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.

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

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