Types

Michael Wollowski

1

Introduction

- Consider:

```
(define add-tax
  (lambda (price)
    (+ price "8%"))
```

- We can define this function.
- However, when called, it will crash.
- The use of types can catch this class of error before the program ever runs.

2

Benefits of Types

- **Safety net.**
  - Types prevent certain classes of runtime errors.
  - A well-typed program will never apply an integer as a function, never add a string to a number.
- **Documentation.**
  - The signature (: add-tax (-> Number Number)) provides documentation.
  - This documentation is automatically verified by the compiler or interpreter.
- **Optimizations.**
  - If the compiler knows x is always an integer, it can use unboxed machine arithmetic instead of dispatching through a generic + that must check tags at runtime.

3

Typing

Untyped: Assembly code

Dynamically Typed: Racket, Python, Javascript.
Types exist but are checked at runtime

Statically Typed: Java, Rust, Go.
Types checked before execution

4

Types as Sets of Values

A type is a name for a set of values:

- **`Num`** = { ..., -2, -1, 0, 1, 2, ... } (integers, or more broadly, numbers)
- **`Bool`** = { true, false }
- **`Num → Num`** = the set of all functions from numbers to numbers

- Saying "expression **`e`** has type **`τ`**" means:
- "When $\in$ is evaluated, the resulting value is a member of the set named **`τ`**."

5

Typing Environments

- When we encounter a variable **`x`**, we need to know its type.
- A *typing environment* (often written **`Γ`**, the Greek letter Gamma) is a mapping from variable names to types:

```
Γ = { x : Num, y : Bool, f : Num → Num }
```

- Think of it as the type-level equivalent of a runtime environment that maps variables to values.

6

Type Datatypes

```
(define-datatype type type?
  (tnum)
  (tbool)
  (tfun
   (arg type?)
   (ret type?)))
```

7

Type Judgements

- A *type judgment* has the form:

  Γ ⊢ e : τ

- We say: "In environment **Γ**, expression **e** has type **τ**."
- Examples:

```
{}                ⊢ 5           : Num
{}                ⊢ true        : Bool
{x : Num}         ⊢ (+ x 1)     : Num
{f : Num → Num}   ⊢ (f 3)        : Num
```

8

Type Inference Rules

- We express the typing relationship through inference rules.
- An inference rule has the form:

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \ldots}{\text{conclusion}}$$

- If all the premises hold, the conclusion holds.

9

Type Checker vs Type Inference

- If types of functions and variables are declared by the programmer, then we end up implementing a *type checker*.
- This is when we check static types.
- This is what happens in Java.
- *Type inference* happens in dynamically typed languages.
- The systems determines types without type declarations.
- This is what happens in Racket.

10

Type Rules for a Subset of Racket

```
e ::= n                          (numeric literal)
    | b                          (boolean literal: true / false)
    | x                          (variable)
    | (+ e₁ e₂)                  (addition)
    | (- e₁ e₂)                  (subtraction)
    | (if e₁ e₂ e₃)              (conditional)
    | (fun (x : τ) : τ_ret e)    (annotated function)
    | (e₁ e₂)                    (application)
    | (let ((x e₁)) e₂)          (local binding)
```

11

Type Rules

- **Numeric Literals**

$$\frac{}{\Gamma \vdash n : \text{Num}} \quad [\text{T-Num}]$$

- No premises needed.
- Any numeric literal always has type **`Num`**.
- **Boolean Literals**

$$\frac{}{\Gamma \vdash n : \text{Bool}} \quad [\text{T-Bool}]$$

- No premises needed.
- Any Boolean literal always has type **`Bool`**.

12

Type Rules

- **Variables**

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{[T-Var]}$$

- Look up the variable in the type environment.
- If it's not there, that is a type error (unbound variable).

13

Type Rules

- **Addition (and similarly, Subtraction)**

$$\frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash (+\ e_1\ e_2) : \text{Num}} \quad \text{[T-Add]}$$

- Both operands must be numbers; the result is a number.
- **Conditionals**

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if}\ e_1\ e_2\ e_3) : \tau} \quad \text{[T-If]}$$

- The condition must be a Boolean.
- The two branches must have the *same* type, and that type is the type of the whole if expression

14

Type Rules

- **Functions**

$$\frac{\Gamma,\ x : \tau\_param \ \vdash\ body : \tau\_ret}{\Gamma \vdash (\text{fun } (x)\ body) : \tau\_param \to \tau\_ret} \quad [\text{T-Fun}]$$

- The programmer has declared the parameter type `τ_param` and the return type `τ_ret`.
- We extend the environment with `x : τ_param`, check that the body actually has a type consistent with `τ_ret`, and produce the type as indicated.

15

Type Rules

- **Application**

$$\frac{\Gamma \vdash e_1 : \tau\_arg \to \tau\_ret \qquad \Gamma \vdash e_2 : \tau\_arg}{\Gamma \vdash (e_1\ e_2) : \tau\_ret} \quad [\text{T-App}]$$

- The function's argument type must match the actual argument's type.
- The result type is the function's return type.

16

Type Rules

- **Let**

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } ((x\ e_1))\ e_2) : \tau_2} \text{ [T-Let]}$$

- Type-check the bound expression, then extend the environment and type-check the body.
- Note that let does *not* need annotations — the type of x is whatever the type of e_1 is.

17