Comp 411
Principles of Programming Languages
Lecture 20

What is a Type?

Corky Cartwright

March 22, 2021

What is a Type?

Canonical example: consider the Jam expression of the form

```
if (big-ugly-expression)
   (5)(6)
else 17
```

which may be embedded deep inside a program. What type should a language translator (<compiler/interpreter) assign to this expression?

How will this expression behave? If *`big-ugly-expression>`* is false, then the expression will produce a `int` result. In this case, it is plausible for the type-checker to return `int` as the type of `17`. But what if *`big-ugly-expression`* is true? Then the expression will generate a run-time error. Even worse, it is a statically detectable run-time error.

Type-checkers generally assume all code fragments are meaningful (reachable in execution). Otherwise, why is the fragment included as part of the program? Hence, all static type checkers will reject this expression – even if *`big-ugly-expression`* is obviously false (*e.g.*, *`big-ugly-expression`* is the constant `false`).

Intuitive Assumptions in Type Checking

Idea 1: Types are names for sets of values.

Idea 2: The valid sets of ``input values'' for each program operation can be described in terms of types (most of the time).

In most cases, the second idea can be made completely true by incorporating it as part of the contract for the operation either by extending the domain of the operation or explicitly reporting when it throws exceptions. Example: `zip` in a functional language. In this case, we typically fudge the definition so it is well-defined on unequal length lists.

Idea 3: The application of program operations and the returning of values as the results of defined operations (methods, functions, procedures) induces constraints on program types.

The mathematical constraints are subtyping (subset) relationships:
(*i*) the type of an operation argument must be a subtype of its declared input type;

(*ii*) the type of the result returned by an operation must be a subtype of its declared result type.

In practice, most type systems force the type equality instead of type containment. It greatly simplifies the structure of the type systems.

Typed λ-Languages

The (simply) typed λ-calculus is the foundation of structural typing which is the overwhelmingly dominant typing discipline in functional (but not OO) languages.

Syntax:

```
M ::= λ V:τ . M | (M M) | V | C
τ ::= D₁ | … | Dₙ | τ → τ
```

where **C** is an optional set of constants (empty in the pure simply typed λ-calculus); D_1, …, D_n are *disjoint* domains of primitive values (only one mythical domain **D** for the pure simply typed λ-calculus) called *primitive types* and **V** is the set of variable symbols. We will almost always work in extensions of the pure λ-calculus like Jam that include constant values and operations and primitive types. We will sometimes omit the typing for a variable introduced in a λ-expression.

Typing Rules for the (Simply) Typed λ-Calculus

A *typing judgment* has form: $\Gamma \vdash M{:}\tau$

where Γ (the Greek letter "capital gamma") is called the *type environment* which maps a finite set of type identifiers to types. Each pair in the function [viewed as a relation] is written $x{:}\tau$ where x is either a variable or a constant and τ is a type.

The inference rules for the pure (simply) typed λ-calculus are:

$$\Gamma,\{x{:}\tau\} \vdash x{:}\tau \qquad \text{(binding axiom)}$$

$$\frac{\Gamma,\{x{:}\sigma\} \vdash M{:}\tau;\ \ x \text{ not free in } \Gamma}{\Gamma \vdash \lambda x.M{:}\sigma\to\tau} \qquad \text{(abstraction rule)}$$

$$\frac{\Gamma \vdash M{:}\sigma\to\tau;\ \ \Gamma \vdash N{:}\sigma}{\Gamma \vdash (M\ N){:}\tau} \qquad \text{(application rule)}$$

Typing Rules for Typed λ-Languages

- A *typing* for an expression **M** given the type environment **Γ** is an inference (proof) tree for a typing judgment of the form $\Gamma \vdash M{:}\tau$.
- Top level programs are typed with respect to a *base* type environment Γ_0 that contains the types of all program constants (including functions). For the *simply* typed λ-calculus, the base type environment is *empty*, because *there are no constants* in the pure simply typed λ-calculus.
- Typed λ-languages require exact matching between the input type of a function and the type of arguments to which it is applied. Why? *There is no subtyping*. Every value belongs to a unique type.
- Every constant **c** in **C** has a corresponding type in the base type environment Γ_0. The arity of each constant **c** must match its type.
- Recall that the pure simply typed calculus *has no constants*.