subset

- 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 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

- $P(U) = \{U' | U' \subseteq U\}$
- Example: Pascal
  set of <ordinal type>
  - var S: set of 1..10;
  - var S: 1..10;
  What’s the difference?

- The order isn’t significant though unordered, values are distinct

set of in 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.
What are these?
• T := S*T;
• If T = [1, 3, 5] then ..;
• x = 3; if x in S then ..;
• If S<=T then ..;

• $\cap$ (intersection)
• $\cup$ (union)
• $\setminus$ (set difference)
• $\subseteq$ (subset)
• $\supseteq$ (superset)

implementations

type
  students = (Alice, Bob, Charlie, Dave, Eve, ...);
var
  CEE3302, CEE3310, whotookboth: set of students;
CEE3302 := { Alice, Bob, Dave };  // Alice, Charlie, Dave, Eve }
CEE3310 := CEE3302 $\cup$ CEE3310;
whotookboth := CEE3302 $\cap$ CEE3310;

Alice Bob Charlie Dave Eve
CEE3302 1 1 0 1 0 ...  // Alice, Charlie, Dave, Eve }
CEE3310 1 0 1 1 1 ...  // Alice, Charlie, Dave, Eve }
whotookboth 1 0 0 1 0 ...  // Alice, Dave

How about other operations? (membership, intersection, union, count, subsumption, ...)}
bit operations for set
Implementation can be very efficient
• bit-and (&), bit-or (|), bit-negation (~), bit-xor
• Hardware support
• Encoding schemes

Alternatives?
• Having Set type is convenient and efficient.
• If not present, can we use other types as alternatives?

Array and Functions
\[ f: U \rightarrow V \]
index type component type
• [0, ...] (C/C++/Java)
• Ordinal type (Ada/Pascal)

C/C++
Allocated on stack, size statically specified.
typedef int TenIntArray [10];
typedef int IntArray [];
TenIntArray x;
int y[5];
int z[1]={1,2,3,4};
IntArray w=(1,2);
IntArray w; //Illegal
int n = ... //from user input
int a[n]; //Illegal

Java
Allocated on heap, size dynamically specified;
Size can be obtained by length
int n = ... //From user input
int [] x = new int [n];
System.out.println(x.length);

Ada
size dynamically specified;
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;
Four Categories of Arrays

Based on subscript binding and location binding time

- **Static** - range of subscripts (static), location (static)
  - e.g. FORTRAN 77
  - Advantage: execution efficiency
- **Fixed stack dynamic** - range of subscripts (static), location (dynamic)
  - e.g. Pascal, C, C++
  - Advantage: space efficiency

Multi-dimensional arrays

- **C/C++**
  ```c
  int x[10][20];
  ```
- **Java**
  ```java
  int [][] x = new int [10][20];
  ```
- **Ada**
  ```ada
  type Matrix_Type is array(1..10, -10..10) of integer;
  x(i,j);
  ```

Array Operations

- **Access individual elements**
  - The whole array
    - Pascal: A := B
    - FORTRAN: A+B
    - Ada: Array concatenation A&B
    - FORTRAN 90: matrix multiplication, transpose

FORTRAN 90

```fortran
INTEGER, ALLOCATABLE, ARRAY :, :: : MAT
(Declares MAT to be a dynamic 2-dim array)
ALLOCATE (MAT (10, NUMBER_OF_COLS))
(Allocates MAT to have 10 rows and NUMBER_OF_COLS columns)
DEALLOCATE MAT
(Deallocates MAT's storage)
```
Functions

- C:
  ```c
  int incfn(int x) { return x+1; }
  typedef int (*IntFunc)(int); // why *?
  IntFunc inc = incfn;
  ```

- ML:
  ```ml
  val inc: IntFunc = fn x => x+1;
  ```

Vector, List, Sequence

Functional languages:

- Vectors: like arrays, more flexibility, especially dynamic resizability.
- Lists: like vectors, can only be accessed by counting down from the first element.
- Sequences: typically (potentially) infinite.

Pointer

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 not fixed (void *))
- Dynamic storage management
- Recursive data structures
  - E.g., linked list
    ```c
    struct CharListNode
    { char data;
      struct CharListNode* next;
    };
    typedef struct CharListNode* CharList;
    ```

Problems with Pointers

- Alias (with side-effect)
  ```c
  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);
  ```

- Dangling pointers (dangerous)
  ```c
  int *a, *b;
  a = (int *) malloc(sizeof(int));
  b=a;
  free(a);
  *b=2;
  printf("%d\n", *b);
  ```

- Garbages (waste of memory)
  ```c
  memory leakage
  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:** Infer the type of an expression, whose data type is not given explicitly.
    - e.g., `x/y`
  - **Type Equivalence:** Compare two types, decide 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

**Weakly-typed and untyped languages**

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

- **Untyped (dynamically typed) languages:** scheme, smalltalk, perl
  - Doesn't necessarily result in data errors.
  - All type checking performed at execution time.
  - May produce runtime errors too frequently.

**Strongly-Typed Languages**

- **Strongly-typed:** Ada, ML, Haskell, Java, Pascal
  - Most data type errors detected at translation time
  - A few checked during execution and runtime error reported (e.g., subscript out of array bounds).

- **Pros:**
  - No data-corrupting errors can occur during execution. (I.e., no unsafe program can cause data errors.)
  - Efficiency (in translation and execution.)
  - Security/reliability

- **Cons:**
  - May reject safe programs (i.e., legal programs is a subset of safe programs.)
  - Burden on programs, may often need to provide explicit type information.

**Security vs. flexibility**

- **Strongly-typed:**
  - No data errors caused by unsafe programs.
  - Maximum restrictiveness, static type checking, illegal safe programs, large amount of type information supplied by programmers.

- **Untyped:**
  - Runtime errors, no data-corruptions. Legal unsafe programs.
  - reduce the amount of type information the programmer must supply.

**Security vs. flexibility**

- **Strongly-typed:**
  - A type system tries to maximize both flexibility and security, where flexibility means: reduce the number of safe illegal programs & reduce the amount of type information the programmer must supply.

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