

CSC 324: Principles of Programming Languages

Type Systems

READING:

Sebesta 4.5-7; 5.4, 5.5, 5.7

©Suzanne Stevenson 2001

1

Data Types

- Enable programmers to think in terms of modelling reality with different kinds of values
 - Program semantics are embedded in types
- Enable additional compile- and run-time checks beyond syntax checking
 - variable declarations
 - operator/operand compatibility
- Determine layout and allocation of data

3

Data Types

Representation: organization of values that corresponds to some meaningful entity.

Type: set of values + valid operations on them.

E.g.,

Integers: + - * *div* < ≤ = ≥ > ...

Booleans: ∧ ∨ ¬ ...

Sets: Union, Intersection, Difference...

Strings: Concatenate, Reverse ...

2

Type Systems

- Primitive (elementary) types
 - C: int, float, char, double, ...
 - Pascal: char, integer, real, boolean, ...
 - Java: char, byte, int, float, boolean, ...
- Constructors for structured types
 - C: *, &, [], struct, ...
 - Pascal: ^, [], record, ...
 - Java: [], class, interface, ...

NOTE:

Either can be **built-in** or **user-defined**.

4

Data Types in C

- Primitives: char, int, float, double
no Boolean—any nonzero value is true
- Aggregates: arrays, structures

```
char a[10], b[2][10];
```

```
struct rectangle {  
    struct point p1;  
    struct point p2;  
}
```

- Enumerations: collection of sequenced values
- Pointers:
 - &i address of i
 - *p dereferenced value of p
 - p+1 pointer arithmetic

```
int *p, i;  
p = &i;  
*p = *p + 1;
```

5

Data Types in Java

- Primitives: char, byte, short, int, long, float, double, boolean.
- Aggregates: arrays, classes, interfaces

```
int[] a = { 1, 2, 3 };  
float[][] b = new float[3][4];
```

```
class Rectangle {  
    public Point p1;  
    public Point p2;  
}
```

```
interface Queue {  
    int MaxSize = 12;  
    public void enqueue(Object o);  
    public Object dequeue();  
    public int size();  
}
```

- References:
Every non-primitive variable is a reference.

6

Determining Types

Explicit Typing:

- Function and variable types are determined by declarations
- Type (usually) invariant throughout execution
- Supports static type determination
E.g., Pascal, Algol, C, C++, Java

7

Determining Types

Implicit Typing:

- Function and variable types are determined by use
- Entails dynamic type determination
E.g., Prolog, Lisp, Smalltalk

Mixture:

- Implicit by default, but allows explicit declarations
E.g., Fortran, Miranda, Haskell, ML

8

Rules for Determining Types of Expressions

General Rule: If f has type $S \rightarrow T$, and x has type S then $f(x)$ has type T :

type of `3 div 2` is `int`

type of `round(3.5)` is `int`

Special Rules: Needed to handle automatic type conversion:

type of `2 + 3.3` is `float`

9

Type Checking

Type Error: Attempt to apply a function of type $S \rightarrow T$ to an argument not of type S .

I.e., is each operator supplied with the correct type of arguments?

reject `round('Nancy')`

reject `3.5 div 2.5`

10

Type Checking and Type Safety

Type-Safe Program: A program that executes without type errors on all possible inputs

Static or compile-time checking: Use declaration information or static usage to determine type.

Dynamic or run-time checking: At execution, check type of objects before performing operations on them (use type tags to record types of values).

Strongly Typed: Programming language definition forces all type-checked programs to be type-safe

11

Type Checking Example

```
int a[10];
int x;
int *p;

a[0] = 1;
a[12] = 100;

scanf("%d", &x);
a[x] = 100;

*p = 8;
a[*p] = i;
```

12

Difficulties of Static Type Checking

Static type checking is not possible when the type of an expression depends not only on the **type** of each sub-expression, but also on the **values** of some sub-expressions; e.g.:

- Accessing elements of an array with a variable
- Taking successors of enumeration types
- Reading from the outside world
- Dereferencing untyped pointers

13

Dynamic Type Checking

Insert ops into compiled code to detect impending errors at run-time.

Problems:

- inefficiency
- undiscovered errors

14

Type Checking Design Principles

- static or dynamic checking (or a combination)
- strong or weak
- expressiveness (how large is the set of possible safe programs?)

15

Type Equivalence

In a type-safe program, assignment must not violate typing.

When are two variables of the same type, and can be assigned to each other?

```
m, n: array [0..9] of integer;
o: array [0..9] of integer;
p: array [1..10] of integer;
type intarray = array [0..9] of integer;
type myarray = intarray;
q: intarray;
r: intarray;
s: myarray;
```

16

Conventions for Type Equivalence

Name Equivalence:

Variables have the same type if they are declared using the same type name.

E.g., `intarray` is not equivalent to `myarray`;
`array[0..9]` of `integer` is not equivalent to
`array[0..9]` of `integer`.

17

Conventions for Type Equivalence

Declaration Equivalence:

Variables have the same type if they have the same type names or type names declared to be equivalent.

E.g., `intarray` is equivalent to `myarray`;
but `array[0..9]` of `integer` is not equivalent to
`array[0..9]` of `integer`

18

Conventions for Type Equivalence

Structural Equivalence:

Variables have the same type if the types have the same name or the same form—i.e., they are formed by applying the same type constructor to equivalent types (with user-defined type names replaced by their definitions).

E.g., `array[0..9]` of `integer` is equivalent to
`array[0..9]` of `integer`;
after type `i = integer`, `array[0..9]` of `i`
is equivalent to `array[0..9]` of `integer`.

19

Anonymous (Unnamed) Types

How are unnamed types, such as `array[0..9]` of `integer`, treated according to name-based type equivalence conventions (i.e., Name or Declaration Equivalence)?

Given:

```
m, n: array [0..9] of integer;
```

are `m` and `n` equivalent?

20

Pros and Cons of Name-Based Equivalence

Advantages:

- simple (just compare names)
- easy to apply consistently
- efficient

Disadvantages:

- very restrictive
- “surprising” incompatibilities
- unnamed types may not be very useful

21

Pros and Cons of Structural Equivalence

Advantages:

- flexible
- “intuitive”

Disadvantages:

- too unrestrictive (surprising **compatibilities**)
- complicated to formulate; details often left to implementation
- expensive

22

Type Conversion

Implicit Coercion

- Simple type conversions inserted by compiler
- Should be relatively safe
- User defined types are not implicitly coerced

Explicit Conversion Operations

- Type conversions specified by programmer
- Can be used to violate typing

23

Implicit Coercion and Overloading

Overloading: Ops can have different meanings in different type contexts.

Implicit Coercion: Type conversions operations inserted by the compiler.

```
x = 2 + 3.0
// Coerce 2 to 2.0 before addition. (Safe)
```

```
char a, b, c;
c = a + b;
// Coerce a and b to int. (Safe)
// Coerce a+b to char. (Not safe)
```

24

Explicit Type Conversion Invoked by Programmer

```
int truncate(double r)
// Converts a real to an int

int atoi(char *str)
// Converts a string to an int

(double) n
// Converts many types to long real
// An example of a "cast"

a[*p] = (int) a;
```

25

Type Hierarchies

Implicit conversions imply a type hierarchy.

Example: $\text{int} \rightarrow \text{float} \rightarrow \text{double} \rightarrow \text{complex}$

Widening:

- Converts a datum of type S to a higher type T
- Example: Convert an `int` to a `float`
- Safe: Will not lose information

26

Type Hierarchies

Narrowing:

- Converts a datum of type T to a lower type S
- Example: Convert a `float` to an `int`
- Unsafe: May lose information

27

Extending the Built-in Types

User-defined types enable the programmer to construct custom types that correspond more closely to the entities that need to be modelled.

- Enumerations
- Subranges
- Arrays
- Structures
- ...

28

Enumeration Types

- Ordered sequence of literal values
- Operators: assignment and comparison (w.r.t. defined order)

E.g., in Ada:

```
type class is ( frosh, soph, junior, senior );  
student_class : class;  
  
junior < senior  
  
for student_class := frosh .. senior do ...  
  
college : array [ frosh .. senior ] of integer;
```

29

Subtypes

A type *S* is a subtype of type *T* if the set of possible values for *S* is a subset of the set of possible values for *T*.

A subtype *S* of type *T* has the property that a value of *S* can be used wherever values of *T* are valid; thus:

- all operations valid on *T* values are valid on *S* values,
- but not necessarily vice versa.

E.g., `int` is a subtype of `float`

30

Subranges

```
subtype day is integer range 1..31;  
  
subtype year is integer range 1900..2000;  
  
g, m, f: day;  
  
m := 2; f := 31;  
  
g := m * f;    -- type error
```

31

Problems with Enumeration Types

```
type class is ( frosh, soph, junior, senior ) ;  
  
type transfer is ( soph, junior ) ;  
  
student1 : class;  
  
student2 : transfer;  
  
student1 := succ(junior);  
  
student2 := succ(junior);
```

32

Arrays

An array is a *homogeneous aggregate* type, where the underlying type elements are accessed via subscripts (random access).

Design choices:

1. Subscript syntax, e.g., [] vs ()
2. Allowable element types
3. Allowable subscript types
4. Bounds part of type or not
5. Allowable dimensionality
6. Compile-time or run-time bounds
7. Bounds checking
8. Initialization choices
9. Built-in operations

33

Arrays in C and Java

Design issue	C	Java
Syntax	[]	[]
Elements	anything	primitive types and references
Subscripts	integral types	integral types
Bounds in type	no	no
Dimensions	any	any
Bounds set at	compile/run time	run time
Bounds checking	no	yes
Initialization	flexible	flexible
Built-in ops	limited	rich

34

Structures

A structure, or record, is a *heterogeneous aggregate* type, where the component elements are accessed via field names (also random access).

Design choices:

1. Field selector syntax
2. Allowable field types
3. Name of fields part of type or not
4. Order of fields part of type or not
5. Initialization choices
6. Built-in operations

35

User-defined Stack Type in C++

```
#include <stream.h>

typedef int elt;
#define MAX 20
#define EMPTY -1

typedef struct {elt s[MAX]; int top; } stack;

stack * create() {
    stack * newstack = new stack;
    newstack->top = EMPTY;
    return (newstack);
}

void push(stack* stk,elt data)
    {stk->s[++stk->top]=data;}
void pop(stack* stk) {stk->top--;}
elt peek(stack* stk) {return (stk->s[stk->top]);}

int main() { /**** using the stack ****/
    stack *x;
    x = create();
    push(x,2); push(x,3);
    cout <<peek(x) << "\n";}
```

36

Problems with Data Types

- Implementation of the type can be seen (E.g., the array inside the stack)
- “Users” of the type can change its value arbitrarily
E.g., `x->s[5] = 10;`
 - doesn't respect push/top access pattern
 - doesn't respect `elt` typedef
- “Users” of the data type can write operations that create inconsistent states (E.g., adding an entry without changing the top index)
- “Users” cannot extend the set of operations in a reliably safe fashion

37

Abstract Data Types (ADTs)

User *may only* manipulate objects of the type through use of provided functions *without* knowing internal representation

- Encapsulation: may only use provided functions
- Information hiding: cannot see internal representation

39

Data Abstraction

Specification rather than implementation:

- Define *behavior* of data type (through interface functions)
- Hide *implementation* of data type
- \Rightarrow hide details irrelevant to the use of the data type

38

Advantages of Abstract Data Types

- Easier to use: as if only type names and function headers were visible
- Safety through access control
 - User can't make inconsistent states
 - User can't make assumptions about data representation
- Designer of ADT can modify implementation without affecting users
- Encourages modularity in programs, facilitating larger, more complex systems

40

Designing an Abstract Data Type

1. Specify interface
2. Identify and maintain invariants

Example: bounded stack

Interface:

- Stack of some kind of element `elt`
- `create` makes a new, empty stack
- `push` pushes new element on stack;
cannot push onto full stack
- `pop` removes an element; cannot pop
from empty stack
- `peek` returns the top element on stack
- `is_empty` determines if the stack is empty
- `is_full` determines if the stack is empty

41

Invariants:

- `peek(push(S, e)) = e`
- `pop(push(S, e)) = S`
- `is_empty(create())`
- `not is_empty(push(S, e))`
- `not is_full(pop(S))`

42

Java Interfaces and ADTs

```
public interface Stack {
    public void push(Object o);
    public Object pop();
    public boolean isEmpty();
    ...
}
public class ArrayStack implements Stack {
    private int EMPTY = 0;
    private int MAX = 100;
    private Object[] space;
    private int top;
    public ArrayStack() {
        space = new Object[MAX];
        top = EMPTY;
    }
    public void push(Object o) {
        if ( top == MAX ) throw new Error("stack overflow");
        space[top++] = o;
    }
    public Object pop() {
        if ( top == EMPTY ) throw new Error("stack underflow");
        return space[--top];
    }
    public boolean isEmpty() {
        return top == EMPTY;
    }
    ...
}
```

43

Parameterized Abstract Data Types

- Stack operations independent of element type
- Want to use same stack definition for stack of different kinds of elements

⇒ **Parameterized** data type

44

Historic Progression of Data Types

1. User-defined types
 - Can define arbitrary operations
 - Transparent: whole structure of type visible
 - Cannot control access to structure
2. Abstract data types
 - Encapsulation, information hiding
 - Access restricted to well-defined interface functions
 - Opaque: hides data representation
3. Object-orientation
 - Inheritance
 - Code re-use
 - Polymorphic behavior

Parameterized ADT in Ada

```
generic
  max: natural;
  type Item is private;
  -- Item is the base type of the stack

package STACKS is
  type Stack is private;
  procedure Push(S: in out Stack;
                Elt: in Item);
  function Pop(S: in out Stack)
    returns Item;
  function Is_empty(S: in Stack)
    returns boolean;
end;
declare package My_stack is
  new Stack(100, real); -- stack of reals
```

45

46

Summary of Type Systems

- Data types
- Type checking
- Type safety
- Type equality
- Type conversion
- Widening and narrowing
- Enumeration types
- Subtypes
- Arrays
- Structures
- Abstract data types

47