Programming Paradigms

Type Systems (Part 4)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Overview

- Introduction
- Types in Programming Languages
- Polymorphism
- Type Equivalence
- Type Compatibility
- Formally Defined Type Systems

Type Equivalence

Prerequisite for type checking:

Clarify whether two types are equivalent

Two approaches

- Structural equivalence
  - Same structure means same type
- Name equivalence
  - Same type name means same type

Structural Equivalence

- **Given two types, compare their structure recursively**
- **Example: Any class with**
  - an int field called “age”,
  - a boolean field called “isRegistered”, and
  - a method called “printRecord”

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

Examples (Pascal-like syntax)

T1 = record a: integer, b: real end;

T2 = record c: integer, d: real end;

T3 = record b: real, a: integer end;

T = record info: integer, next: ^T end;

U = record info: integer, next: ^V end;

V = record info: integer, next: ^U end;

Limitation of Structural Equivalence

- Cannot distinguish **different concepts** that happen to be **represented the same way**
- Example:

```
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:

```
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;
```

Strict vs. Loose Name Equivalence

- Aliases are **distinct types**
- `type A = B;` is a **definition**

- Aliases are **equivalent types**
- `type A = B;` is a **declaration**

Type Conversion

- **Explicit conversion (cast) of a value form one type to another**
- **Three cases**
  - Types are structurally equivalent: Conversion is only conceptual, no code generated
  - Types have different sets of values, but are represented in the same way in memory: May need check that value is in target type
  - Different low-level representations: Need special instructions for conversion

Examples (Ada)

```
n: integer
r: long_float
t: test_score                 -- integer range 0...100
c: celsius_temp               -- type alias for integer

t := test_score(n)            -- runtime, semantic check needed
n := integer(t)               -- no check needed
r := long_float(n)            -- runtime conversion
n := integer(r)               -- runtime conversion & check
n := integer(c)               -- purely conceptual
c := celsius_temp(n)          -- purely conceptual
```

.

.

Quiz: Types

Which of the following statements is true?

- Types are compatible if and only if they are equal
- Coercions mean that a programmer casts a value from one type to another type
- Type conversions are guaranteed to preserve the meaning of a value
- PLs with type inference may provide static type guarantees

Please vote in Ilias.

Quiz: Types

Which of the following statements is true?

- ~~Types are compatible if and only if they are equal~~
- ~~Coercions mean that a programmer casts a value from one type to another type~~
- ~~Type conversions are guaranteed to preserve the meaning of a value~~
- PLs with type inference may provide static type guarantees

Please vote in Ilias.