COEN 175

Lecture 10: Type Systems

Type Systems

- The **type system** of a language is a set of rules that assign types to expressions.
- Sometimes the rules are obvious:
  - Adding two integers in C yields an integer.
  - Subtracting two integers in C yields an integer.
- Sometimes the rules are not so obvious:
  - Can you add a pointer and an integer in C?
  - Yes, the pointer is moved by the integer number of objects.
  - Can you add two pointers in C?
  - No, this is illegal (even in C).

Type System Specification

- We can specify a type system in a variety of ways.
- We can use English text.
  - No explanation needed, but often ambiguous.
- We can use type tables.
  - These are like truth tables, but for types.
  - Simple for single operators, but no way of composing them to determine types of larger expressions.
- We can use type expressions.
  - A formal way of describing a type system.
  - Can be composed algebraically.

Overloaded Operators

- An operator is said to be **overloaded** if it does **different operations** depending upon the number or types of its operands.
- As an example, & is overloaded in C:
  - In its unary form, it takes the address.
  - In its binary form, it performs bitwise-and.
- As another example, * is overloaded in Simple C:
  - In its unary form, it performs a dereference.
  - In its binary form, it performs multiplication.

Polymorphic Operators

- An operator is said to be **polymorphic** if it does the **same operation** regardless of the type or number of its operands.

- As an example, consider the address operator:
  - It operates on any type of operand, but performs the same operation regardless of the type.

- True polymorphic functions don't really exist in C.
  - Macros in C and especially templates in C++ come close, though they are examples of compile-time polymorphism.
  - Most functional languages have run-time polymorphism.

Type Conversions

- Languages allow you to convert an object of one type to an object of another type.
- An **implicit** type conversion is called a **coercion**.
  - Adding a `float` and an `int` in C results in the `int` being converted to a `float` without your interaction.
  - Some languages perform run-time checks on coercions.
- An **explicit** type conversion is called a **type cast**.
  - In C, the desired type is written in parentheses before the expression.
  - In other languages, an explicit function or method call (often a constructor) is required.

Type Equivalence

- What does it mean for two types to be equivalent?
  - Type equivalence is important as it is used for assignment, which includes passing parameters by value.
  - We can only assign "apples to apples."
- Under **name equivalence**, two types are equivalent if and only if they have the same name.
- Under **structural equivalence**, two types are equivalent if and only if they have the same structure.

Name Equivalence

- Which objects have equivalent types under name equivalence?

```
typedef int height;
struct foo { int x, y; } s1;
struct bar { int x, y; } s2;
height x;
int y;
```

- None of these objects have equivalent types.
  - We cannot even initialize `x` with a value such as 123.

Structural Equivalence

- Which objects have equivalent types under structural equivalence?

```
typedef int height;
struct foo { int x, y; } s1;
struct bar { int x, y; } s2;
height x;
int y;
```

- The variables `x` and `y` have equivalent types.
- The variables `s1` and `s2` have equivalent types.

Type Equivalence in Practice

- Pure name equivalence is too restrictive.
- Pure structural equivalence is too expensive and does not work on recursive types.
- Most languages use a compromise.
- C uses structural equivalence for everything but structures, for which it uses name equivalence.
  - In essence, any `typedef` is expanded.

Static vs. Dynamic Typing

- A language can use **static** or **dynamic** typing.
- Statically typed languages such as C perform type checking at compile time.
- Dynamically typed languages such as PhP and Python perform type checking at run time.
- C++ is still statically type-checked, but does some run-time type lookups for polymorphic functions.
- Static type checking lets us catch errors early.

Type Tables

- Type tables are like truth tables but types are used instead of Boolean values.
- Consider a language with types `integer` and `real`. What would the table for addition be?

+	integer	real
integer	integer	real
real	real	real

Handling Errors

- We can introduce an error type to make handling of errors explicit.
- Consider a language with types `integer` and `real` along with the error type. What would the table for addition be?

+	integer	real	error
integer	integer	real	error
real	real	real	error
error	error	error	error

Summary

- Type tables are simple and intuitive.
- However, they do not scale well if a language has a rich set of operators and types.
- More importantly, they do not let us manipulate types in an algebraic way.
- Type expressions are a compact specification that let us manipulate types algebraically.
- However, like regular expressions, type expressions take some getting used to.