A Book Chapter

By Narendra Kumar Chahar

on Data Types

Index:

Chapter 2

Data Types

Specifications and Implementation of Elementary and Structured Data Types:

Basic differences among programming languages:

- types of data allowed
- types of operations available
- mechanisms for controlling the sequence of operations

Elementary data types: built upon the available hardware features

Structured data types: software simulated

- **Data objects, variables, and constants**

  1. 1. **Data object:**

     a run-time grouping of one or more pieces of data in a virtual computer.
     a location in memory with an assigned name in the actual computer.

     Types of data objects:

     - Programmer defined data objects - variables, arrays, constants, files, etc.
     - System defined data objects - set up for housekeeping during program execution, not directly accessible by the program. E.g. run-time storage stacks.

     Data value: a bit pattern that is recognized by the computer.

     Elementary data object: contains a data value that is manipulated as a unit.
     Data structure: a combination of data objects.

     Attributes: determine how the location may be used. Most important attribute - the data type.

     Attributes and Bindings

     - Type: determines the set of data values that the object may take and the applicable operations.

- Name: the binding of a name to a data object.
- Component: the binding of a data object to one or more data objects.
- Location: the storage location in memory assigned by the system.
- Value: the assignment of a bit pattern to a name.

Type, name and component are bound at translation, location is bound at loading, value is bound at execution

1. 2. **Data objects in programs**

In programs, data objects are represented as variables and constants

Variables: Data objects defined and named by the programmer explicitly.

Constants: a data object with a name that is permanently bound to a value for its lifetime.

- Literals: constants whose name is the written representation of their value.
- A programmer-defined constant: the name is chosen by the programmer in a definition of the data object.

1. 3. Persistence

Data objects are created and exist during the execution of the program. Some data objects exist only while the program is running. They are called transient data objects. Other data objects continue to exist after the program terminates, e.g. data files. They are called persistent data objects. In certain applications, e.g. transaction-based systems the data and the programs coexist practically indefinitely, and they need a mechanism to indicate that an object is persistent. Languages that provide such mechanisms are called persistent languages.

2. Data types

A data type is a class of data objects with a set of operations for creating and manipulating them.

Examples of elementary data types:
integer, real, character, Boolean, enumeration, pointer.

2. 1. Specification of elementary data types

- Attributes that distinguish data objects of that type Data type, name - invariant during the lifetime of the object

- stored in a descriptor and used during the program execution
- used only to determine the storage representation, not used explicitly during execution
- Values that data object of that type may have determined by the type of the object usually an ordered set, i.e. it has a least and a greatest value

- Operations that define the possible manipulations of data objects of that type.

    Primitive- specified as part of the language definition
    Programmer-defined (as subprograms, or class methods)

    An operation is defined by:

    - Domain - set of possible input arguments
    - Range - set of possible results
    - Action - how the result is produced

    The domain and the range are specified by the operation signature

    - the number, order, and data types of the arguments in the domain,
    - the number, order, and data type of the resulting range

        mathematical notation for the specification:

            op name: arg type x arg type x … x arg type ® result type

        The action is specified in the operation implementation

    Sources of ambiguity in the definition of programming language operations

    - Operations that are undefined for certain inputs.
    - Implicit arguments, e.g. use of global variables
    - Implicit results - the operation may modify its arguments

            (HW 01 - the value of a changed in x = a + b)

    - Self-modification - usually through change of local data between calls,
      i.e. random number generators change the seed.

Subtypes: a data type that is part of a larger class.
Examples: in C, C++ int, short, long and char are variations of integers.

The operations available to the larger class are available to the subtype.
This can be implemented using inheritance.

2. 2. Implementation of a data type

- **Storage representation**

  Influenced by the hardware Described in terms of:

  Size of the memory blocks required Layout of attributes and data values within the block

  Two methods to treat attributes:

  - determined by the compiler and not stored in descriptors during execution - C
  - stored in a descriptor as part of the data object at run time - LISP Prolog

- **Implementation of operations**

  - Directly as a hardware operation. E.g. integer addition
  - Subprogram/function, e.g. square root operation
  - In-line code. Instead of using a subprogram, the code is copied into the program at the point where the subprogram would have been invoked.

- **Declarations**

Declarations provide information about the name and type of data objects needed during program execution.

- Explicit – programmer defined
- Implicit – system defined

e.g. in FORTRAN - the first letter in the name of the variable determines the type
Perl - the variable is declared by assigning a value

$abc = 'a string' $abc is a string variable
$abc = 7 $abc is an integer variable

Operation declarations: prototypes of the functions or subroutines that are programmer-defined.

Examples:

declaration: float Sub(int, float)
signature: Sub: int x float --> float

Purpose of declaration

- Choice of storage representation
- Storage management

- Declaration determines the lifetime of a variable, and allowes for more efficient memory usage.
- Specifying polymorphic operations.

    Depending on the data types operations having same name may have different meaning, e.g. integer addition and float addition

    In most language +, -. *, / are overloaded
    Ada - aloows the programmer to overload subprograms
    ML - full polymorphism

Declarations provide for static type checking

- **Type checking and type conversion**

Type checking: checking that each operation executed by a program receives the proper number of arguments of the proper data types.

Static type checking is done at compilation.
Dynamic type checking is done at run-time.

Dynamic type checking – Perl and Prolog Implemented by storing a type tag in each data object.

Advantages: Flexibility
Disadvantages:

- Difficult to debug
- Type information must be kept during execution
- Software implementation required as most hardware does not provide support

Concern for static type checking affects language aspects:

Declarations, data-control structures, provisions for separate compilation of subprograms

Strong typing: all type errors can be statically checked

Type inference: implicit data types, used if the interpretation is unambiguous.
Used in ML

Type Conversion and Coercion

Explicit type conversion: routines to change from one data type to another.

Pascal: the function round - converts a real type into integer
C - cast, e.g. (int)X for float X converts the value of X to type integer

Coercion: implicit type conversion, performed by the system.

Pascal: + integer and real, integer is converted to real
Java - permits implicit coercions if the operation is widening
C++ - and explicit cast must be given.

Two opposite approaches to type coercions:

- No coercions, any type mismatch is considered an error : Pascal, Ada
- Coercions are the rule. Only if no conversion is possible, error is reported.

Advantages of coercions: free the programmer from some low level concerns,as adding real numbers and integers.

Disadvantages: may hide serious programming errors.

- **Assignment and Initialization**

Assignment- the basic operation for changing the binding of a value to a data object.

Two different ways to define the assignment operation:

- does not return a value
- returns the assigned value

The assignment operation can be defined using the concepts L-value and R-value

Location for an object is its L-value.Contents of that location is its R-value.

Consider executing: A = A + B;

1. Pick up contents of location A: R-value of A
2. Add contents of location B: R-value of B
3. Store result into address A: L-value of A.

For each named object, its position on the right-hand-side of the assignment operator (=) is a content-of access, and its position on the left-hand-side of the assignment operator is anaddress-of access.

- Address-of is an L-value
- contents-of is an R-value
- Value, by itself, generally means R-value

Initialization

Uninitialized data object - a data object has been created, but no value is assigned, i.e. only allocation of a block storage has been performed.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type. This automatic conversion is called coercion.
- a type error is the application of an operator to an operand of an inappropriate type.
- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- Def: A programming language is strongly typed if type errors are always detected.

Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
- FORTRAN 77 is not: parameters, **EQUIVALENCE**
- Pascal is not: variant records
- C and C++ are not: parameter type checking can be avoided; unions are not type checked.
- Ada is, almost (**UNCHECKED CONVERSION** is loophole)(Java is similar)
- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name.
- Easy to implement but highly restrictive:
- Subranges of integer types are not compatible with integer types
- Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Consider the problem of two structured types:
- Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [0..9])
- Are two enumeration types compatible if their components are spelled differently?
- With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float).

  Language examples:

- Pascal: usually structure, but in some cases name is used (formal parameters)
- C: structure, except for records
- Ada: restricted form of name
- Derived types allow types with the same structure to be different
- Anonymous types are all unique, even in:

A, B : array (1..10) of INTEGER:

Scope

- The scope of a variable is the range of statements over which it is visible
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Static scope
- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
- In Ada: **unit.name**
- In C++: **class_name::name**
- Blocks
- A method of creating static scopes inside program units--from ALGOL 60
- Examples:

  C and C++: **for (...)**

{

int index;

...

}

Ada: **declare LCL : FLOAT;**

begin

...

end

- Evaluation of Static Scoping
- Consider the example:

Assume MAIN calls A and B

A calls C and D

B calls A and E

Static Scope Example

Static Scope

- Suppose the spec is changed so that D must now access some data in B

  Solutions:

- Put D in B (but then C can no longer call it and D cannot access A's variables)
- Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals
- Dynamic Scope
- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

  Scope Example

MAIN

- declaration of x

SUB1

- declaration of x -

...

call SUB2

...

SUB2

...

- reference to x -

...

...

call SUB1

…

Scope Example

- Static scoping
- Reference to x is to MAIN's x
- Dynamic scoping
- Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
- Advantage: convenience
- Disadvantage: poor readability

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **static** variable in a C or C++ function.

References:

[1]. Swierstra, Wouter. "Data types à la carte." *Journal of functional programming* 18.4 (2008): 423-436.
[2]. Guttag, John V., and James J. Horning. "The algebraic specification of abstract data types." *Acta informatica* 10.1 (1978): 27-52.
[3]. Pavlidis, Paul, et al. "Learning gene functional classifications from multiple data types." *Journal of computational biology* 9.2 (2002): 401-411.
[4]. Bruce, Kim B. "Safe type checking in a statically-typed object-oriented programming language." *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993.
[5]. Cardelli, Luca. "Basic polymorphic typechecking." *Science of computer programming* 8.2 (1987): 147-172.
[6]. Mössenböck, Hanspeter, and Niklaus Wirth. "The programming language Oberon-2." *Structured Programming* 12.4 (1991): 179-196.