Programming Paradigms

Type Systems (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Overview

- **Introduction**
- **Types in Programming Languages**
- **Type Equivalence**
- **Type Compatibility and Conversions**
- **Formally Defined Type Systems**
  - Arithmetic Expressions
  - Lambda Calculus

Quiz: JavaScript Expressions

What values do these JavaScript expressions evaluate to?

```
"" == 0
1 == true
"true" == 1
false == "false"
```

Quiz: JavaScript Expressions

What values do these JavaScript expressions evaluate to?

```
"" == 0                 // true
1 == true               // true
"true" == 1             // false
false == "false"        // false
```

Quiz: JavaScript Expressions

What values do these JavaScript expressions evaluate to?

```
"" == 0                // true
1 == true              // true
"true" == 1            // false
false == "false"       // false
```

Number and string: String is coerced into a number (here: 0 and NaN)

Quiz: JavaScript Expressions

What values do these JavaScript expressions evaluate to?

```
"" == 0              // true
1 == true            // true
"true" == 1          // false
false == "false"     // false
```

Number and boolean: Boolean gets converted to number (here: 1)

Quiz: JavaScript Expressions

What values do these JavaScript expressions evaluate to?

```
"" == 0              // true
1 == true            // true
"true" == 1          // false
false == "false"     // false
```

Boolean and another type:

- **Boolean gets coerced to a number (here: 0)**
- **String also get coerced to a number (here: NaN)**
- **The two numbers differ**

Types

- **Most PLs: Expressions and memory objects have types**
- **Examples**
  - Assignment `x=4` (implicitly) says `x` has a number type
  - Declaration `int n;` says `n` has integer type
  - Expression `a+b` has a type, which depends on the type of `a` and `b`
  - `new X()` has a type

Why Do We Need Types?

Reason 1: Provide context for operations

- Meaning of `a+b` depends on types of `a` and `b`
  - E.g., addition vs. string concatenation
- Meaning of `new x` depends in the type of `x`
  - E.g., which initialization code to call?

PL implementation uses this context information

Why Do We Need Types?

Reason 2: Limit valid operations

- Many syntactically valid operations don’t make any sense
  - Adding a character and a record
  - Computing the logarithm of a set

Helps developers find bugs early

Why Do We Need Types?

Reason 3: Code readability and understandability

- Types = stylized documentation
- Makes maintaining and extending code easier

But: Sometimes, types make code harder to write

Why Do We Need Types?

Reason 4: Compile-time optimizations

- Compiler knows that some behavior is impossible
  - E.g., assignment of type T1 may not influence values of type T2

Works both for explicitly specified and implicitly inferred types

Bits Are Untyped

- **(Most) hardware stores and computes on raw bits**
  - Bits may be code, integer data, addresses, etc.
- **(Most) assembly languages are untyped**
  - Operation of any kind can be applied to values at arbitrary locations

Type Systems

- **Definition of types and their association with PL constructs**
  - Every PL construct that has/refers to a value has a type (e.g., named constants, variables, record fields, functions)
- **Rules for**
  - Type equivalence
  - Type compatibility
  - Type inference

Type Checking

Ensure that program obeys the type compatibility rules

Example (Java):

```
int a = 3;
String b = a - 2;
```

Type Checking

Ensure that program obeys the type compatibility rules

Example (Java):

```
int a = 3;
String b = a - 2;
```

Type error: Can’t assign int value to String variable

Overview

- **Introduction**
- **Types in Programming Languages** ←
- **Type Equivalence**
- **Type Compatibility and Conversions**
- **Formally Defined Type Systems**
  - Arithmetic Expressions
  - Lambda Calculus

Strongly Typed PLs

PL implementation enforces:

Operations only on values of proper type

- Most PLs since 1970s
- C is mostly strongly typed
  - Exceptions, e.g.,:
    - Subroutines with variable number of parameters
    - Interoperability of pointers and arrays

Statically Typed PLs

Strongly typed and checked at compile-time

- Strictly speaking, practically no PL is statically typed
  - E.g., Java: Upcasts and reflection allow for runtime type errors
- In practice, means ”mostly statically typed”

Dynamically Typed PLs

Type checking is delayed until runtime

- Type errors found only later in development process
- Common in “scripting languages”, e.g., JavaScript and Python
- Note: Every value has a type and type errors manifest as runtime errors

Gradual Typing

Middleground between statically and dynamically typed PLs

- Annotating types is optional
    - Can quickly write code and add types later
- Static type checker warns about errors obvious from the available types
    - No guarantee to find all type errors

Quiz: * Typed PLs

Which of the following statements is true?

- In a dynamically typed language, no type errors can occur.
- In practical, statically typed languages, all type errors are caught before running the code.
- Gradual typing let's programmers choose which types to annotate.
- Strong typing exists only in statically typed PLs.

Quiz: * Typed PLs

Which of the following statements is true?

- ~~In a dynamically typed language, no type errors can occur~~.
- ~~In practical, statically typed languages, all type errors are caught before running the code.~~
- Gradual typing let’s programmers choose which types to annotate.
- ~~Strong typing exists only in statically typed PLs.~~

Polymorphism

- **Greek origin: "Having multiple forms"**
- **Two kinds**
  - Parametric polymorphism: Code takes (set of) type(s) as parameter
    - E.g., generics in Java, containers in C++
  - Subtype polymorphism: Extending of refining a supertype
    - E.g., subclasses in Java or C++

Polymorphic Variables

In some PLs, a single variable may refer to objects of completely different types

Example (pseudo language):

```
a = "abc"
b = 42
a = b
a = "def"
```

Polymorphic Variables

In some PLs, a single variable may refer to objects of completely different types

Example (pseudo language):

```
a = "abc"   // a holds a string
b = 42      // b holds an int
a = b       // a holds an int
a = "def"   // a holds a string (again)
```

Polymorphic Variables

In some PLs, a single variable may refer to objects of completely different types

Example (pseudo language):

```
a = "abc"   // a holds a string
b = 42      // b holds an int
a = b       // a holds an int
a = "def"   // a holds a string (again)
```

Type-correct in most dynamically typed (and even some statically typed) PLs

Special Types and Values

- **`void` type**: **Indicates the absence of a type and has only one (trivial) value**
- **`null` value**: **Means "does not hold a value of its type"**
- **Option types**: **Indicates that the value may or may not hold a value of a specific type**
  - ☐ E.g., `Option[int]` in Python means `int` or `None`

Overview

- **Introduction**
- **Types in Programming Languages**
- **Type Equivalence** ←
- **Type Compatibility and Conversions**
- **Formally Defined Type Systems**
  - Arithmetic Expressions
  - Lambda Calculus

Type Equivalence

Prerequisite for type checking:
Clarify whether two types are equivalent

Two approaches

- Structural equivalence
  - Same structure means same type
  - Compares structure recursively
- Name equivalence (aka nominal equivalence)
  - Same type name means same type

Example: OCaml

Objects: Structurally typed by the **names** and **types** of their **methods**

Example: OCaml

Objects: Structurally typed by the **names** and **types** of their **methods**

```
let x =
  object
    val mutable x = 5
    method get_x = x
    method set_x y = x <- y
  end;;
```

Creates an object with one field and two methods

Example: OCaml

Objects: Structurally typed by the **names** and **types** of their **methods**

```
let x =
  object
    val mutable x = 5
    method get_x = x
    method set_x y = x <- y
  end;;
```

Creates an object with one field and two methods

```
let y =
  object
    method get_x = 2
    method set_x y =
      Printf.printf "%d\n" y
  end;;
```

Creates an object with two methods

Example: OCaml

Objects: Structurally typed by the **names** and **types** of their **methods**

```
let x =
  object
    val mutable x = 5
    method get_x = x
    method set_x y = x <- y
  end;;
```

```
let y =
  object
    method get_x = 2
    method set_x y =
      Printf.printf "%d\n" y
  end;;
```

`x = y;;` ← **Type-correct assignment**

Variation Across Languages

- **Do names matter?**
  - Same memory representation, but differently named
  - E.g., different field names in a record
- **Does order matter?**
  - Different memory representation, but lossless reordering possible
  - E.g., same fields but in different order

Limitation of Structural Equivalence

- Cannot distinguish different concepts that happen to be represented the same way
- Example (Pascal-like syntax):

```
type student = record
  name, address : string;
  age: integer
end;
```

vs.

```
type school = record
  name, address : string;
  age: integer
end;
```

Limitation of Structural Equivalence

- Cannot distinguish different concepts that happen to be represented the same way
- Example (Pascal-like syntax):

```
type student = record
  name, address : string;
  age: integer
end;
```

vs.

```
type school = record
  name, address : string;
  age: integer
end;
```

```
{ This is allowed: }
x : student; y : school;
x := y;
```

Name Equivalence

- Types with different names are different
- Assumption: Programmer wants it that way
- Used in many modern languages, e.g., Java

Limitations of Name Equivalence

- Alias types cause difficulties
- Example:

```
{ Here, we want both types to be the same }
type stack_element = integer;


{Here, we want distinct types,
  to prevent mixed computations}
type celsius = real;
type fahrenheit = real;
```

Quiz: Type Equivalence

```
type foo = record
  x : integer;
  y : integer
end;

type bar = record
  m : integer;
  n : integer
end;

a : foo;
b : bar;

a := b;
b := a;
```

Is this pseudo-code type-correct in a

(a) PL with structural type equivalence,

(b) PL with nominal type equivalence?

Quiz: Type Equivalence

```
type foo = record
  x : integer;
  y : integer
end;

type bar = record
  m : integer;
  n : integer
end;

a : foo;
b : bar;

a := b;
b := a;
```

Is this pseudo-code type-correct in a

(a) PL with structural type equivalence,

→ Yes

(b) PL with nominal type equivalence?

→ No