typing numbers and Booleans

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

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

Typing addition

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

Exercise: typing **and**

Write the type judgment for **and** statements.

Exercise: typing **and**

Write the type judgment for **and** statements.

$$\frac{\Gamma \vdash e_1 : \text{boolean} \qquad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash (\text{and } e_1\ e_2) : \text{bool}}$$

Exercise: typing **and**

Write the type judgment for **and** statements.

$$\frac{\Gamma \vdash e_1 : \text{boolean} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash (\text{and } e_1\ e_2) : \text{bool}}$$

Is this accurate for Racket?

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.

$$\textbf{(lambda (}\underset{\substack{\uparrow\\ \text{parameter}}}{\textbf{x}}\textbf{) }\underset{\substack{\uparrow\\ \text{body}}}{\textbf{e}}\textbf{)}$$

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)

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

Typing functions

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

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

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

Typing variables

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

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

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

Typing function application

What about function application?

We have two expressions: the function and its argument.

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

Typing function application

What about function application?

We have two expressions: the function and its argument.

((lambda (x) (+ x 5)) 10)

We can recursively type check those:

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

Typing function application

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

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