Comp 411
Principles of Programming Languages
Lecture 20
What is a Type?

Corky Cartwright

March 14, 2014

What is a Type?

Canonical example: consider the expression of the form

```
(if big-ugly-expression
  (5 6)
  a-nice-value)
```

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 legal result. In this case, it is plausible for the type-checker to return the type of this value as the type of **`a-`*****`nice-value`***. 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 should assume all code fragments are meaningful (reachable in execution). Otherwise, why is the fragment included as part of the program? Hence, all 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).

Perhaps the second idea can be made completely true by imposing it as part of the contract for any operation. Example: `zip` in a functional language. In this case, contract should include check for equal lengths.

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 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 Lambda Calculus

The (simply) typed lambda calculus is the foundation of structural typing which is the overwhelmingly dominant typing discipline in functional languages.

Syntax:

```
M ::= λ V:τ . M | (M M) | V
τ ::= D | τ → τ
```

where `D` is an unspecified "ground" (non-functional) type like `int`. and `V` is the set of variable symbols.

Typing Rules for Typed Lambda Calculus

Typing Judgment has form: $\Gamma \mid M : \tau$

where Γ is a set of typings of the form $x : \tau$ where x is either a variable or a constant.

$$\frac{\Gamma,\ x : \sigma \mid M : \tau}{\Gamma \mid \lambda x : \sigma .\ M : \sigma \to \tau} \qquad \text{(abstraction rule)}$$

$$\frac{\Gamma \mid f : \sigma \to \tau; \quad \Gamma \mid M : \sigma}{\Gamma \mid (f\ M) : \tau} \qquad \text{(application rule)}$$

Typing Rules for Typed Lambda Calculus

- Top level programs are typed with respect to a *base* type environment that contains the typings of all program constants (including functions). For the *simply* typed lamba calculus, the base type environment is *empty*, because *there are no constants*.
- In some formulations of structural typing rules, the typings of constants are placed in a separate constant type environment that is always implicitly available in addition to the explicit type environment Γ appearing in type judgments.
- The typed lambda-calculus requires 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.