Lecture Overview

Types

1. Type systems
2. How to think about types
3. The classification of types
4. Type equivalence
    - structural equivalence
    - name equivalence
5. Type compatibility
6. Type inference

[Scott, chapter 7]
[Sebesta, chapter 6]

Type Systems

Computers can interpret memory in many different ways, but they never keep track of which interpretations are used for which memory. Programming is orchestrating these interpretations.

High-level languages, on the other hand, are built to keep track of interpretations and ensure that each piece of memory gets only one interpretation (ideally). This is a tremendous help in avoiding errors of misinterpretation.

Types are associated with values in high-level languages. These types provide context and help with error-checking.

A **type system** has:

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 equivalence details which types are considered the same. Type compatibility states when a value of a certain type can be used in a given context. Type inference defines the type of an expression based on the types of its constituent parts.

In languages with polymorphism, there is also a distinction between the type of an expression (the static type) and the type of the object to which it refers (the dynamic type). For instance, in java, a variable of type **List** may dynamically refer to an instance of type **ArrayList** (which is a subclass of List).

Type checking is the process of ensuring that a program conforms to the language's type compatibility rules.

A language is **strongly typed** if it prohibits, in an enforceable way, any operation to any object that is not designed to support that operation. One hears comparatives with strongly typed: "Pascal is **more** strongly typed than C." and "Pascal is **almost** strongly typed."

A language is **statically typed** if all type checking can be performed at compile time.

A language is **dynamically typed** if the type checking is all performed at run time. Dynamically scoped languages tend to be dynamically typed.

How to think about types

Types can be thought of from 3 points of view:

1. **Denotational**. Here we consider a type as a set of values. Anything with one of the values in the set has the type.

2. **Constructive.** Here we consider a type either a **primitive type** (built-in to to language) or one created by applying **type constructors** to primitive types. A type constructor describes how to create a new type from given type(s)—for instance, “array of” is a type constructor in many languages, as is “record” (struct) or “class”.

3. **Abstraction-based**. Here, a type is an **interface** consisting of a consistent set of operations with consistent semantics (meaning).

The Classification of Types

- **Boolean.** Typically one byte with 0 representing false and 1 representing true.

- **Numeric.**
  - distinguished based on length?
  - allows signed and unsigned?
  - integer
  - floating
  - rational (Scheme, Common Lisp)
  - fixed-point (Ada)
  - complex

- **Character.** Sometimes considered a numeric type. Typically 1 or 2 bytes.

- **Enumeration.**
  - type-compatible with integer (C)
  - integer-based type incompatible with integer (Modula)
  - interface-based type consisting of singletons (java)

- **Subrange.** Based on a contiguous subset of a **base type**. The base type can be numeric, character, enumeration, or any type with a complete ordering.

```
type percent_vote = 0..100;
     weekday = Monday..Friday;
```

- **Strings.** Often a **reference type** containing a reference (pointer) to a heap-allocated record. String records can be as simple as a null-terminated list of characters, or they may have header information, e.g. the length of the string.

- **Composite types**. These are types constructed with a type constructor.

    - **Records.** A collection of **fields**, each of which has a simpler type. Equivalent of math **tuples.** Simple **objects** can be considered records with their own subroutines.
    - **Variant records.** Allows different types of data to occupy the same space, but only one at a time!
    - **Arrays.** The most common composite type.

- **Sets.** A set type is the powerset of its base type, which must usually be discrete.
- **Pointers.** Pointers are reference types: the value of a pointer is a reference to the pointed-at object.
- **Lists.** Like arrays, contain a sequence of elements, but without indexing. Typically recursively defined.
- **Files.** Like arrays, but often can only be accessed sequentially. Intended to represent data on mass storage devices.
- **Subclasses.** These are types that use a special type constructor to **inherit** the properties of its base class (in addition to any other properties it declares).

Type equivalence

There are two principal ways of defining type equivalence. They are **structural equivalence** and **name equivalence**.

Two types are structurally equivalent if they are composed of the same parts and constructed in the same manner. The following three types are all structurally equivalent.

```
struct {int a, b;}

struct {
    int a, b;
}

struct {
    int a;
    int b;
}
```

But consider the following type:

```
struct {
    int b;
    int a;
}
```

In ML, that type is equivalent to the earlier three. In most languages, it is not.

Structural equivalence is a very implementation-oriented concept, and it fails to distinguish between types that are coincidentally the same but conceptually different.

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

type school = record
        name, address: string
        age: integer

x: student;
y: school;

x := y;
```

Most programmers would probably want to be informed if they assigned a value of type school to a variable of type student.

The solution is **name equivalence.** If the programmer wrote two type definitions, they probably want them to represent different types. In name equivalence, each type definition defines a different type.

name equivalence

Some languages allow **type aliasing**, or giving another name to a type.

```
TYPE stack_element = INTEGER;
```

If the language considers these two types to be distinct, then it is said to have **strict name equivalence**. If they are considered equivalent, then the language is said to have **loose name equivalence**. Pascal-family languages use loose name equivalence. A problem with this is as follows:

```
TYPE celsius_temp = REAL;
       fahrenheit_temp = REAL;
VAR c: celsius_temp;
       f: fahrenheit_temp;
…
f := c;
```

That's legal in Pascal.
Ada allows both loose name equivalence and strict name equivalence, at the behest of the programmer. A subtype (keyword: subtype) is equivalent to the base type, and a derived type (keyword: type) is not.

```
subtype stack_element is integer;
type fahrenheit_temp is new integer;
```

The **BRANDED** keyword in Modula-3 has much the same effect as Ada's subtype.

Type compatibility

Most languages do not require equivalence of types in most contexts; they require compatibility instead. For instance, in

```
a := expression;
```

the type of the expression must be compatible with that of a. The types of the operands of + must both be compatible with integers, or both be compatible with the floating-point type.

The definition of type compatibility differs from language to language. Whenever a language allows nonequivalent types to be compatible, it must perform implicit type conversions (called **coercions**) behind the scenes.

Coercions are controversial because they allow types to be mixed without the explicit intent of the programmer. They weaken **type security** (the prevention of type errors). The languages C and C++ are notoriously coercive, and has a programmer-extensible set of coercions (as does Scala).

Fortran allows arrays to appear as operands for its arithmetic operators. C allows arrays and pointers to be intermixed.

Many languages have a **generic reference type** that can hold a reference to any object. In C and C++, this is **void ***; in Modula-3 it is **refany**; in C#, **object**. Since a variable of this type can hold any reference, no type-checking is needed when assigning to such a variable.

But if a generic reference type is assigned **to** a specific reference type, then either type safety suffers or a type check must be performed.

```
void *v = some_ptr;
double *d;
…
d = v;          // is v really a double* ?
```

This cannot, in general, be type-checked statically. The trend is to make dynamic type checks.

To do this, one must make the objects self-descriptive. That is, each object contains an indication of its type. Then the run-time system can check to see if this assignment is legal and issue an error (and generally halt) if it is not. This is now common in object-oriented languages.

Type inference

Type checking ensures that an operator's operands or a function's arguments have the correct types. But what determines the type of the overall expression?

Generally arithmetic operators give a result that is the same type as the operands, and functions give the return type that was declared with the function. However, operations on subranges and on composite objects do not necessarily preserve the type of the operand.

Suppose we have

```
type Atype = 0..20;
var   a, b: Atype;
```

then what is the type of the expression a+b ? Is it another Atype, or is it some new type with a range of 0 to 40? Both could be reasonable answers. If we also have variable c as an Atype, the assignment

```
c := a + b;
```

needs a dynamic check to ensure that a+b fits into an Atype.

Operations on some composite types offer similar challenges. In Ada, "cat" is a three-character array, and "alog" is a four-character array. The expression "cat" & "alog" is a seven-

character array (In Ada, as in Go, the size of an array is part of its type.)

Functional languages often have parametric polymorphic operators, where typechecking requires the inference of what the parameters are. For instance, the operator **map** may take a left operand of type List<T> for some type T, and a right operand of type Proc<T→S> for types T and S. (I.e. the right operand is a procedure that maps a T to an S.) The type T in the left and right arguments must match, and the result is of type List<S>. The **signature** of map is therefore:

(List<T>, Proc<T→S>) → List<S>

An example of map is:

[1.23, 3.5, -2.7] map floor

where the result would be the integer array

[1, 3, -3]

Here, the compiler would see that the left argument was List<floating>, and know that T must be floating. Then it would see the right argument is Proc<floating→int>, and verify that the T in this expression is also floating, and that S is an int. It would conclude that the result is of type List<S>, which is List<int>.

This is simple **type unification**, as the compiler has to unify (ensure the sameness of) the type T in both arguments. ML, Miranda, and Haskell have this type of sophisticated type inferencing.