CSE 3302
Programming Languages
Lecture 4: Data Types

(based on the slides by Chengkai Li)
Leonidas Fegaras
University of Texas at Arlington

Data Types

- What is a data type?
  A designation with certain properties:
  - The values that can be stored, the internal representation,
    the operations, …
- A data type is a set of values
  - e.g., int in Java:
    ```java
    int x;
    x ∈ Integers = [-2147483648, 2147483647]
    ```
- A data type is also a set of operations on the values

Why are data types important?

- Allows consistency checking between variable/function declarations and their use
  - Static type-checking catches type errors at compile time
- Ensures that most unsafe programs will be rejected at compile-time \( \Rightarrow \) no data-corrupting errors at run-time
- Disambiguates overloading
- Example: \( z = x / y \); (Java)
  - int x, y; x=5; y=2;
    - Integer division, \( x/y \) results in 2.
    - int z; z = 2;
    - double z; z = 2.0;
  - double x, y; x=5.0; y=2.0;
    - floating-point division, \( x/y \) results in 2.5
    - int z; wrong!
    - double z; z = 2.5;

Java Types

```
Primitive

<table>
<thead>
<tr>
<th>boolean</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
</tr>
</tbody>
</table>
```

Reference

```
Array

<table>
<thead>
<tr>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
</tr>
</tbody>
</table>
```

Type structure of Java
Simple Data Types

- No internal structure:
  e.g., integer, double, character, and boolean
- Often directly supported in hardware
  - machine dependent
- Most predefined types are simple types
  - Exceptions: String in Java
- Some simple types are not predefined
  - Enumerated types
  - Subrange types

Type structure of C

Enumerated Types

Ordered sets whose elements are named and listed explicitly.

- Examples:
  enum Color_Type {Red, Green, Blue};  (C)
  type Color_Type is (Red, Green, Blue);  (Ada)
  datatype Color_Type = Red | Green | Blue;  (ML)

- Operations:
  Successor and predecessor

Ada Example

type Color_Type is (Red, Green, Blue);

x : Color_Type := Green;
x : Color_Type’Succ(x);
x : Color_Type’Pred(x);
put(x); -- prints GREEN

- No assumptions about the internal representation of values
- Prints the value name itself
Pascal Example

type
cardsuit = (club, diamond, heart, spade);
card = record
  suit: cardsuit;
  value: 1 .. 13;
end;

var
  hand: array [ 1 .. 13 ] of card;

  Succ(diamond) = heart; Pred(spade) = heart;
  club < heart; is true
  for acard := club to heart do ...

C Example

#include <stdio.h>
enum Color {Red, Green, Blue};
enum Courses (CSE111=1, CSE3302=3, CSE3310=3, CSE5555=4);
main() {
  enum Color x = Green;
  enum Courses c = CSE3302;
  x++;
  printf("%d\n",x);
  printf("%d\n",Blue+1);
  printf("%d\n",c);
  return 0;
}

• Enum in C is simply int
• Can customize the values

Java Example

public class Card {
  public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
                    SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
  public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

  for (Suit suit : Suit.values())
    for (Rank rank : Rank.values())
      protoDeck.add(new Card(rank, suit));

Evaluation of Enumeration Types

• Efficiency -- e.g., compiler can select and use a compact efficient representation (e.g., small integers)

• Readability -- e.g. no need to code a color as a number

• Maintainability -- e.g., adding a new color doesn’t require updating hard-coded constants

• Reliability -- e.g. compiler can check operations and ranges of value
C Example for Maintainability

```c
enum Color {White, Green, Blue, Black};
enum Color {White, Yellow, Green, Blue, Black};
main(){
    enum Color x = Black;
    int i = x;
    while (i >= White){
        if (i < Green)
            printf("this is a light color!\n");
        i--;
    }
    // What if no enumeration?
    if (i < 1) printf("this is a light color!\n");
    // Has to be changed to:
    if (i < 2) printf("this is a light color!\n");
}
```

Ada Example for Reliability

```ada
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
type DIRECTION is (NORTH, EAST, SOUTH, WEST);

GOAL : DIRECTION;
TODAY : DAY;
START : DAY;

TODAY := MON;
GOAL := WEST;
START := TODAY;

TODAY := WEST; -- Illegal: WEST is not a DAY value
TODAY := 5; -- Illegal: 5 is not a DAY value
TODAY := TODAY + START; -- Illegal: *+* is not defined for DAYS
```

Subrange Types

Contiguous subsets of simple types, with a least and greatest element.

- Type: Digit_Type is range 0..9; (Ada)
- Not available in C, C++, or Java. Equivalent to:
```
byte digit;  //=-128..127
```
- Defined over ordinal types:
  - Ordered, every value has a next/previous element
    - E.g., integer, enumerations, and subrange itself

Type constructors: Defining new types

- Type constructors as set operations:
  - Cartesian product
  - Union
  - Subset
  - Functions (including vectors/arrays)
- Some type constructors do not correspond to set operations (e.g., pointers)
- Some set operators do not have corresponding type constructors (e.g., intersection)
Cartesian Product

- **Ordered Pairs of elements from U and V**
  \[ U \times V = \{ (u, v) \mid u \in U \text{ and } v \in V \} \]

- **Operations:**
  - projection
    \[ p_1 : U \times V \rightarrow U; \quad p_2 : U \times V \rightarrow V \]
    \[ p_1 ((u, v)) = u; \quad p_2 ((u, v)) = v \]

Examples

- **struct in C**
  ```c
  struct IntCharReal
  {
    int i;
    char c;
    double r;
  }
  
  int x char x double
  ```

- **record in Ada**
  ```ada
  type IntCharReal is record
    i: integer;
    c: character;
    r: float;
  end record;
  ```

Are these the same type?

```c
struct IntCharReal
{
  int i;
  char c;
  double r;
}
```

What about these types?

```c
struct IntCharReal
{
  int i;
  char c;
  double r;
}
```

```c
struct IntCharReal
{
  char c;
  int i;
  double r;
}
```
Records are not exactly Cartesian products

- Component selector: projection by a component name

```c
struct IntCharReal x;
  x.i;
```

- Most languages consider component names to be part of the type
- The previous two types can be considered different, even though they represent the same Cartesian product
  - They have different component selectors

ML: Pure Cartesian Products

- `type IntCharReal = int * char * real;`
- `(2, "a", 3.14)`
- `#3(2, "a", 3.14) = 3.14`

Union

- `U ∪ V = { x | x ∈ U or x ∈ V }`
  - data items with different types are stored in overlapping region, reducing memory allocation
  - Only one type of value is valid at one time
  - e.g.,
  ```c
  union IntOrReal {
    int i;
    double r;
  }
  ```
  ```c
  union IntOrReal x;
  x.i = 1;
  printf("%f\n", x.r);
  ```
  - Can be unsafe

Undiscriminated Union in C
Discriminated Union in C++

```cpp
struct IntOrReal {
    bool isInt;
    union {
        int i;
        double r;
    }
};

IntOrReal x;
if (x.isInt) printf("%d\n", x.i);
else printf("%f\n", x.r);
```

- Far safer now (but not safe)

Discriminated Union in Ada

```ada
type Disc is (Int, Real);
type IntOrReal (which: Disc) is record
    case which is
        when Int => i: integer;
        when Real => r: float;
    end case;
end record;

x: IntOrReal := (Real, 2.3);
put (x.i);  -- generates ERROR
```

- Variant record (with tag or discriminator)
- Safe: programmers cannot create inconsistent data

Discriminated Union in Pascal

- Variant record

- Can be unsafe:
  - the tag is optional
  - the tag can be set inconsistently

Discriminated Union in ML

```ml
datatype IntOrReal =
    IsInt of int | IsReal of real;

val x = IsReal(2.3);
```
“Union” in Java

- public abstract class A {...};
- public class B extends A {...};
- public class C extends A {...};

Abstract class A: is the union of B and C

- Discriminated union: instanceof

Subset

- \( U = \{ v \mid v \text{ satisfies certain conditions and } v \in V \} \)

- Ada subtype

Example 1
- type Digit_Type is range 0..9;
- subtype IntDigit_Type is integer range 0..9;

subtype in Ada

- Example 2
  type Disc is (IsInt, IsReal);
  type IntOrReal (which: Disc) is record
    case which is
    when IsInt => i: integer;
    when IsReal => r: float;
  end case;
end record;

  subtype IRInt is IntOrReal(IsInt);
  subtype IRReal is IntOrReal(IsReal);
  x: IRReal := 2.3;

Powerset

- \( P(U) = \{ U' \mid U' \subseteq U \} \)

- Example: Pascal
  - set of <ordinal type>
    - var S: set of 1..10;
    - var S: 1..10;
    - What’s the difference?

- Element order is not significant
- Values are distinct
set of in Pascal

```pascal
var S, T: set of 1 .. 10;
S := [1, 2, 3, 5, 7];
T := [1 .. 6];

Set operations can be performed on the variables:
- T := S*T;
- if T = [1, 3, 5] then ..;
- if 3 in S then ..;
- if S<=T then ..;

∩ (\cap) \cup (+) - (\neg) = (=) \neq (<>)
\supset (\supset) \supseteq (\supseteq) \subseteq (\subseteq) \in (\in)
```

C/C++ arrays

- Allocated on run-time stack
- Its size is statically specified

```c
typedef int TenIntArray [10];
typedef int IntArray [];

TenIntArray x;
int y[5];
int z[] = {1,2,3,4};
IntArray w = {1,2};
IntArray w;    //illegal
int n = ...  //from user input
int a[n];    //illegal
```

Arrays and Functions

```
f: U \rightarrow V

index type  \begin{align*}
\text{component type}\end{align*}

- Index types:
  - [0, ...) (C/C++/Java)
  - Ordinal type (Ada/Pascal)
```

Java vectors

- Allocated on heap
- The size is dynamically specified
- The size can be obtained by .length

```java
int n = ...  //from user input
int [] x = new int [n];
System.out.println(x.length);
```
Ada arrays

- The size is dynamically specified
- Index is from a set of subscripts

type IntToInt is array(integer range <>) of integer;

get(n); // from user input
x: IntToInt(1..n);
for i in x’range loop
  put(x(i));
end loop;

Multi-dimensional arrays

- C/C++
  int x[10][20];

- Java
  int [][] x = new int[10][20];

- Ada
  These two are different
  type Matric_Type is array(1..10, -10..10) of integer;
  x(i,j);
  type Matric_Type is array(1..10) of array (-10..10) of integer;
  x(i)[];

Storage

- Storing x: array(1..10, -10..10) of integer
  - row-major form
    x[1,-10],x[1,-9],..,x[1,10],x[2,-10],..,x[2,10],x[3,-10],..
  - column-major form
    x[1,-10],x[2,-10],..,x[10,-10],x[1,9],..,x[10,9],x[1,8],..

- Passing arrays to functions:
  - C/C++
    int array_max(int a[][20], int size)
  - Java
    int array_max(int [][] a)

Function types in C

typedef int (*IntFunction)(int);

int square(int x) { return x*x; }

IntFunction f = square;

int evaluate(IntFunction g, int value){
  return g(value);
}
...
printf("%d\n",evaluate(f,3));
Function Types in ML

type IntFunction = int -> int;

fun square(x: int) = x * x;

val f = square;

Fun evaluate(g:IntFunction, value:int) = g value;
... evaluate(f,3);

Function Types in Java

interface Comparison {
  boolean compare ( int x, int y );
}

void sort ( int[] A, Comparison cmp ) {
  for (int i = 0; i<A.length; i++)
    for (int j=i+1; j<A.length; j++)
      if (cmp.compare(A[i],A[j]))
        ...
  }

class Leq implements Comparison {
  boolean compare ( int x, int y ) { return x <=y; }
  }

sort(A,new Leq());

... or better

class Comparison {
  abstract boolean compare ( int x, int y );
}

  sort(A,new Comparison() {
    boolean compare ( int x, int y ) { return x <=y; } })

Vectors vs Lists

Functional languages:

• Vectors: like arrays, more flexible, dynamically resizable

• Lists: like vectors, can only be accessed by counting down from the first element.
Pointers

- A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null)
- Advantages:
  - Addressing flexibility: address arithmetic, explicit dereferencing (*) and address-of (&), domain type is not fixed (void *)
  - Dynamic storage management
  - Recursive data structures
    - e.g., linked lists, trees
    - struct CharListNode
      - char data;
      - struct CharListNode* next;
    - Types of struct CharListNode* CharList;

Pointers with Pointers

- Dangling pointers (dangerous)
  - int *a, *b;
  - a = (int *) malloc(sizeof(int));
  - *a = 1;
  - b = a;
  - free(a);
  - printf("%d\n", *b);

Problems with Pointers

- Alias (with side-effect)
  - int *a, *b;
  - a = (int *) malloc(sizeof(int));
  - *a = 2;
  - b = (int *) malloc(sizeof(int));
  - *b = 3;
  - b = a;
  - *b = 4;
  - printf("%d\n", *a);

Problems with Pointers

- Space leaks: garbage
  - wastes memory
    - int *a;
    - a = (int *) malloc(sizeof(int));
    - *a = 2;
    - a = (int *) malloc(sizeof(int));
Type System

- **Type Constructors:** Build new data types upon simple data types

- **Type Checking:** The translator checks if data types are used correctly
  - **Type Inference:** Infers the type of an expression whose data type is not given explicitly
    - e.g., \( x/y \)
  - **Type Equivalence:** Compares two types and decides if they are the same
    - e.g., \( x/y \) and \( z \)
  - **Type Compatibility:** Can we use a value of type \( A \) in a place that expects type \( B \)?
    - Nontrivial with user-defined types and anonymous types

Strongly-Typed Languages

- Strongly-typed: (Ada, ML, Haskell, Java, Pascal)
  - Most data type errors detected at compile-time
  - A few errors are checked during run-time (e.g., subscript out of array bounds)

- Pros:
  - No data-corrupting errors can occur during execution
    - i.e., a type-correct program cannot cause a data error
  - Efficiency in translation and execution
  - Security/reliability

- Cons:
  - May reject safe programs (since legal programs are a subset of safe programs)
  - Burden on programmers: may often need to provide explicit type information

Weakly-typed and untyped languages

- Weakly-typed: C/C++
  - e.g., interoperability of integers, pointers, arrays

- Untyped (dynamically typed) languages: Scheme, Smalltalk, Perl
  - All type checking is performed at execution time
  - May produce run-time errors too frequently
  - May delay detection of some hard-to-find errors

Security vs. Flexibility

- Strongly-typed:
  - Type-correct programs cannot cause data errors
  - Large amount of type information supplied by programmers
  - May occasionally be too restrictive: may reject some safe (but type-incorrect) problem

- Untyped:
  - Run-time errors (but no data-corruption)
  - Delays error detection
  - Reduces the amount of type information the programmer must supply
Security vs. flexibility

- Strongly-typed:
- A type system tries to maximize both flexibility and security
  - flexibility: reduces the number of safe illegal programs and reduces the amount of type information the programmer must supply

- Flexibility, no explicit typing or static type checking vs.
- Maximum restrictiveness, static type checking

Type Equivalence

- How to decide if two types are the same?
- Structural Equivalence
  - Types are sets of values
  - Two types are equivalent if they contain the same values
- Name Equivalence

Structural Equivalence

- struct RecA {
  char x;
  int y;
}
- struct RecB {
  char x;
  int y;
}
- struct RecC {
  char u;
  int v;
}
- struct RecD {
  int y;
  char x;
}

Type Equivalence in C

- In C:
  struct RecA {
    char x;    int y;
  };
  struct RecB {
    char x;    int y;
  };
  struct RecA a;
  struct RecB b;

  b=a;  // Error: incompatible types in assignment
Type Equivalence in C

- In C:
  ```c
  struct RecA {
    char x; int y;
  };
  struct RecB {
    char x; int y;
  };
  struct RecA a;
  struct RecB* b;

  b = &a;  // Warning: incompatible types in assignment
  ```

- In C:
  ```c
  struct RecA {
    char x; int y;
  };
  struct RecB {
    char x; int y;
  };
  struct RecA a;
  struct RecB* b;

  b = (struct RecB*)&a;  // OK, but does not mean they are equivalent
  ```

Type Equivalence in Java

- In Java:
  ```java
  class A {
    char x; int y;
  };
  class B {
    char x; int y;
  };

  A a = new B();  // ?
  ```

The Type Equivalence Algorithm

- If structural equivalence is applied:
  ```c
  struct RecA {
    char x; int y;
  };
  struct RecB {
    char x; int y;
  };
  struct RecA a;
  struct RecB b;

  b = a;
  ```
Replacing the names by declarations

typedef struct {
    char x;  int y;
} RecB;
RecB b;

struct {
    char x;  int y;
} c;

Replacing the names by declarations?

typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode {
    char data;  CharList next;
};

struct CharListNode2 {
    char data;  CharList2 next;
};

Cannot do that for recursive types

typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode {
    char data;  struct CharListNode* next;
};

struct CharListNode2 {
    char data;  struct CharListNode2* next;
};

There are techniques for dealing with this
Structural Equivalence

- Can be complicated when there are names, anonymous types, and recursive types
- A simpler, but stricter rule: name equivalence

Name Equivalence

```c
struct RecA { char x; int y; };
typedef struct RecA RecB;
struct RecA *a;
RecB *b;
struct RecC c;
struct { char x; int y; } d;
struct { char x; int y; } e,f;
a=&c;    // ok
a=&d;    // warning: incompatible pointer type
b=&d;    // warning: incompatible pointer type
a=b;    // ok. typedef creates alias for existing name
   e=d;  // error: incompatible types in assignment
```

Type Equivalence in C

- Name Equivalence: struct, union
- Structural Equivalence: everything else — typedef doesn’t create a new type

Example

```c
struct A { char x; int y; };
struct B { char x; int y; };
struct { char x; int y; };
typedef struct A C;
typedef C* P;
typedef struct A * R;
typedef int S[10];
typedef int T[5];
typedef int Age;
typedef int (*F)(int);
typedef Age (*G)(Age);
struct A and C
struct A and struct B
struct B and C
struct A and struct { char x; int y; },
   P and R
   S and T
   int and Age
   P and G
```
Type Equivalence in Java

- `typedef`: makes it less complicated
- `class/interface`: a new type (name equivalence for class/interface names)
- `arrays`: structural equivalence

Example

```java
long y;
float x;
double c;
x = y/2+c;
```

- `y` is long, 2 is int, so 2 is promoted to long, so `y/2` is long
- `c` is double, `y/2` is long, so `y/2` is promoted to double, `y/2+c` is double
- `x` is float, `y/2+c` is double, then ...
  - C?
  - Java?

Type Checking

- **Type Checking**: Determine whether the program is correct based on data types
  - **Type Inference**: Infers the types of expressions
  - **Type Equivalence**: Are two types the same?
  - **Type Compatibility**: Relaxing exact type equivalence under certain circumstances

Example: C

```c
struct RecA {int i; double r;};
int p(struct (int i;double r;) x)
{ ... }
int q(struct RecA x)
{ ... }

struct RecA a;
int b;

b = p(a);  // error
b = q(a);
```
Type Conversion

- Can we just use code to designate conversion?
  - No: automatic/implicit conversion
  - Yes: manual/explicit conversion
- Has the data representation changed?
  - No, just the type
  - Yes, both the type and the value changed

Example: Java

- Implicit conversion:
  - Representation change (type promotion, e.g., int to double)
  - No representation change (upcasting)
- Explicit conversion:
  - Representation change: double x = 1.5; int y = (int)x
  - No representation change (downcasting)

Casting in Java

class A {public int x;}
class SubA extends A { public int y;}
A a1 = new A();
A a2 = new A();
SubA suba = new SubA();

a1 = suba;  // OK (upcasting)
suba = (SubA) a1;  // OK (downcasting)
suba = a2;  // compilation error
suba = (SubA) a2;  // compiles OK, runtime error
a1.y;  // compilation error
if (a1 instanceof SubA) { ((SubA) a1).y; }  // OK

Polymorphic Type Checking

- An extension of type inference to determine the types of names in a declaration without explicitly giving those types
  - eg, what is the type of A[i]+i if we have not declared A and i?
- Also known as Hindley-Milner type checking
- Method: use type variables for the unknown types
  - A: α
  - i: β
- Type variables correspond to unknown types
  - They can be instantiated to concrete types or other type variables
- Type unification:
  - Any type variable unifies with any type expression
  - Any two base types unify if they are the same type
  - Any two type constructors unify if they are of the same type and their corresponding components unify (recursively)
  - Must detect and reject cyclic bindings using occur-checks
### Parametric Polymorphism

- A kind of function overloading
  - Uses type variables (denoted by Greek letters $\alpha$, $\beta$, ...)
  - e.g., list append (++) has type $[\alpha] \to [\alpha] \to [\alpha]$
  - $[1,2,3]++[4,5] = [1,2,3,4,5]$
  - $["a", "b"]++["c"] = ["a", "b", "c"]$
- It means that ++ can be applied to multiple types $\alpha$
  - For $\alpha$=Int, $[\text{Int}] \to [\text{Int}] \to [\text{Int}]$
  - For $\alpha$=String, $[\text{String}] \to [\text{String}] \to [\text{String}]$
  - For $\alpha$=[Int], $[[\text{Int}]] \to [[\text{Int}]] \to [[\text{Int}]]$
    - e.g., $[[1,2],[3]]++[[4]] = [[1,2],[3],[4]]$
- **Ad-hoc polymorphism**: explicit overloading, where you define all instances of the overloading functions
  - 1+2, 2.3+4, 4.6+7.8

### Polymorphic type inference

- Must infer a principal type for each expression
  - *principal type*: a most general type that makes the expression type-correct
- It is the type of the expression specialized based on the type variable instantiations
- Polymorphic type inference is undecidable for unrestricted polymorphic functions
  - *Let-bound polymorphism* restricts the introduction of type variables
  - Makes polymorphic type inference decidable (but still exponential)
  - Used by ML and Haskell