Comparative Programming Notes

Sean Moloney

January 2020

Contents

---

1 Data, Values and Types

A *value* is any entity that can be manipulated by a program. It can be evaluated, stored, passed as a parameter to a function or procedure and returned from a function.

1.1 Types and Operations

Most programming languages group *Values* into *Types*. A *Type* is a set of values. Hence, if we say that v is a value of type T, we are simply saying that $v \in T$.

We set a restriction on the kind of sets that can be used to form *Types*. Each operation associated with a *Type* must act uniformly when applied to all values of that *Type*.

Types are defined by the values the set contains and the operations of those values.

1.2 Primitive Types

A *Primitive Value* is one that cannot be decomposed into simpler values. A *Primitive Type* is a set of *Primitive Values*. Every programming language has a set of built-in primitive types, and some languages allow the user to define new primitive types.

The cardinality of a type T, denoted $\#T$, is the number of distinct values for that type, for example, $\#Boolean = 2$.

1.2.1 Built-ins

The most common built-ins are *Boolean*, *Character*, *Integer*, and *Floating*.

Not all languages have a distinct Boolean and Character class; For example, In C++, the *Boolean* type **bool** is just small numbers. Similarly, in C, C++, and Java, the *Character* type **char** is actually just small integers, meaning the character 'A' and the value 65 are the same.

Many languages have different sizes of *Integers*. They even have the same names, such as **short**, **int**, and **long** in Java, C and C++.

Some languages allow the programmer to define the ranges of integer and floating-point values to avoid portability issues between machines with different architectures.

1.2.2 Discrete Primitives

A *discrete primitive type* has a one-to-one mapping with the range of integers. in Ada, te types **Boolean**, **Character**, and **enumerated** types are all *discrete primitive types*, which can be very useful:

```
freq: array(Character) of Natural;

for ch in Character loop
    freq(ch) := 0;
end loop;
```

```
type Month is (jan, feb, mar, apr, may, jun,
               jul, aug, sep,oct, nov, dec);
length: array(Month) of Natural :=
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
for mth in Month loop
    put(length(mth));
end loop;
```

1.3 Composite Types

A *Composite Type* is a value made up of simpler values, meaning it is a data structure. A *Composite Type* is a set of *Composite Values*.

The variety of *Composite Types* among programming languages is vast but they can be grouped under the following categories:

- Cartesian Products such as tuples and records.
- Mappings such as arrays.
- Disjoint Unions such as algebraic types, discriminated records and objects.
- Recursive types such as lists and trees.

1.3.1 Cartesian Products

In a *Cartesian Product*, the values from several types are grouped into tuples.

The notation (x, y) denotes a pair whose first value is x and second is y.

The basic operations on pairs are:

- **Construction** of a pair of values
- **Selection** of either the first or second value

In C++ structures can be understood in terms of Cartesian Products.

```
enum Month {jan, feb, mar, apr, may, jun,
            jul, aug, sep, oct, nov, dec};

struct Date {
    Month m;
    byte d;
};
```

This structure has the values:

$$Date = Month * byte = jan, feb, .., nov, dec * 0, 1, ..., 255.$$

The cardinality of a cartesian product is:

$$\#(S * T) = \#S * \#T$$

If the cartesian product is homogeneous, meaning it is a tuple of components from the same set, it's cardinality is:

$$\#(S^n) = \#S * \#S * ... * \#S = (\#S)^n$$

1.3.2 Mappings

The concept of *Mapping* from one set to another set is very important in programming languages and underlies two important features in programming, arrays and functions.

The notation $m : S \to T$ represents a mapping m from set S to set T, meaning every value in S is mapped to value in T.

If m maps the value x in S to the value y in T, we write $y = m(x)$ and say that y is the image of x under m.

The basic operations on arrays are:

- **Construction** of an array from its elements
- **indexing** which selects a given element from an array based on its index

Function procedures, supported by some programming languages are also *mappings*. For example, in C++:

```
bool is_even (int i) {
    return (i % 2 == 0);
}
```

is a mapping $Integer \rightarrow Boolean$. Even if we change the implementation of the function, it is still the same mapping.

1.3.3 Disjoint Unions

In a *Disjoint Union* a value is selected from one of several sets. Let the notation $S + T$ represent the *Disjoint union* of sets S and T.
Each element of $S + T$ consists of a tag which identifies which original set the element came from and a variant which is the value from the original set.

$$left\, S + right\, T = \{left\, x | x \in S\} \cup \{right\, y | y \in T\}$$

Where the *tags* are irrelevant we can leave them out:

$$S + T = \{left\, x | x \in S\} \cup \{right\, y | y \in T\}$$

The cardinality of a *Disjoint Union* is:

$$\#(S + T) = \#S + \#T$$

The basic operations of *Disjoint unions* are:

- **Construction** by appropriately tagging each element from both sets
- **Tag test** to determine if a *variant* was from S or T
- **Projection** to recover the *variant* in S or the *variant* in T

Disjoint unions can be used to understand Haskell's algebraic types:

```
-- Declare algebraic type
data Number = Exact Integer | Inexact Float

-- Construction
let pi = Inexact 3.1416
in ...

-- Tag test and projection
rounded num =
    case num of
        Exact i => i
        Inexact i => round i
```

Ada's driscriminated records:

```
type Form is (point, circle, rectangle);
type Figure (f: Form) is
    record
        x, y: float
        case f is
            when point =>
                null;
            when circle =>
                r: Float;
            when rectangle =>
                w, h: Float;
            end case;
        end record;
```

the set of objects in a Java program:

```
class Point {
    float x, y;

    ... //methods
}

class Circle extends Point {
    float radius;

    ... //methods
}

class Rectangle extends Point {
    float width, height;

    ... //methods
}
```

and C's unions when declared inside structures.

```
enum Accuracy {exact, inexact};
struct Number {
    Accuracy acc;
    union {
        int i;  /* used when acc = exact */
        float f; /* used when acc = inexact */
    }   content;
};
```

1.4 Recursive types

A *Recursive Type* is a type that is defined in terms of itself. The typical *recursive types* that occur in programming languages are lists and strings.

1.4.1 Lists

A list is sequence of values. If all the elements are the same type, the list is *homogeneous*, otherwise it is *heterogeneous*. The number of elements in a list is called the *length*. A list with no elements is called an *empty list*.

The typical operations of a list are:

- **Construction**, add a new element as the head of the new list
- **Length**
- **Empty test**
- **Head selection**, select the list's first element.
- **Tail selection**, select any of the other elements.
- **Concatenation**, joining two lists

In some programming languages lists are called *sequences*. The equation for a finite lists of type T is:

$$T^* = Unit + (T * T^*)$$

For integer lists, the set of values is:

$$\begin{gathered}\{nil\} \\ \cup\{cons(i, nil)|i \in Integer\} \\ \cup\{cons(i, cons(j, nil))|i, j \in Integer\} \\ \cup\{cons(i, cons(j, cons(k, nil)))|i, j, k \in Integer\} \\ +...\end{gathered}$$

In imperative programming languages, like C, C++ and Ada, recursive types are implemented using pointers, but in functional and some object-oriented programming languages, they can be defined and implemented directly.

An example of lists in Java:

```
class IntList{
    public IntNode first;

    public IntList(IntNode first) {
        this.first = first;
    }
}

class IntNode {
    public int elem;
    public IntNode succ;

    public IntNode (int elem, IntNode succ) {
        this.elem = elem;
        this.succ = succ;
    }
}
```

Haskell has a built-in list type, but if it didn't we could define it as:

```
data IntList = Nil | Cons Int IntList
```

1.4.2 Strings

A *String* is a sequence of characters. The number of characters in a string is called the *length* fo the string. A string with no characters is called an *empty string*.

The typical string operations are:

- **Length**
- **Equality test**
- **Lexicographical comparison** to order two strings.
- **Character selection**
- **Substring selection**
- **Concatenation** to join two strings.

1.4.3 General Recursive Types

Some languages, such as Haskell, support general recursive types.
These are defined by the equation:

$$R = ... + (...R...R)$$

In general the cardinality of a recursive type is infinite, even though the elements of a recursive type are finite.
The following is an example of general recursive type, implementing binary trees in Haskell:

```
data BinIntTree = Empty
    | BinIntTree int BinIntTree BinIntTree
```

1.5 Type Systems

A *Type System* allows a programming language to group values into types. A *type system* prevents a programmer from perform stupid operations like multiplying a string by a boolean, which results in a *Type Error*
In a *statically typed* language, each variable and expression has a fixed type. This type is either defined explicitly, or is inferred, but before every operation the compiler can check that the operands have the correct type at compile time.
In a *Dynamically Typed* language, values are typed, but variables and expressions are not. Every time an operand is computed, it may result in a value of a different type. Therefore, operands can only be type checked at run time as they are computed but before the operation is performed.
Most languages are statically typed, but some, like python, prolog, lisp, perl, and smalltalk, are dynamically typed.
Statically typed languages are:

- More efficient
- More secure

whereas Dynamically typed languages are:

- More flexible and necessary in applications where the type of data is not known in advance.

Here's an example of Python's dynamic typing:

```
def respond(prompt):
    try:
        response = raw_input(prompt)
        return int(response)
    except ValueError:
        return response

m = respond("Enter month: ")
if m = "Jan":
    m = 1
elif m = "Dec":
    m = 12
elif not (isinstance(m, int) and 1<=m<=12):
    raise DataError, f"{m} is not a valid string or value"
```

1.5.1 Type Equivalence

In order to type check, we need to ask if the two operands T_1 and T_2 are equivalent. There are two commonly used definitions of *type equivalence*:

- Name equivalence
- Structural equivalence

T_1 is *name equivalent* to T_2, $T_1 \equiv T_2$ if they are defined in the same place.
For example:

```
struct Date {int d, m};
struct OtherDate {int d, m};
struct OtherDate yesterday;
struct Date today, tomorrow;
```

Here, today and tomorrow are name equivalent since they are defined on the same line but they are not equivalent to yesterday.

Java and Ada use name equivalence.

T_1 is structurally equivalent to T_2, $T_1 \equiv T_2$ if T_1 and T_2 consist of the same components.
For example:

```
struct Date {int d, m};
struct OtherDate {int d, m};
struct OtherDate yesterday;
struct Date today, tomorrow;
```

Here the variables today and tomorrow are structurally equivalent.

Algol-68, ML, C and Modula-3 all use structural equivalence.

1.6 Expressions

An *Expression* is a programming construct that evaluates a value.

A *Literal* denotes a fixed value of some type, for example, 2.5, false, and "Hello" denore a real number, a boolean, and a string.

1.6.1 Constructions

A *construction* is an expression that builds a composite value from its components.

A C++ structure construction:

```
enum Month {jan, feb, mar, apr, may, jun,
            jul, aug, sep, oct, nov, dec};
struct Date {
    Month m;
    byte b;
};

struct Date today = {mar, 17};
```

An ada construction:

```
size: array (Month) of Integer :=
    (feb => 28, apr|jun|sep|nov => 30, others => 31);
```

A Haskell list construction:

```
[31, if isLeapYear (year) then 29 else 28, 31, 30,
 31, 30 ,31 ,31, 30, 31]
```

C++ constructions can only occur as initialisers and the components must be literals.

Java objects have a constructor which is invoked by the **new** command.

1.7 Bindings and Environment

A *binding* is a fixed association between an identifier and a variable, value, or procedure.

An *environment* also called a *namespace* is a set of bindings.

Consider the following Ada snippet:

```
procedure p is
    z: constant Integer := 0;
    c: Character;

    procedure q is
        c: constant Float := 3.0e-02;
        b: Boolean;
    begin
        ...                         --(2)
    end q;
begin
    ...                             --(1)
end p;
```

At point (1) the environment is:
c: a character
p: a procedure
q: a procedure
z: an integer 0

at point (2) the environment is:
b: a boolean
c: a real number 3.0e-02
p: a procedure
q: a procedure
z: an integer 0

A *bindable* entity is one that can be bound to an identifier.
Languages differ on what is bindable:

C	Java	Ada
types	values	types
variables	local variables	values
function procedures	instance variables	variables
	class variables	procedures
	methods	exceptions
	classes	packages
	packages	tasks

1.8 Scope

The *Scope* is the portion of the program for which the declaration or binding has an effect.

1.8.1 Blocks

A *block* is a construct that delimits the scope of any declaration within it.

In C, blocks include block commands, function bodies, compilation units, and the program itself.

In Python, the blocks are modules, function bodies and class definitions.

In Java, the blocks are block commands, method bodies, class declarations, packages and the program itself,

In Ada, the blocks are block commands, procedure bodies, packages, tasks, protected objects, and the program itself.

When the program has only one block, it has *monolithic block structure*, meaning every declaration is global.

A language with *flat block structure* has a number of non-overlapping blocks, meaning a declaration's scope is the block in which it is declared.

In a language with *nested block structure*, allows blocks to be constructed within other blocks, and in languages like Ada the blocks can even overlap.

A language is said to be *statically* scoped if the body of a procedure is executed in the environment of the procedure's definition. Inversely, in a *dynamically* scoped language, the procedure is executed in the environment of the procedure's invocation.

2 Pointers and Memory Management

2.1 Dynamic Data Structures

Generally we don't know how much data a program will have to process. This can be handled in 2 ways:

- Create a fixed data structure.
  - Inefficient and wasteful, especially when individual elements may vary in size.
- Create a dynamically sized structure that grows and shrinks as needed
  - As more data arrives, memory is requested from memory pool.
  - When the memory holding the data is no longer needed, it returns to the memory pool.

A *Pointer* is a value that is a reference to a block of memory. Pointers are used either explicitly, like in C, C++, or Ada, or implicitly, like in Java.
In languages designed for system-level programming, pointers are used to hold the address of memory-mapped hardware resources.

There are 2 operations on dynamic memory:

- **Allocate** memory
  - Most languages do this by requesting a block of memory, the run-time system then allocates a sufficient block of memory for the memory pool and returns a pointer to that allocated block.
- **Release** memory
  - The allocated memory is returned to the memory pool
  - In some languages, like C, this is controlled by the programmer, meaning if memory is not release, it is lost in what is called a *"memory leak"*.
  - In some languages, like C++, it is controlled by both the programmer and the *data lifetime structure*
  - In others, like Java, it is totally automatic, and is controlled by a process called *garbage collection*

2.1.1 Pointers and Dynamic Memory in C

In C, a pointer is declared as `type *name;`.
The address of the variable called name is `&name`.
`*ptr` accesses the memory being pointed to.
`p->x` accesses the element `x` of a struct that `p` points to.

The most common library function for requesting memory in C, is `malloc(size)`, which returns a pointer to the allocated block. If there is insufficient memory, it returns a `NULL` value.

The library function for releasing dynamic memory pointed to by `p` is `free(p)`

2.1.2 Lists in C

The following is a way to implement a list of Ints in C:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int item;
    struct Node *next;
};

void add_to_list (struct Node **list , int value) {
    struct Node **ptr = list;
    struct Node *new_node = malloc(sizeof(struct Node));

    new_node->item = value;
    new_node->next = *ptr;
    *ptr = new_node;
    return;
}

void print_list (struct Node *list) {
    struct Node *ptr = list;

    printf("[");
    while (ptr != (struct Node *) NULL) {
        pritnf("%d ", ptr->item);
        ptr = ptr->next;
    }
    printf("]\n");
    return;
}

void delete_item (struct Node **list , int item) {
    struct Node *ptr, *prev = (struct Node *) NULL;

        ptr = *list;
        while (ptr != (struct Node *)NULL &&
                ptr->item != item) {
                prev = ptr;
                ptr = ptr->next;
        }
    if (ptr != (struct Node *)NULL) {
        if (prev == (struct Node *)NULL)
            *list = ptr->next;
```

```
        else
            prev->next = ptr->next;

        free(ptr);
    }
    return;
}
```

2.2 Lists in C++

```
#include <iosteam>
using namespace std;

class List {
    struct Node {
        int item;
        Node *next;
    };

    Node *head;

public :
    List() {              // constructor
        head = NULL;
    }

    ~List() {             // destructor
        Node *n = head->next;
        delete head;
        head = n;
    }

    void add(int value) {
        Node *n = new Node;
        n->item = value;
        n->next = head;
        head = n;
    }

    void remove(int item) {
        Node *n = head, *prev = NULL;

        while (n != NULL && n->item != item) {
```

```
            prev = n;
            n = n->next;
        }

        if (n != NULL) {
            if (prev == NULL)
                head = n->next;
            else
                prev->next = n->next;

            delete n;
        }
    }

    void print (void) {
        Node *n = head;
        cout << "[";
        while (n != NULL) {
            cout << n->item << " ";
            n = n->next;
        }
        cout << "]\n";
    }
};
```

2.3 Linked Lists in Java

```
class Node {
    public int item;
    public Node next;

    public Node() {
        next = null;
    }

    public void display() {
        System.out.print(item+" ");
    }
}

public class LinkedList {
    private Node head;
```

```
    public LinkedList() {
        head = null;
    }

    public void add (int item){
        Node newNode = new Node();
        newNode.item = item;
        newNode.next = head;
        head = newNode;
    }

    public void remove (int item) {
        Node temp = head, prev = null;

        while (temp != null && temp.item != item) {
            prev = temp;
            temp = temp.next;
        }

        if (temp != null) {
            if (prev == null) {
                head = temp.next;
            }
            else {
                prev.next = temp.next
            }
        }
    }

    public void print() {
        Node temp = head;

        System.out.print ("[");
        while (temp != null) {
            temp.display();
            temp = temp.next;
        }
        System.out.println("]");
    }
}
```

2.4 Garbage Collection

The memory used to store Java objects is called the *heap*. Whenever the `new` operator is called, sufficient memory is allocated from the heap for the object being created, and a *reference* to the allocated heap memory is returned.

There are 2 major approaces to Garbage Collection:

- reference counts
- mark and sweep

2.4.1 Reference Counts

Whenever the reference to a block of allocated heap memory is assigned, the *reference count* for that block is incremented.

When a variable that contains a reference to the allocated memory goes out of scope, the *reference count* is decremented.

When the *reference count* for that block of memory becomes zero, them it is no longer used and is available for garbage collection.

2.4.2 Mark and Sweep

Mark and Sweep is the approach used in Java's *Garbage Collection*. There are a few approaches to *mark and sweep* but they all follow this basic structure:

1. All other threads/activities are suspended for the duration of the garbage collection process
2. Perform a *mark and sweep*
    - The mark phase starts with the following *garbage collection roots*:
        (a) local variables and input parameters of currently executing modules are marked
        (b) active threads are marked
        (c) static fields of loaded classes are marked
        (d) JNI references are marked
    - The *Garbage Collector* transverses the object graph starting from the roots, marking every visited object as alive.
    - All unmarked objects are therefore not alive and are returned to the *heap*
3. Sometimes the memory may be *compacted* after garbage collection, meaning all remaining blocks are in a contiguous block of memory at the start of the heap, allowing for better partitioning of heap memory.

Generational Garbage Collectors work on the assumption that most objects are short-lived and will be ready for garbage collection shortly after being created. They can be divided into three sections:

1. The Young Generation:
   - Consists of `Eden`, where all new objects are created
   - Also has two `survivor areas` to which objects will move from `Eden` if they survive a garbage collection cycle
2. The Old Generation:
   - Where long-lived objects move to from the young generation
3. The Permanent Generation:
   - This contains the program's classes and methods.
   - Classes that are no longer in use may be garbage collected from the Permanent Generation.

3 Abstraction

Abstraction allows us to focus on general ideas rather than specific manifestations of these ideas.

It allows us to seperate *what* a program does from *how* it does it.

3.1 Control Flow

Control flow constructs can be classified as:

`sequencers`	Influence the flow of control
`jumps`	Transfer control to anywhere in the program
`escapes`	Transfer control out of an enclosing command or procedure
`exceptions`	Signal and handle abnormal situations

3.1.1 Sequencers

A *sequencer* transfers control from one point in a program to the sequencer's `destination`.

Sequencers are able to carry values that can be used at the sequencer's `destination`.

3.1.2 Jumps

Jumps are sequencers that transfer control to a specified point in a program. Some languages support jumps, but some modern languages, like Java, do not.

Jumps are usually implemented by `goto L`, where `L` is a label in the program.

Jumps are usually restricted to the scope of the label `L`. In C, the scope of a label is te smallest enclosing block command. It is possible to jump within the block command, but it is not possible to jump to a label in a block command from outside the label's block.

3.1.3 Escapes

An *escape* sequencer terminates the execution of a textually enclosing block command or procedure. Escape allows us to build single-entry, multiple-exit code fragments.

The most common examples of escape sequencers are `exit` in Ada, and `break` in

C, C++, and Java, as well as the `return` statement in all of the aforementioned languages.

3.1.4 Exceptions

Exception sequencers handle abnormal situations that stop the program from continuing normally, such as arithmetic overflows or incomplete I/O operations.

The simplest way to deal with an abnormal situation is to simply terminate the program with an error message, but it control can be transferred to a *handler* which will try to recover from the abnormal situation.

3.2 Procedures

Procedural Abstraction concerns:

- proper and function procedures
- passing parameters and arguments
- implementation of procedures

A procedure is defined by its *implementers* and their focus is on how the procedure's outcome is achieved efficiently.

A procedure is used by *application programmers* and thier focus is on the procedure's *observable outcome*.

3.2.1 Proper Procedures

A *proper procedure* contains a command that will be executed and, when it is called, it will update variables. The application programmer only observes these updates, and not how they were implemented.

In C and C++, a proper procedure's definition has the from:

$$\texttt{void}\ I\ (FPD_1, ..., FPD_n)\ B$$

where I is the procedure's identifier, FPD_i are the *formal parameter declarations*

3.2.2 Function Procedures

When a *function procedure* is called, it effectively evaluates an expression and returns a *result*.

In C and C++, the form of a function procedure is:

$$T\ I(FPD_1, ..., FPD_n)\ B$$

where I is the function's identifier, FPD_i are the *formal parameter declarations*, B is a block command called by the function's body, and T is the *result type* returned by the function body.

A function is called by an expression:

$$I(AP_1, ..., AP_n)$$

where AP_i are *actual parameters*. This call of I causes B to be executed. The first `return` of B determines the result.