Type Systems & Checking

COMP 524: Programming Languages
Srinivas Krishnan
March 29, 2011

Based in part on slides and notes by Bjoern Brandenburg, S. Olivier and A. Block.
Purpose

Types provide implicit context

- Compilers can infer information, so programmers write less code.

- e.g., The expression \( a + b \) in Java may be adding two integer, two float or two strings depending on context.

Types define a set of semantically valid operations

- Language system can detect semantic mistakes

- e.g., Python’s list type supports append() and pop(), but complex numbers do not
Type Systems

A type system consists of:

1. A mechanism to define types and associate them with language constructs.

2. A set of rules for “type equivalence,” “type compatibility,” and “type inference.”
Type Systems: Type Checking

Enforcement of type system rules.

- **Type Checking** is the process of ensuring that a program **obeys the language’s type compatibility rules**.

Several approaches to type checking.

- Strongly typed: ADA, Java, Haskell, Python, …
- Weakly typed: C, C++, …
- Statically typed: Haskell, Miranda, …
- Dynamically typed: Python, Ruby, …
Strong vs. Weak Typing

**Strongly typed languages always detect type errors:**
- All expressions and objects must have a type
- All operations must be applied to operands of appropriate types.
- **High assurance**: any type error will be reported.

**Weakly typed languages may “misinterpret” bits.**
- “anything can go”
- Operations are carried out, possibly with unintended consequences.
- Example: adding two references might result in the sum of the object’s addresses (which is nonsensical).
Strong vs. Weak Typing

**Strongly typed languages always detect type errors:**

- All expressions and objects must have a type.
- All operations must be applied to operands of appropriate types.
- High assurance: any type error will be reported.

**Weakly typed languages may “misinterpret” bits.**

- “anything can go”
- Operations are carried out, possibly with unintended consequences.
- Example: adding two references might result in the sum of the object’s addresses (which is nonsensical).

Strong typing is **essential for secure execution** of untrusted code!

Otherwise, **system could be tricked** into accessing protected memory, etc.

Examples: Java applets, Javascript.
Static vs. Dynamic Type Checking

**Static Type Checking.**
- All checks performed at compile time.
- Each variable/expression has a fixed type.

**Dynamic Type Checking.**
- Only values have fixed type.
- Expressions may yield values of different types.
- All checks done necessarily at runtime.
Static vs. Dynamic Type Checking

**Static** Type Checking.

- All checks performed at **compile time**.
- Each **variable/ expression** has a fixed type.

**Dynamic** Type Checking.

This **terminology is not absolute**: most statically, strongly typed languages have a **(small) dynamic component**.

Example: **disjoint union types** in strongly typed languages require tag checks at runtime.
Type Checking

Type **Equivalence**
- When are the types of two values are the same?

Type **Compatibility**:
- Can a value of A be used when type B is expected?

Type **Inference**:
- What is the type of expressions if no explicit type information is provided?
- If type information is provided by the programmer, does it match the actual expression’s type?
Type Equivalence

When are two types semantically the same?

➡ For example, when combining results from separate compilation.

➡ Two general ideas:
  › **structural equivalence**
  › **name equivalence**

➡ In practice, many variants exist.
Structural Equivalence

Two types are structurally equivalent if they have equivalent components.

typedef struct{int a,b;} foo1;

typedef struct {  
  int a,b;  
} foo2;

Equivalent!
Structural Equivalence

Two types are structurally equivalent if they have equivalent components.

typedef struct{int a,b;} foo1;

does not mean that foo1 and foo2 are equivalent, in most languages. 

typedef struct{
    int b;
    int a;
} foo2;
typedef struct{
    char *name;
    char *addre;
    int age;
} student;

typedef struct{
    char *name;
    char *addre;
    int age;
} school;

... but probably not intentional.
Name Equivalence

Name equivalence assumes that two definitions with different names are not the same.

Programmer probably had a good reason to pick different names…

Solves the “student-school” problem.

Standard in most modern languages.
Type Aliases / Type Synonyms

Under name equivalence, it may be convenient to introduce alternative names.
E.g., for improved readability.

Such a construction is called an alias.

```type ItemCount = Integer```
Name Equivalence: Aliases

type ItemCount = Integer

Two ways to interpret an alias:

➡ Strict name equivalence
  ‣ ItemCount is different from Integer.
  ‣ This is called a derived type.

➡ Loose name equivalence
  ‣ ItemCount is equivalent to Integer.
Name Equivalence: Aliases

`
type ItemCount = Integer
`

Two ways to interpret an alias:

➡ **Strict name equivalence**
  
  ‣ `ItemCount` is different from `Integer`.
  
  ‣ This is called a **derived type**.

➡ **Loose name equivalence**

**Haskell**: uses loose name equivalence by default.

Strict name equivalence is available with the `newtype` keyword:

```
newtype ItemCount = Integer
```
Problem with Loose Equivalence

```plaintext
TYPE celsius_temp = REAL;
  farhen_temp = REAL;
VAR  c: celsius_temp;
  f: farhen_temp;
...
f:=c;(* probably should be an error*)
```
Type Conversion

Type mismatch.

- Intention: to use a value of one type in a context where another type is expected.
  - E.g., add integer to floating point
- Requires type conversion or type cast.

Bit representation.

- Different types may have different representations.
- Converting type cast: underlying bits are changed
- Non-converting type cast: bits remain unchanged.
  - But are interpreted differently.
  - Useful for systems programming.
Type Coercion: Implicit Casts

float x = 3;

When does casting occur?

➡ **Type coercion**: compiler has rules to **automatically cast values** in certain situations.
➡ E.g., integer-to-float promotion.
➡ Some languages allow coercion for user-defined types (e.g., C++).

Two-edged features.
➡ Makes code performing arithmetic more **natural**.
➡ Can hide type errors!
Type Coercion: Implicit Casts

\[ \text{float } x = 3; \]

When does casting occur?

- **Type coercion**: compiler has rules to automatically cast values in certain situations.
- E.g., integer-to-float promotion.

**Haskell**: no type coercion.

Any type conversion must be explicit.