Types & Type Systems

Programming Abstractions

Department of Computer Science, Stony Brook University

Dr. Ritwik Banerjee

Undefined programs

- Recall that using λ-calculus, we were able to construct programs like $(\lambda x.\, x)y$ that have no defined semantics, and as such, are irreducible.
- We have no way to restrict the functions either, and could end up with meaningless programs because of this. For example, consider

$$\lambda f.\, \lambda x.\, f\ x\ x$$

  - This is a well defined program if the argument given to it is a function that returns another function. However, if we give it a function that returns a non-function value

$$(\lambda f.\, \lambda x.\, f\ x\ x)(\lambda y.\, y)$$

    then the program reaches an erroneous 'stuck' state.
  - Currently, the semantics don't allow us to say something like f can only be of the form $f ::= \lambda x.\, \lambda y.\, x\ y$.

Invariants

- In programming, as in λ-calculus, an **invariant** is simply a condition that always remains true. For example,
  - a loop-invariant is a condition that is true at the beginning as well as the end of each iteration of the loop; a for-loop may have an inequality as its invariant, such as a condition **`x ≤ 10`**.
  - in the λ-calculus program $\lambda n.\frac{1}{n}$, the following condition is an invariant: n must be a non-zero number.
- Design considerations:
  1. How to write an invariant? That is, what is the 'language' of invariants?
  2. Can (or should) invariants be inferred from a program, or must they be provided by the programmer?
  3. Can (or should) an invariant be checked *statically* (i.e., before the program is run) or *dynamically* (i.e., at runtime)?

Invariants

Consider this Java code:

```
private double invert(double y) {
    assert y != 0;
    return 1/y;
}
```

We are using a built-in keyword "assert".

- It takes a boolean expression and raises an error if the expression evaluates to **false**.
- The 'language' of invariants is the same as the host language (in this case, Java). [design consideration 1]
- Programmer must provide this invariant. [design consideration 2]
- Invariant checked at runtime. [design consideration 3]

There are other invariants as well: the type of the parameter must be **double**.

- Again, the 'language' is the same as the host language.
- Programmer must provide this invariant.
- Invariant checked at compile-time.
- But keep in mind that a single language may perform some type checks statically and other type checks dynamically.

Data type | Data structure

A **data structure** is a description of a set of data and its internal organization such that the set is considered as a single entity.

- The organization is meant to facilitate certain operations to be performed efficiently.
- E.g., a binary search tree allows logarithmic search

An **abstract data type** is a mathematical model of a data structure and its operations.

- We can think of an abstract data type as an interface, and a data structure as its concrete implementation.

Data types

There are multiple ways in which we can formally think of data types:

- We can implicitly picture values from a **domain**. This is the **denotational semantics** approach.
- We can think of a type to be defined in terms of the internal structure of the data, where complex structures are described in terms of simpler constituents. This is the **structural semantics** approach.
- Along with the data, we also define a collection of operations that can be applied to objects of that type.
  - In Java, for example, these are defined (but not necessarily implemented) in an abstract class or interface.

Data type as an abstraction

- At the lowest level, computers are, of course, *un-typed* systems. Everything at the machine-level is binary.
- With each layer of abstraction, it becomes very helpful to think of a 'value' having a type:
  - integer: -3, -200, 5, 972
  - float: -5.38, 2.7182, 3.224e3
  - string: "CSE 216"
- In light of these simple examples, we can think of **data types** as
  - a set of concrete objects with some specific properties, or
  - a set of values that can be concretely constructed and represented, or
  - a set of values together with a collection of *operations* that can be performed on the elements of that set.

A Java programmer's way of thinking:

- these are 'instances' of the corresponding 'classes'

… and these define the 'interfaces'.

Data type as an abstraction

- When viewing the type of data as a set of properties or operations, an interesting idea is that of **duck typing**. It is based on an idea from abductive reasoning:

  > *If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.*

- Duck typing is essentially an approach that checks the type of data at runtime by looking at the runtime 'behavior' of the data.

Duck Typing

Any object can be used in any context, as long as its runtime behavior allows the operations.

```
class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()
whale = Whale()
lift_off(duck) # Duck flying
lift_off(airplane) # Airplane flying
lift_off(whale) # Error: 'Whale' object has no attribute 'fly'
```

Python example from https://en.wikipedia.org/wiki/Duck_typing

Data type as “expressiveness”

- The purpose of computer programs is to perform computations on data (or in other words, to process information).
- Information, of course, comes in various forms.
- Therefore, the set of data types available in a programming language is a good measure of how sophisticated the language is, in terms of its ease of information processing.
- A more “expressive” language can deal with complex types of information more easily.
- A language that allows the programmer to define new data types easily can become *dynamically expressive* as the need arises. Therefore, the ability to easily define new data types is an important component of expressiveness in a programming language.

Data Types as Descriptors

What can have a type?

- The type of a **constant** literal (i.e., a concrete value) is defined by which basic set it belongs to.
  - *E.g.*, 'a' is a character; 4.22 is float or double; 1 is an integer
- The type of an **expression** is the type of the value it evaluates to.
- The type of a **variable** is defined by the type of the value/expression it refers to.
- In some languages, subroutines, classes, and modules have their own types.

Everything "has a" type

- We can think conversely: the type of a construct (constant, variable, expression, etc.) is its *attribute*.
- This way, we view the type of an object as a **descriptor** of that object.
- That is, the object "has a" data type.
- Depending on whether a language is statically and/or dynamically typed, the 'type' attribute is **assigned*** at compile-time and/or runtime.

*We will sometimes use the verb 'assign' when talking about data types. To make sense of such statements, you should keep in mind this idea of the data type as a *descriptor*.

Typed and untyped languages

- Programming languages are often classified according to some of the major programming paradigms – **procedural**, **functional**, and **object oriented**.
- Within each paradigm, some languages are **typed** and others are **untyped**.
- Within computer science, "untyped" really means **dynamically typed**.
  - That is, a variable or an expression is assigned the type of the corresponding data (i.e., the "value" denoted by the variable or the expression) at runtime.
- Similarly, "typed" typically means **statically typed**.
- Within object oriented languages, C++ is typed and Smalltalk is untyped.

Type system

- A **type system** of a programming language is a set of rules that assigns a data type to the constructs of a program in that language.
- It provides a set of rules for (a) **type equivalence**, (b) **type compatibility**, and (c) **type inference**.
- It provides the (sometimes implicit) *context* for the operations on a data type.
  - Example 1: the binary operator "+" means concatenation if the arguments are strings, but arithmetic addition if the arguments are numeric.
  - Example 2: the unary operator "<<" in C++ is a 1-bit left-shift if the argument is an integer, but if the argument is an output stream, it means writing to the stream.
  - This is called **operator overloading**.

Before understanding these terms, though, we will need to first understand how types are checked in various languages. Thus, the next few slides divert to *type checking* and *type errors*.

Advantages of a type system

- **Documentation/legibility** – typed languages are easier to read and understand since the code itself provides partial documentation of what a variable actually means.
- **Safety** – typed languages provide early (compile-time) detection of some programming errors, since a type system provides checks for type-incompatible operations.
- **Efficiency** – typed languages can precisely describe the memory layout of all variables, since every 'instance' of an 'object' of a certain type will occupy the same amount of space.
  - Except for dynamically resizing objects like a list. But even then, we at least know how much memory each 'cell' of the list will occupy.
- **Abstraction** – typed languages force us to be more disciplined programmers. This is especially helpful in the context of large-scale software development.

Type errors and type safety

- A type error is a program error that results from the incompatible use of differing data types in a program's construct

```
int n = 2.55;
```

- To prevent (or at least discourage) type errors, a programming language puts in rules for **type safety**.
  - Type safety contributes to a program's correctness.
  - But keep in mind that it does not guarantee complete correctness.
  - Even if all operations in a program are type safe, there may still be bugs.
  - E.g., division of one number by another is type safe, but division by zero is unsafe unless the programmer explicitly handles that situation in some other manner.

Type checking

- **Type checking** is the process of verifying and enforcing the rules of type safety in a program.
- This may be done at compile-time, called **static type checking** (and the language is called a **statically typed language**).
- Or, it may be done at runtime, which is known as **dynamic type checking** (and the language is called a **dynamically typed language**).
- Another way to distinguish between the type checking in a language is based on how *strongly* it enforces the conversion of one data type to another.
  - If a language generally only allows automatic type conversions that do not lose information, it is called a **strongly typed language**.
  - Otherwise, it is called **weakly typed**.

Type checking

- Java is strongly typed, and uses a mix of static and dynamic type checking:

```
String aString = 1;          // static type check by javac
int    anInt   = 10.0;       // static type check by javac
Phone  phone   = (Phone) o; // dynamic type check by JVM
```

- Python is strongly typed, and uses dynamic type checking:

```
astring = "2"
anInt   = 10
result = anInt + astring ← runtime type check and type error
```

- Perl is weakly typed, and uses dynamic type checking:

```
$a = "2"
$b = 10
$a + $b // no error; implicit type conversion
```

Type checking

Each approach to type checking has its dis/advantages:

- **Strong static type checking**
  - \+ type errors are caught early at compile time
  - – verbose code
- **Strong dynamic type checking**
  - \+ quick prototyping with lesser ‘amount’ of code
  - – type errors are caught only at runtime
- **Weak dynamic type checking**
  - \+ least verbose code writing
  - – type errors are often not caught even at runtime
  - – unintended program behavior may occur due to implicit type conversion at runtime

Type equivalence

- As we mentioned earlier, a type system provides a set of rules for (a) **type equivalence**, (b) **type compatibility**, and (c) **type inference**.
- A type checking system usually has rules of the form:

  `if` two expressions have equivalent type

  `then` return that type

  `else` return **`TYPE_ERROR`**

Type equivalence asks the key question needed for the above rule:

When do two given expressions have equivalent types?

There are two possible approaches:

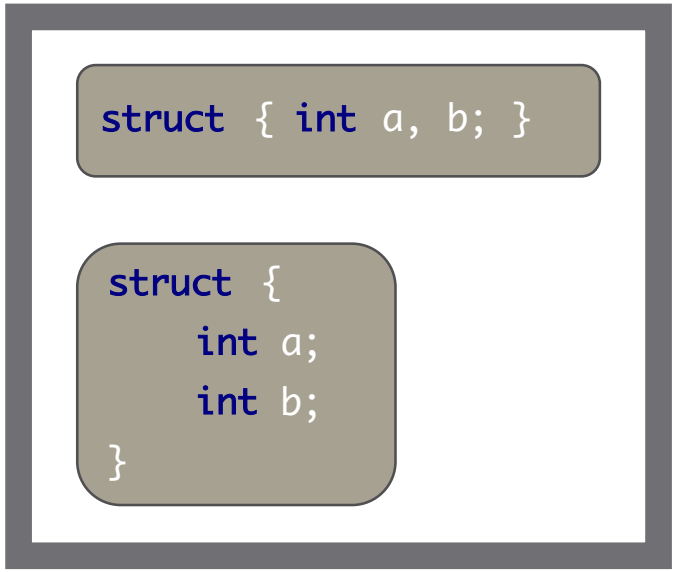
1. **Name equivalence**: two types are equal if and only if they have the same constructor expression (i.e., they are bound to the same name)
2. **Structural equivalence**: two types are equivalent if and only if they have the same "structure".

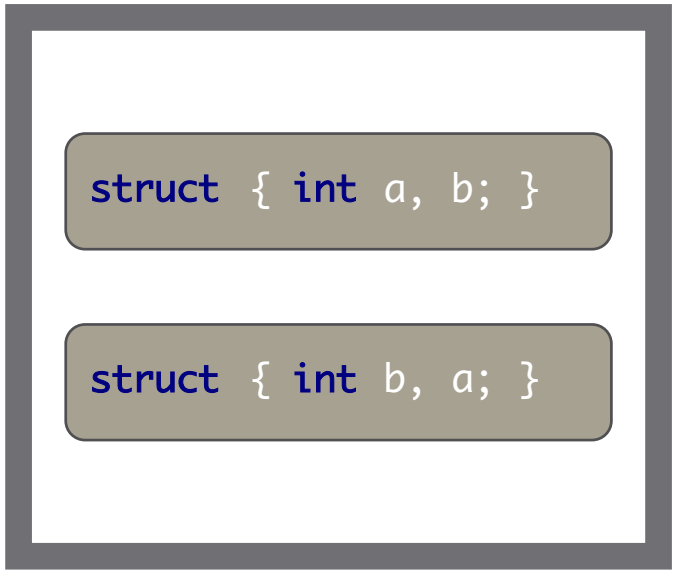
Type equivalence

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

x : student;
y : school;

x = y;
```

Name equivalence

- If this hypothetical language uses name equivalence, the last line will lead to a type error.
- Most modern languages (*e.g.*, Java, C#) use name equivalence because they assume that
  - if a programmer has gone through the trouble of repeatedly defining the same structure under different names,
  - then s/he probably wants these names to represent different types.





Type equivalence

Structural equivalence

- The exact definition of structural equivalence is a bit murky, and varies from one language to another.
  - People have differed over questions like “what really is a structure”, and “when should two structures be considered equivalent”.
- The format, or course, doesn’t matter. The two code bodies on the top are equivalent types.
- But what about the order of the components in a structure?
- This depends on the language. ML, for example, treats them as equivalent types.

Type equivalence

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

x : student;
y : school;

x = y;
```

- Structural equivalence is a simple in theory, but things get complicated when we get recursive or pointer-based types.
  - It is, in some sense, a low-level (i.e., *un*-abstract) implementation-oriented way of distinguishing types.
- In our hypothetical language example, both **`student`** and **`school`** have the same fields, and the fields have the same types.
  - But the programmer clearly would like to treat them as two *different* types, even though they are structurally identical.

Name equivalence

- It is sometimes a good idea to introduce synonymous names, e.g., for better readability of programs:

```
TYPE new_type = old_type; (* Modula-2 *)
TYPE human = person;
TYPE item_count = integer;
```

  - This is known as **type aliasing**, and the new type is called an **alias** of the old type.

- Name equivalence can be
  1. **strict** (aliased types are considered to be distinct) or
  2. **loose** (aliased types are considered to be equivalent).

Name equivalence

- In this Modula-2 code, **`stack`** is meant to be an abstraction that allows for the creation of a stack of any desired type (in this case, **`integer`**).
- If alias types were not considered equivalent, a programmer would have to replace every occurrence of **`stack_element`** with **`integer`**.
  - The language Modula-2 uses loose name equivalence to avoid this problem.
  - *But this is probably better designed using generic types (e.g., in Java and C#).*
- Many modern languages use a 'middle-ground' approach between loose and strict name equivalence to indicate whether an alias represents a **derived type**.

```
TYPE    stack_element = integer;
MODULE  stack;
IMPORT  stack_element;
EXPORT  push, pop;
 . . .
PROCEDURE  push(elem : stack_element);
 . . .
PROCEDURE  pop() : stack_element;
```

Name equivalence

- In other scenarios, aliased types should *not* be treated the same.
- Using **derived types** (a.k.a. subtypes) resolves these issues. This brings forth the concept of a type hierarchy.
- A subtype is **type compatible** with its parent type. This way, we have a one-sided "is-a" relation instead of complete type equivalence:
  - subtype "is-a" parent type (every **`celsius_temp`** is-a real number)
  - but parent is not a subtype (but not every real number is a **`celsius_temp`**)

```
TYPE  celsius_temp  = REAL;
      fahrenheit_temp  = REAL;
VAR   c: celsius_temp;
VAR   f: fahrenheit_temp;
. . .
f := c; (* this should probably not be
           allowed *)
```

Type conversion

In statically typed languages, the values expected in a context must be of a certain type, e.g., **`x := expression`**, both sides of the assignment must have the same type.

If this expectation is violated, either the programmer needs perform an explicit type conversion (also called a type cast), or it will cause a type error.

A. The types are structurally equivalent, but the language uses name equivalence.
   - In this case, the two structurally equivalent types have the same low-level representation, and the conversion happens *implicitly*, with no additional code.
   - We are simply re-interpreting the bits without changing the underlying implementation. This is an **implicit non-converting typecast**.

```
/* Java example */
/* implicit non-converting cast when Student is a subtype of Person */
Person p = new Student();
```

Type conversion

In statically typed languages, the values expected in a context must be of a certain type, e.g., **`x := expression`**, both sides of the assignment must have the same type.

If this expectation is violated, either the programmer needs perform an explicit type conversion (also called a type cast), or it will cause a type error.

B. The types have different sets of values, but the intersecting values have a common representation.
   - Additional code must be executed at runtime to make sure that the value does, indeed, lie at the intersection.
   - This is an **explicit non-converting typecast**.

```
/* Java example */
/* If the low-level implementation of ‘p’ is identical to the type ‘Student’ */
Student s = (Student) p;
```

Type conversion

In statically typed languages, the values expected in a context must be of a certain type, e.g., **`x := expression`**, both sides of the assignment must have the same type.

If this expectation is violated, either the programmer needs perform an explicit type conversion (also called a type cast), or it will cause a type error.

C. The types have different low-level representations. Nonetheless, it is possible to define some obvious correspondence. For example:
   i. a 32-bit integer can be converted to a a **`double`** with no ambiguity.
   ii. a **`double`** can be converted to an integer (by rounding or truncating).
   - Most languages provide low-level machine instructions for such a conversion. Unlike the previous type conversions, every such conversion is a **converting cast**.

```
/* Java example */
int k = (int) 3.14; // truncating a floating-point number
double d = 3; // converting cast with no loss of precision
```

Type compatibility

- Most languages require only that types be “compatible”, instead of completely equivalent, e.g.,
  - in an assignment statement, the type of the right-hand side must be compatible with the type of the left-hand side.
  - in an operation (say, +), the operands must be compatible with some common type that supports the ‘+’ operation (e.g., integers, floats, doubles, strings, or maybe even sets).
  - in a subroutine call, the types of
    1. the arguments passed to the subroutine must be compatible with the types of that subroutine’s formal parameters, and
    2. any formal parameters passed back to the caller must be compatible with the types of the corresponding arguments.
- The exact definition of type compatibility varies a lot between languages. But the core is the that it poses the following question:

When can a value of type *A* be used when type *B* is expected?

Type coercion

- Whenever a language allows a value of type *A* to be be used when another type *B* is expected, the language implementation performs an *automatic implicit* conversion to the expected type *B*.
  - There are languages that don't allow this at all (e.g., Haskell).
  - **If** the language allows this, the "how" is language-dependent, and which coercions are performed/allowed is also language-dependent.

```
 /* Type coercion in Java */

/* converting cast as coercion */
 double d = 3;

 /* non-converting cast as coercion */
 Object o = "type coercion example";
 if (o instanceOf String) {
     String s = (String) o;
 }
```