Types, Part II

Programming Languages
CS 214

Review

Recall: A type consists of ___data and operations___.

The set constructors:

- *product*: A×B× … ×N is the basis for ___aggregates___;
- *function*: (A) → B is the basis for ___operations___; and
- *Kleene closure*: A* is the basis for ________.

These three provide a formal way to construct types:

→ Use the *product* and *Kleene closure* to represent the ____;

→ Use the *function* constructor to represent the __________ on the type.

Declarations

One purpose of types is to permit objects to be *declared*.

– A declaration ________________ of an object (for the compiler).

– Example: If an object's value may vary, it is a *variable*; otherwise it is a *constant*.

- Issue: Where may declarations occur?
  - – Many languages restrict the location of declarations:

    <ada-program> ::= procedure identifier ; <declaration-section> <block> ;

    C++ and Lisp unusual in allowing declarations "anywhere"...
  - – In Smalltalk they must appear near the start of a method or block:

    <smalltalk-block> ::= [<parameters> '|' <identifiers> '|' <expressions>]
  - – Lisp declarations are functions/expressions, and so are permitted anywhere an expression may occur.

Constant Declarations

- Issue: How are constants distinguished from variables?
  - Most imperative languages use a ______________________:

<ada-const-dec> ::= identifier : constant <type> := <expression> ;

```
PI : constant real := 3.1459;
Mass, Energy : real;
```

  - C++ is similar, but uses the keyword *const*, and in a prefix form.

```
const double PI = 3.1459;
double mass, energy;
```

  - Java is similar to C++, but uses the keyword *final*.

```
final double PI = 3.1459;
double mass, energy;
```

  - Lisp constants are declared using the *defconst* function.

```
(defconst PI 3.14159)
(defvar mass 0.0)
```

Imperative Issue

- Fortran-family languages (e.g., Fortran, C/C++ and Java) declare variables using a form like this: <type> <id-list> ; while Algol-family languages (e.g., Ada, Pascal, Modula-2) use a form like this: <id-list> : <type> ;

Which approach is preferable?

```
double mass, energy;
```

1. The compiler ______________

____

2. Each id in the list can be

___________________________:

```
double mass = 1.0,
       energy = 0.0;
```

```
mass, energy : real;
```

1. The compiler doesn’t ‘know’ the type as it processes the ids.

2. Each id must be initialized elsewhere (or else all ids are limited to the same value):

```
mass, energy : real := 0.0;
```

Fundamental Type Operations

Operation	C++	Ada	Modula-2	Smalltalk	Lisp
+, -, *	+,-,*	+,-,*	+,-,*	+,-,*	+,-,*
______________	/	/	/	/	/
______________	/	/	DIV	//	/
______________	%	mod	MOD	\\	mod
______________		rem		rem:	rem
______________		**		raisedTo:	expt
______________		and		&	
______________	&&	and then	AND	and:	and
______________		or		|	
______________	||	or else	OR	or:	or
______________	!	not	NOT	not	not
______________	==	=	=	=, ==	=, eq, equal
______________	!=	/=	#, <>	~=, ~~	/=
______________			uniformly <,>,<=,>=		

Short-Circuit Operators

Logical-and, *logical-or* operators evaluate both operands.

By contrast:

- A *short-circuit-or* operator only evaluates its second operand if its first operand is _________; and
- A *short-circuit-and* operator only evaluates its second operand if its first operand is _________.

Short-circuit behavior is ______________________, and can be exploited in certain situations:

```
while (ptr != NULL && ptr->value != searchVal)
  ptr = ptr->next;
```

- If && were a logical-and instead of short-circuit-and, this condition would ___________ when *searchVal* is not in the list

Modeling Real-World Values

Suppose we want to model the seven "ROY G BIV" colors.

One approach:

```
const int RED=0, ORANGE=1, YELLOW=2, GREEN=3,
          BLUE=4, INDIGO=5, VIOLET=6;
int aColor = BLUE;
```

This approach requires ____________ to map colors to integers.

Instead:

```
enum Color { RED, ORANGE, YELLOW, GREEN,
             BLUE, INDIGO, VIOLET } ;
Color aColor = BLUE;
```

Most imperative languages support such *enumerations*…

Ada:

```
type Color = ( RED, ORANGE, YELLOW, GREEN,
               BLUE, INDIGO, VIOLET ) ;
aColor : Color := BLUE;
```

An enumeration is a type ______________________________.

Enumerations: Compiler Side

An enumeration's values must be valid *identifiers*:

<enumeration-type> ::= enum identifier { <id-list> } ;

and the compiler treats a declaration:

```
enum NewType { id0, id1, id2, …, idN-1 };
```

as being (approximately) equivalent to:

```
const int id0 = 0, id1 = 1, id2 = 2, …, idN-1 = N-1;
```

Thus, after processing

```
enum Color { RED, ORANGE, YELLOW, GREEN,
             BLUE, INDIGO, VIOLET };
```

so far as the compiler is concerned:

```
RED == _ && ORANGE == _ && YELLOW == _ && … && VIOLET == _
```

Enumerations: User Side

Enumerations thus provide an automatic means of mapping:

(identifier) $\rightarrow$ int

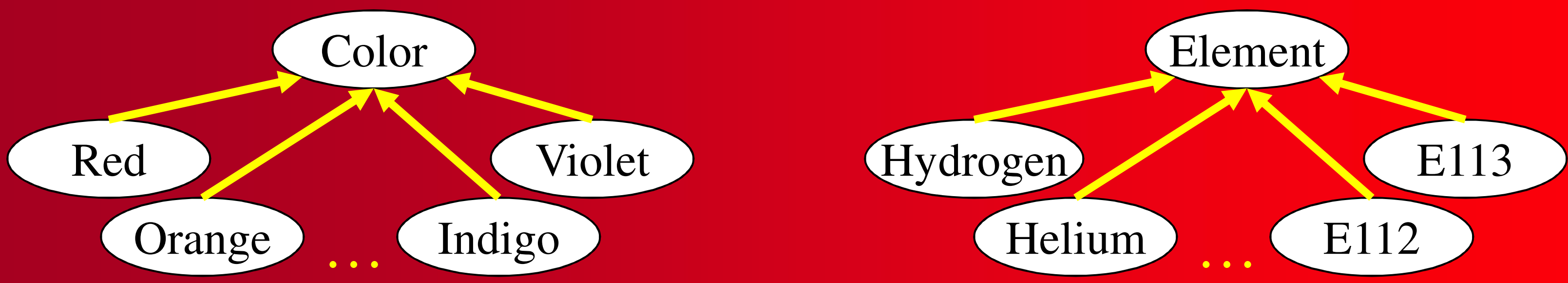
whose chief benefit is ______________________:

```
enum ElementName { HYDROGEN,
                   HELIUM,
                    …
                 };
ElementName anElement;
// ...
switch (anElement) {
  case HYDROGEN:
       atomicNumber = 1; break;
  case HELIUM:
       atomicNumber = 2; break;
  …
}
```

Enumerations allow real-world 'values' to be represented using ________________, instead of (arbitrary) integer codes.

Enumerations and OO

OO purists replace enums with class hierarchies:



This permits the creation of real-world _________:

```
// Smalltalk
aColor := new Blue.
```

```
// Smalltalk
anElement := new Helium.
```

as opposed to real-world *values* provided by an enumeration.

For this reason, "pure" OO languages like Smalltalk don't provide an enumeration mechanism.

Subranges

Many imperative languages let us declare a *subrange*: a type whose values are ______________________.

```
// Ada
subtype TestScore is Integer range 0..100;
subtype CapitalLetter is Character range 'A'..'Z';

type DaysOfWeek is (Sunday, Monday, Tuesday, Wednesday,
                    Thursday, Friday, Saturday);
subtype WeekDay  is DaysOfWeek range Monday..Friday;
```

If a subrange variable is declared:

```
WeekDay today;
```

and assigned an invalid value:

```
today := Saturday;
```

then an exception occurs that, if not caught, halts the system.

____________ is an essential feature for *life-critical systems*…

Sequence Types

There are two common structures for storing sequences:

- The _________ (and/or _________) that stores values in contiguous memory locations;
- The _______, that stores values anywhere there's room.

Arrays are _________________________________: any value in the array can be accessed in the same amount of time…

– The address of the value at index *i* can be computed in O(1):

  (arrayBaseAddress + (*i* - firstIndex) × ElementSize)

Once declared, the size of an *array* is usually _________.

A *vector* is an array-like structure whose size can _______.

Arrays

Most languages let us build arrays and/or vectors:

```
double anArray[8];                                  // C++
double * aVector = new double[n];
vector<double> anotherVector(n);
```

```
type ArrayType is array(1..8) of real;              -- Ada
anArray : ArrayType;
type Vector is array(integer range <>) of real;
type VectorAccess is access Vector;
aVector : VectorAccess;
begin
    aVector := new Vector(1..n);
```

```
(setq anArray (vector 0 1 2 3 4 5 6 7 8))  "Lisp"
```

```
| anArray aVector |                                 "Smalltalk"
anArray := Array new: 8.
aVector := OrderedCollection new: n.
```


Array Indexing

Arrays/vectors are *random-access indexed structures*: the value stored at index *i* can be accessed in O(1) time:

____

The default *firstIndex* is different in different languages:

- C/C++: 0
- Lisp: 0
- Ada: 1, can be programmer-specified
- Smalltalk: 1

At Issue: There is an *efficiency-vs-convenience tradeoff*:

- Accesses to 0-relative arrays require one fewer operation:

  (arrayBaseAddress + _____ × ElementSize)

  = arrayBaseAddress + ___ × ElementSize

- Programmer-specified index values can be very ____________:

```
type LetterCounter is array(CapitalLetter) of integer;
type DailySales is array(WeekDay) of real;
```


Array Access

An important array operation is to access the value at index *i*.

There are two different flavors to this operation:

– the *read* version _________ the value at index *i*;

– the *write* version lets us _________ the value at index *i*.

Most languages use the same syntax for both operations:

– C/C++:

```
oldValue = anArray[i];                // read
anArray[i] = newValue;                // write
```

– Ada:

```
oldValue = anArray(i);                -- read
anArray(i) = newValue;                -- write
```

– Lisp:

```
(setq oldValue (aref anArray i))      "read"
(setq (aref anArray i) newValue)      "write"
```

Other languages provide distinct operations for the two:

– Smalltalk:

```
oldValue := anArray at: i.            "read"
anArray at: i put: newValue.          "write"
```


Array Access (ii)

Although C++ uses the same syntax for both read and write, the operations are implemented as _________________________:

```
template<class Type>
class vector {
 public:
   …
   // returns a read-only reference to v[i]
   const Type& operator[](unsigned i) const; // read version

   // returns a writable reference to v[i]
   Type& operator[](unsigned i);             // write version
   …
};
```

The compiler links a given call to the proper function:

```
oldValue = anArray[i];            // linked to read version
anArray[i] = newValue;            // linked to write version
```


Aggregates: Records/Structs

Like an array, an __________ type can store multiple values; but (unlike an array) it can store values of ______________.

```
struct Student {    // C++
  int id;
  bool fTime;
  double gpa;
};
```

```
type Student is    -- Ada
  record
    id : integer;
    fTime: boolean;
    gpa : float;
  end record;
```

Smalltalk has no records/structs, as its classes can do anything a record/struct can do...

```
"Lisp"
(defstruct Student
   id fTime gpa)
```

Once we have an aggregate type, we can build variables:

```
// C++
Student stu;
```

```
-- Ada
stu : Student;
```

```
"Lisp"
(setq stu make-Student)
```


Record/Struct Projection

We can then ______________________ of the record/struct:

```
// C++
stu.id = 1234;
stu.fTime = true;
stu.gpa = 3.0;
```

```
-- Ada
stu.id:= 1234;
stu.fTime:=true;
stu.gpa:= 3.0;
```

```
"Lisp"
(setq (Student-id stu) 1234)
(setq (Student-fTime stu) t)
(setq (Student-gpa stu) 3.0)
```

Most languages use similar syntax to ___________________:

```
// C++
cout
 << stu.id;
 << stu.fTime
 << stu.gpa;
```

```
-- Ada
put(stu.id);
put(stu.fTime);
put(stu.gpa);
```

```
"Lisp"
(princ (Student-id stu))
(princ (Student-fTime stu))
(princ (Student-gpa stu))
```

These represent the ____________ operation in each language.

Pointers

Most languages permit a programmer to define variables that can store ______________: also known as *pointer variables*.

These can be used to build lists of linked nodes:

```
// C++
struct Node {
  SomeType value;
  Node * next;
};
```

```
-- Ada
type Node;
type NodePtr is access Node;
type Node is record
  value: SomeType;
  next: NodePtr;
end record;
```

Smalltalk and Java have no pointer types because ______ ______________ *is actually a pointer variable*.

Lisp variables are also pointers:

```
"Lisp"
(defstruct Node
   value next)
```


Dynamic Allocation & Deallocation

Each language supports ______________________________,
most commonly via the ______ operation:

```
// C++
Node * nPtr = new Node;
```

```
-- Ada
nPtr: NodePtr := new Node;
```

```
"Smalltalk"
| nPtr |
nPtr := Node new.
```

```
"Lisp"
(setq nPtr (make-Node))
```

Many languages also provide a ________________ operation:

```
// C++
delete nPtr;
```

Ada, Lisp, and Smalltalk instead provide a _________________ that eliminates memory leaks by automatically reclaiming unused memory (i.e., when no pointers to it exist).

Pointer Assignment and Copying

Each language provides a means of assigning pointer values:

```
// C++
Node * tempPtr;
tempPtr = nPtr;
```

```
-- Ada
tempPtr: NodePtr;
tempPtr := nPtr;
```

```
"Smalltalk"
| nPtr tempPtr|
…
tempPtr := nPtr.
```

```
"Lisp"
(setq tempPtr nPtr)
```

Most languages also allow copying of the object pointed to:

```
// C++
tempPtr = new Node;
*tempPtr = *nPtr;
```

```
-- Ada
tempPtr := new Node;
tempPtr.all := nPtr.all;
```

```
"Smalltalk"
tempPtr := nPtr copy;
```

```
"Lisp"
(setq tempPtr (copy-Node nPtr))
```

Other Pointer Operations

Languages provide a notation for ____________________:

```
// C++
nPtr = NULL;
```

```
-- Ada
nPtr:= null;
```

```
"Smalltalk"
nPtr := nil.
```

```
"Lisp"
(setq nPtr nil)
```

Most languages also permit the fields of an aggregate to be __________ via a pointer:

```
// C++
nPtr->value = 1234;
nPtr->next = new Node;
```

```
-- Ada
nPtr.value:= 1234;
nPtr.next:= new Node;
```

Smalltalk "fields" are accessed via accessor and mutator methods

```
"Lisp"
(setq (Node-value nPtr) 1234)
(setq (Node-next nPtr) (make-Node))
```


Type Systems

A *type system* is a set of rules by which a language associates ________ with _________________.

The system generates a *type-error* when its rules do not permit a type to be associated with an expression.

Example: Early Fortran versions had only integers and reals.

- Declarations not required: implicit typing of identifiers
  - Identifiers beginning with I-N are integers; all others are reals.
- Literals with decimal points are real; others are integers.
- Type System Rule: If E1 and E2 are expressions of the same type T, then E1+E2, E1-E2, E1*E2, and E1/E2 produce a result of type T.
  - I+N produces a value of type _________; X+Y produces a value of type _____.
  - Expressions like X+I (e.g., 0.5+1)or N-Y generate ______________.

Type System Formalism

If f is a function from $(S) \rightarrow T$, and $s \in S$, then $f(s) \in T$.

Ada defines:	$+(\text{int} \times \text{int}) \rightarrow \text{int}$
and	$+(\text{real} \times \text{real}) \rightarrow \text{real}$
but neither	$+(\text{real} \times \text{int}) \rightarrow \text{real}$ nor $+(\text{int} \times \text{real}) \rightarrow \text{real}$
so both	$2 + 3$ and $2.0 + 3.0$ are valid expressions;
but neither	$2.0 + 3$ nor $2 + 3.0$ are valid expressions.

Arithmetic expressions ______________________ cause type errors.

Ada's other arithmetic operators behave the same way.

Why would Ada's designers choose such a type system?

- Ada is designed for building __________________________ ...
- Ada's type system is perhaps ___________________ of any HLL.
- Ada compilers ________________ that slip by in other languages.

Coercion

Ada is unusual in rejecting mixed-type arithmetic expressions; its goal is to prevent ________________________________.

Most HLLs permit arithmetic types to be freely intermixed.

To prevent information loss,

such languages take an expression: $2 + 1.5$

"expand" the ___________operand: $2.0 + 1.5$

and then perform the ________ operation: $+(\text{real} \times \text{real})$

The automatic conversion of an operand's type to prevent rejection by the type system is called a _________________.

Some languages use the term ____________ to describe this; others describe it as ______________.

Overloading

Note: operators like +, -, *, ... are ________________. In $a + b$:

+ means "perform integer addition" if a and b are integers;

+ means "perform real addition" if a and b are reals.

___________ symbols have different meanings in different contexts.

Formally: For any function $f(D) \rightarrow R$:

– The set of all possible arguments D is the function's *domain*;

– The set of all possible results R is the function's *range*.

An overloaded function is __________________________________.

+, -, *, / are overloaded in most HLLs

To process such operations, the compiler must check the context (operand types) and find a definition with the matching domain.

A __________ occurs when no definition with that domain is found.

Type Checking

A type system enforces its rules by *type checking*:

- Analyzing the code, looking for type errors
- Only permitting programs without type errors to execute.
- A program with no type errors is described as ____________.

Type checking is accomplished at two levels:

1. ___________________: check for type-errors at ______________.
2. ______________________: check for type-errors at ___________.

Ada performs both static and dynamic checking, but the language is designed to maximize the number of errors that can be detected ______________ (i.e, by the compiler).

Static Checking Examples

a. In C++ expressions of the form: x % y

the compiler looks up the types of x and y
(in a data structure called the ____________________)
and rejects the expression if both are not of type int.

b. In C++ expressions of the form: sqrt(x)

the symbol table contains both the type *T* of argument x
and the domain-set *D* for which sqrt() is defined,
allowing the compiler to reject the expression if $T \notin D$.

- Original (K&R) C did not require that function prototypes contain parameter types, making it impossible for the compiler to type-check function calls (ANSI-C corrected this).

Dynamic Checking Examples

Dynamic checking is checking for errors only detectable at run-time by inserting checks before an expression's code.

Expression: x / y

```
// without dynamic checks
      mov x, R0
      div R0, y
```

```
-- with dynamic checking
      mov x, R0
      mov y, R1
      cmp R1, #0
      be  DivideByZero
      div R0, R1
```

A[i]

```
// without dynamic checks
      mov A, R0
      add R0, i
```

```
-- with dynamic checking
      mov A, R0
      mov i, R1
      cmp R1, firstIndex
      blt  IndexTooLow
      cmp R1, lastIndex
      bgt IndexTooHigh
      add R0, R1
```

Dynamic checking is ________________________________...

Type Strength

A language is ________________ if it has a strict type system.

A language is ________________ if it has a loose type system.

From this perspective, languages lie somewhere on a continuum, based on the the strength of their type system:

Lisp C (pre-ANSI) C++ Smalltalk Java Ada

weaker stronger

Fortran-I, -II Fortran-IV Fortran-77 Fortran-90

Language type systems have tended to ____________________ as they evolve through different versions.

– The importance of type-strength has increased as the systems being built have increased in ____________________________.

Type Compatibility

What determines if two types *T1* and *T2* are compatible (e.g., can *T1* arguments be passed to *T2* parameters)?

```
typedef int IntArray[32];
IntArray x, y;
// Are x, y compatible?
```

```
int x[32];
void f(int y[32]);
// Are x, y compatible?
```

```
struct Student {            struct Employee {
  int id;                      int id;
  string name;                 string name;
};                          };
Student stu;                Employee emp;
// Are stu, emp compatible?
```

```
struct Student {            struct Employee {
  int studentID;               int empID;
  string studentName;          string empName;
};                          };
Student stu;                Employee emp;
// Are stu, emp compatible?
```


Equivalence

Compatibility depends on whether a language views two types as *equivalent*.

There are two broad categories of equivalence:

– Languages that use ____________________ view two types as equivalent if they have ______________________________.

– Languages that use ____________________ view two types as equivalent if they are ______________________________.

To illustrate, suppose that we have these declarations:

```
struct Student {                    struct Employee {
  int studentID;                       int empID;
  string studentName;                  string empName;
};                                  };
Student stu;                        Employee emp;
// Are stu, emp equivalent?
```

Structural Equivalence (SE)

Structural equivalence relies on three "rules":

- SE1: *A type name is structurally equivalent to itself.*

```
Student stu1;
Student stu2;
```

→ Since their types ____________________,
stu1 and *stu2* are structurally equivalent.

- SE2: *Two types formed by applying the same constructor to SE types are structurally equivalent.*

```
Student stu;
Employee emp;
```

→ Since both are _________________________,
stu and *emp* are structurally equivalent.

- SE3: *If one type is an alias of another, the two types are structurally equivalent.*

```
typedef Student Transfer;
Student stu;
Transfer trans;
```

→ Since *Transfer* is __________ *Student*,
stu and *trans* are structurally equivalent.

Name Equivalence (NE)

There are different varieties of name equivalence:

- Pure NE: To be equivalent, types ______________________.

```
-- Ada uses pure name equivalence
type IntArray is array(1..32) of Integer;
type IntList is array(1..32) of Integer;

a1: IntArray;
a2: IntList;
a3: IntArray;
a4: array(1..32) of Integer;
a5: array(1..32) of Integer;
```

→ Since *a1* and *a3* are declared with ________________, they are equivalent.

→ If we declare: `procedure print(IntArray anArray);` then Ada's type system will only accept __________ as arguments.

→ *a2's* type has a name, but it is a different name from the others.

→ *a4* and *a5's* types have no name: __________________ in Ada.

Name Equivalence (ii)

- ______________: *A type name is equivalent to itself (pure NE), plus it can be declared equivalent to other type names.*

```
-- C++ uses transitive name equivalence
struct Student {                      struct Employee {
  int idNumber;                          int idNumber;
  string name;                           string name;
};                                    };
typedef Student Transfer;
Student stu;                          Employee emp;
Transfer trans;
```

→ *stu* and *trans* are compatible; *emp* is not compatible to either.

→ If we declare: `void print(Student aStudent);`
then the type system will only accept *stu* or *trans* as arguments, but will reject *emp* as an argument.

Which is better?

- Consider type-checking on aggregates:
  - Type-checking is much __________ under name equivalence, as the type-checker just has to do ____________________________.
  - Under structural equivalence, the type-checker must do ____ ______________________________________ (e.g., nested records??).
- Name equivalence encourages _________________:
  - NE encourages detail-hiding (ADT) by rejecting anonymous types:

```
procedure Put(seq: Sequence);
```

→ Any Sequence accepted.

```
procedure Put(seq: array(1..32) of Integer);
```

→ Nothing accepted.

  - SE discourages abstraction by accepting anonymous types:
    - SE may permit programs to be written faster (abstraction takes time).
    - Such programs may be harder to maintain; may be type-unsafe.

Summary

A type consists of ________________________________.

The set constructors: *product*, *function*, and *Kleene closure* provide a formal way to represent type construction, using *product* and *Kleene closure* to represent the _______, and *function* to represent the _____________ on the new type.

__________________ let us use *real-world values* for type data.

______________ constrain the values of existing data types.

__________________________ are sequences stored in *adjacent memory locations* that permit *O(1)* time access to any value.

_______ are sequences stored in *dynamically allocated nodes*, that require *O(n)* time (on average) to access a value.

Summary (ii)

_____________ store *multiple values of arbitrary types*.

____________ store *addresses*, permit us to build linked nodes.

A *type system* performs type-checking using _____________ *equivalence* or a version of _________ *equivalence*.

Type-checking may be:

- *Static*: done at ________________; and/or
- *Dynamic*: done at _______________.

The more type-checking a language requires, the __________ its type-system, and the fewer type-errors slip past.

______ has a very strong type-system.

