What is a Type?

- A set of values
What is a Type?

- A set of values
  - Together with a set of operations on these values that possess certain properties
Topics

- Data types in modern languages
  - simple and compound types
- Type declaration
- Type inference and type checking
- Type equivalence, compatibility, conversion and coercion
- Strongly/Weakly/Un-typed languages
- Static Vs Dynamic type checking
Simple Types

- **Predefined**
  - `int`, `float`, `double`, etc in C

- All other types are constructed, starting from predefined (aka primitive) types
  - **Enumerated:**
    - `enum colors {red, green, blue} in C`
    - `type colors = Red|Green|Blue in OCAML`

**Built-in**     **User-defined**

*Algebraic data types*
Detour: Evolution of Programming Languages

- Imperative (Assignment-based)
  - FORTRAN (50's)
  - PL/1
  - ALGOL
  - PASCAL
  - C
  - C++
  - Go

- Declarative
  - LISP (50's)
  - SIMULA
  - CSP
  - Smalltalk

- Functional
  - Object-oriented
  - Standard ML
  - Scheme
  - Java

- Logic
  - Prolog

- Pure Functional
  - Haskell
  - Type classes
  - traits

- Automatic backtracking
  - $f(x) = P(x)$
  - $x = \ldots$
Compound Types

- Types constructed from other types using type constructors
  - Cartesian product (*)
  - Function types (→)
  - Union types (∪)
  - Arrays
  - Pointers
  - Recursive types

- Predefined type constructors
- User-defined type constructors
Let $I$ represent the integer type and $R$ represent real type.

The cross product $I \times R$ is defined in the usual manner of product of sets, i.e.,

$$I \times R = \{(i, r) \mid i \in I, r \in R\}$$

Cartesian product operator is non-associative.
Labeled Product types

- In Cartesian products, components of tuples don’t have names.
  - Instead, they are identified by numbers.
- In labeled products each component of a tuple is given a name.
- Labeled products are also called records (a language-neutral term)
Labeled Product types (Continued)

- **struct** is a term that is specific to C and C++

```c
struct t {int a; float b; char *c;}; in C
```

```c
int * float * ptr (char)
```

$$1 \times F \times \text{ptr}(c)$$

$$(3, 2.5, \ldots)[1]$$
Function Types

\( T_1 \rightarrow T_2 \) is a function type

- Type of a function that takes one argument of type \( T_1 \) and returns type \( T_2 \)

OCAML supports functions as first class values.

- They can be created and manipulated by other functions.

In imperative languages such as C, we can pass pointers to functions, but this does not offer the same level of flexibility.

- E.g., no way for a C-function to dynamically create and return a pointer to a function;
- rather, it can return a pointer to an EXISTING function

Recent versions of C++ (as well Python, JavaScript and recent Java versions) support dynamically created functions (aka lambda abstractions)

- See Functional Programming for Imperative Programmers for a discussion of functional programming features in C++.
Union types

- Union types correspond to set unions, just like product types corresponded to Cartesian products.
- `->` operator is right-associative, so we read the type as float -> (float -> float).
- Unions can be tagged or untagged. C/C++ support only untagged unions:

```c
union v {
    int ival;
    float fval;
    char cval;
};
```
Tagged Unions

- In untagged unions, there is no way to ensure that the component of the right type is always accessed.
  - E.g., an integer value may be stored in the above union, but due to a programming error, the fval field may be accessed at a later time.
  - fval doesn’t contain a valid value now, so you get some garbage.

- With tagged unions, the compiler can perform checks at runtime to ensure that the right components are accessed.

- Tagged unions are NOT supported in C/C++. 
Tagged Unions (Continued)

- Pascal supports tagged unions using VARIANT RECORDs

```
RECORD
  CASE b: BOOLEAN OF
    TRUE: i: INTEGER; |
    FALSE: r: REAL END
END
END
```

- Tagged union is also called a discriminated union

\[ x \cdot \lambda = 10 \]

\[ x \cdot r = x \cdot r \]

\[ \text{type } x = \]

\[ \text{INT } \&\text{ int } 1 \]

\[ \text{FLOAT } \&\text{ float } 1 \]

\[ \text{INT(10)} \]

\[ \text{FLOAT(5.0)} \]

\[ \text{algebraic data type} \]
Array types

- Array construction is denoted by
  - `array(<range>, <elementType>).`

- C-declaration
  - `int a[5];`
  - defines a variable a of type `array(0-4, int)`

- A declaration
  - `union tt b[6][7];`
  - declares a variable b of type `array(0-4, array(0-6, union tt))`

- We may not consider range as part of type
A pointer type will be denoted using the syntax

- `ptr(<elementType>)`
- where `<elementType>` denote the types of the object pointed by a pointer type.

The C-declaration

- `char *s;`
- defines a variable `s` of type `ptr(char)`

A declaration

- `int (*f)(int s, float v)`
- defines a (function) pointer of type `ptr(int*float → int)`
Recursive types

- Recursive type: a type defined in terms of itself.

- Example in C:

```c
struct IntList {
    int hd;
    IntList tl;
};
```

- Does not work:
  - This definition corresponds to an infinite list.
  - There is no end, because there is no way to capture the case when the tail has the value “nil”
Recursive types (Continued)

- Need to express that tail can be nil or be a list.

- Try: variant records:

  ```pascal
  TYPE charlist = RECORD
    CASE IsEmpty: BOOLEAN OF
    TRUE: /* empty list */ |
    FALSE:
      data: CHAR;
      next: charlist;
    END
  END

  END

- Still problematic: Cannot predict amount of storage needed.
Recursive types (Continued)

- Solution in typical imperative languages:
  - Use pointer types to implement recursive type:

    ```
    struct IntList {
        int hd;
        IntList *tl;
    };
    ```

- Now, tl can be:
  - a NULL pointer (i.e., nil or empty list)
  - or point to a nonempty list value

- Now, IntList structure occupies only a fixed amount of storage
Recursive types In OCAML

- Direct definition of recursive types is supported in OCAML using type declarations.

- Use pointer types to implement recursive type:

```ocaml
# type intBtree =
   LEAF of int
 | NODE of int * intBtree * intBtree;;

type intBtree = LEAF of int | NODE of int * intBtree * intBtree
```

- We are defining a binary tree type inductively:
  - Base case: a binary tree with one node, called a LEAF
  - Induction case: construct a binary tree by constructing a new node that stores an integer value, and has two other binary trees as children
Polymorphism

 Ability of a function to take arguments of multiple types.

 The primary use of polymorphism is code reuse.

 Functions that call polymorphic functions can use the same piece of code to operate on different types of data.
Overloading (adhoc polymorphism)

- Same function **NAME** used to represent different functions
  - implementations may be different
  - arguments may have different types

- Example:
  - operator ‘+’ is overloaded in most languages so that they can be used to add integers or floats.
  - But implementation of integer addition differs from float addition.
  - Arguments for integer addition or ints, for float addition, they are floats.

- Any function name can be overloaded in C++, but not in C.

- All virtual functions are in fact **overloaded** functions.
Polymorphism & Overloading

- **Parametric polymorphism:**
  - same function works for arguments of different types
  - same code is reused for arguments of different types.
  - allows reuse of “client” code (i.e., code that calls a polymorphic function) as well

- **Overloading:**
  - due to differences in implementation of overloaded functions, there is no code reuse in their implementation
  - but client code is reused
Parametric polymorphism in C++

Example:

```cpp
//template <class C>
C min(const C* a, int size, C minval) {
    for (int i = 0; i < size; i++)
        if (a[i] < minval)
            minval = a[i];
    return minval;
}
```

Note: same code used for arrays of any type.

- The only requirement is that the type support the "<" and "=" operations

- The above function is parameterized wrt class C
  - Hence the term "parametric polymorphism".

Unlike C++, C does not support templates.
Code reuse with Parametric Polymorphism

- With parametric polymorphism, same function body reused with different types.
- Basic property:
  - does not need to "look below" a certain level
  - E.g., min function above did not need to look inside each array element.
  - Similarly, one can think of length and append functions that operate on linked lists of all types, without looking at element type.
Code reuse with overloading

- No reuse of the overloaded function
  - there is a different function body corresponding to each argument type.

- But client code that calls a overloaded function can be reused.

Example

- Let C be a class, with subclasses C1,...,Cn.
- Let f be a virtual method of class C
- We can now write client code that can apply the function f uniformly to elements of an array, each of which is a pointer to an object of type C1,...,Cn.
Example:

```c
void g(int size, C *a[]) {
    for (int i = 0; i < size; i++)
        a[i]->f(...);
}
```

Now, the body of function `g` (which is a client of the function `f`) can be reused for arrays that contain objects of type $C_1$ or $C_2$ or ... or $C_n$, or even a mixture of these types.
Type Equivalence

- **Structural equivalence**: two types are equivalent if they are defined by identical type expressions.
  - array ranges usually not considered as part of the type
  - record labels are considered part of the type.

- **Name equivalence**: two types are equal if they have the same name.

- **Declaration equivalence**: two types are equivalent if their declarations lead back to the same original type expression by a series of redeclarations.
Type Equivalence (contd.)

- Structural equivalence is the least restrictive.
- Name equivalence is the most restrictive.
- Declaration equivalence is in between.

TYPE t1 = ARRAY [1..10] of INTEGER; VAR v1: ARRAY [1..10] OF INTEGER;

TYPE t2 = t1; VAR v3,v4: t1; VAR v2: ARRAY [1..10] OF INTEGER;

<table>
<thead>
<tr>
<th></th>
<th>Structurally equivalent?</th>
<th>Declaration equivalent?</th>
<th>Name equivalent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1, t2</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>v1, v2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>v3, v4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Declaration equivalence

- In Pascal, Modula use decl equivalence
- In C
  - Decl equiv used for structs and unions
  - Structural equivalence for other types.
    ```
    struct { int a; float b; } x;
    struct { int a; float b; } y;
    x and y are structure equivalent but not declaration equivalent.
    ```
  - typedef int* intp;
    typedef int** intpp;
    intpp v1;
    intp *v2;
  - v1 and v2 are structure equivalent.
Type Compatibility

- Weaker notion than type equivalence
- Notion of compatibility differs across operators
- Example: assignment operator:
  - \( v = \text{expr} \) is OK if \(<\text{expr}>\) is type-compatible with \(v\).
  - If the type of \(\text{expr}\) is a Subtype of the type of \(v\), then there is compatibility.
- Other examples:
  - In most languages, assigning integer value to a float variable is permitted, since integer is a subtype of float.
  - In OO-languages such as Java, an object of a derived type can be assigned to an object of the base type.
Procedure parameter passing uses the same notion of compatibility as assignment.

Note: procedure call is a 2-step process:
- assignment of actual parameter expressions to the formal parameters of the procedure
- execution of the procedure body

Formal parameters are the parameter names that appear in the function declaration.

Actual parameters are the expressions that appear at the point of function call.
Type Checking

- **Static (compile time)**
  - Benefits
    - no run-time overhead
    - programs safer/more robust

- **Dynamic (run-time)**
  - Disadvantages
    - runtime overhead for maintaining type info at runtime
    - performing type checks at runtime
  - Benefits
    - more flexible/more expressive
Examples of Static and Dynamic Type Checking

- C++ allows
  - **Upcasts**: casting of subclass to superclass (always type-safe)
  - **Downcasts**: superclass to subclass (not necessarily type-safe) – but no way to check since C++ is statically typed.
    - Actually, runtime checking of downcasts is supported in C++ but is typically not used due to runtime overhead

- Java uses combination of static and dynamic type-checking to catch unsafe casts (and array accesses) at runtime.
Type Checking (Continued)

- Type checking relies on type compatibility and type inference rules.
- Type inference rules are used to infer types of expressions. e.g., type of \((a+b)+c\) is inferred from type of \(a\), \(b\) and \(c\) and the inference rule for operator '+'.
- Type inference rules typically operate on a bottom-up fashion.
- Example: \((a+b)+c\)

```
            +:float
            /   \
       +:float    c:float
       /     \    
 a:int    b:float
```
In OCAML, type inference rules capture bottom-up and top-down flow of type info.

Example of Top-down: let f x y : float * int = (x, y)

```
f : float * int

x : float
y : int
```

Here types of x and y inferred from return type of f.

Note: Most of the time OCAML programs don’t require type declaration.
  - But it really helps to include them: programs are more readable, and most important, you get far fewer hard-to-interpret type error messages.
Strong Vs Weak Typing

- Strongly typed language: such languages will execute without producing uncaught type errors at runtime.
  - no invalid memory access
    - no seg fault
    - array index out of range
    - access of null pointer
  - No invalid type casts

- Weakly typed: uncaught type errors can lead to undefined behavior at runtime

- In practice, these terms used in a relative sense

- Strong typing does not imply static typing
Type Conversion

- Explicit: Functions are used to perform conversion.
  - example: strtol, atoi, itoa in C; float and int etc.

- Implicit conversion (coercion)
  - example:
    - If a is float and b is int then type of a+b is float
    - Before doing the addition, b must be converted to a float value. This conversion is done automatically.

- Casting (as in C)

- Invisible “conversion:” in untagged unions
Data Types Summary

- Simple/built-in types
- Compound types (and their type expressions)
  - Product, union, recursive, array, pointer
- Parametric Vs subtype polymorphism, Code reuse
- Polymorphism in OCAML, C++,
- Type equivalence
  - Name, structure and declaration equivalence
- Type compatibility
- Type inference, type-checking, type-coercion
- Strong Vs Weak, Static Vs Dynamic typing