Hugh Baldwin
up2157117

Discrete Mathematics and Functional Programming

M21274
TB2

University of Portsmouth
BSc Computer Science
2nd Year

Contents

Part I

Discrete Mathematics

Lecture - Sets

17:00 23/01/24 Janka Chlebikova

A set is a collection of objects, known as elements or members (I will stick to members). Each member only appears once in the set. There is no particular order for members of a set, so there are several different ways to represent the same set. The members of a set can be just about anything, as long as they all abide by the same rules, and are in some way related.

Notation

There are several ways of noting a set, such as writing out all of the members of the set, or by using a rule which describes all of the members of a set.

For example, the following sets are equivalent

- $A = \{1, 2, 3, 4, 5\}$
- $A = \{x \mid 0 < x \leq 5\}$

If the object, x is in the set S, you would write it as $x \in S$. If not, it would be written as $x \notin S$ You can also describe a set by specifying a propery that the members share, e.g.

- $B = \{3, 6, 9, 12\}$
- $B = \{x \mid x \text{ is a multiple of } 3, \text{ and } 0 < x \leq 15\}$
- $$\begin{aligned} S &= \{\dots, -3, -1, 1, 3, \dots\} \\ &= \{x \mid x \text{ is an odd integer}\} \\ &= \{x \mid x = 2k + 1 \text{ for some integer } k\} \\ &= \{x \mid x = 2k + 1 \text{ for some } k \in \mathbb{Z}\} \\ &= \{2k + 1 \mid k \in \mathbb{Z}\} \end{aligned}$$

The Number Sets

Some letters are reserved for specific sets of numbers which can be used elsewhere to simplify definitions, the following are the most commonly used number sets

- $\mathbb{N}$ is used for the set of natural numbers, $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$
- $\mathbb{Z}$ is used for the set of integers, $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- $\mathbb{Q}$ is used for the set of rational numbers, $\mathbb{Q} = \{0, \frac{1}{2}, \frac{1}{3}, \dots\}$
- There is also the empty or null set, $\emptyset$ which contains no items, so $\emptyset = \{\}$

A set can either by finite or infinite, and the cardinality of a set is the number of members, e.g. $|S| =$ the number of members of S. For example, $\mathbb{N}$ and $\mathbb{Z}$ are infinite sets, and the set $A = \{1, 2, 3\}$ is a finite set with a cardinality of 3, so $|A| = 3$

Subsets

If every member of A is also a member of B, A is said to be a subset of B, which can be written as $A \subseteq B$. If B also has at least 1 member which is not a member of A, then A is a proper subset of B, which can be written as $A \subset B$. If A is not a subset of B, it can be written as $A \not\subseteq B$. Since the null set, $\emptyset$ contains no elements, it is a subset of every other set.

Equality of Sets

If two sets, A and B are equal, they have exactly the same members, which can be written as $A = B$. Alternatively, $A = B$ if the following conditions are true:

- $A \subseteq B$, and so for each x, if $x \in A$ then $x \in B$
- $B \subseteq A$, and so for each y, if $y \in B$ then $y \in A$

Operations

The intersection of two sets, A and B is every member in both sets, $A \cap B$. For example, the intersection of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \cap Y = \{4, 5\}$. If there are no common members, then the two sets are said to be disjoint. You can remember this by the fact that $\cap$ looks like an n, and therefore is the I**n**tersection of two sets.

The union of two sets, A and B is every member in either set, $A \cup B$. For example, the union of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \cup Y = \{1, 2, 3, 4, 5, 6, 7, 8\}$. You can remember this by the fact that $\cup$ looks like a U, and therefore is the **U**nion of two sets.

The difference of two sets, A and B are all members of the first set which are not members of the second set, $A \backslash B$. For example, the difference of the sets $X = \{1, 2, 3, 4, 5\}$ and $Y = \{4, 5, 6, 7, 8\}$ is $X \backslash Y = \{1, 2, 3\}$. This is the effectively subtracting the sets, $X - Y$.

If we consider all of the sets to be a subset of a particular set, U which contains all of the members of the "Universe of Discourse", then the complement of a set, A is any members of U whcih are not in A. This is represented as either A' or $\overline{A}$

All of these operations can be represented using a Venn diagram.

Like binary arithmetic, these operations follow a few rules:

- Commutative - $A \cup B = B \cup A$ and $A \cap B = B \cap A$
- Associative - $(A \cup B) \cup C = A \cup (B \cup C)$ and $(A \cap B) \cap C = A \cap (B \cap C)$
- Distributive - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- de Morgan's - $(A \cap B)' = A' \cup B'$ and $(A \cup B)' = A' \cap B'$

To get the cardinality of the union of two finite sets, you might think it would just be $|A \cup B| = |A| + |B|$, however, this results in counting $|A \cap B|$ twice, and so the correct cardinality is $|A \cup B| = |A| + |B| - |A \cap B|$

The Power Set

The power set is a set containing all subsets of the set, so if $S = \{a, b, c\}$, then the power set $P(S)$ would be $P(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. If the set S has n members, $P(S)$ has 2^n members.

Part II

Functional Programming

Lecture - Intro to Functional Programming

12:00 22/01/24 Matthew Poole

- For this module, we will be using the GHC (Glasgow Haskell Compiler), or more specifically it's interactive shell, GHCi

Imperative VS Functional Programming

- Most programming languages are imperative
  - Such as Python, JavaScript, C, etc
- Functional programming is another programming paradigm, which is based upon the mathematical concept of a function
- Imperative programming has state, statements (or commands) and side effects
- Pure Functional programming has no state, statements, or side effects
- A side effect is the change of state caused by calling a functionl assigning a variable, etc
  - This means that it is not always possible to predict the result of running a program, even with access to it's source code
- Since most programs need to cause a side effect (usually outputting data), most functional programming languages are not purely functional, but tend to organise the code such that only one part causes side effects

Functional Programming Languages

- There are two types of functional programming languages
- Pure
  - Languages such as Haskell
  - Has absolutely no state or side effects
- Impure
  - Languages such as ML, Clojure, Lisp, Scheme, OCaml, F#
  - Has some state or side effects, either everywhere or in a specific part of code
- There are also some functional constructs in major imperative langages such as Python, JavaScript, and more

FP Basics

Expressions

- An expression is a piece of text which has a value
- To get the value from the expression, you evaluate it
- This gives you the value of the expression
- e.g.
- ```
  Expression -> evaluate -> Value
  2 * 3 + 1  -------------> 7
  ```

Functions

- A function whose output relies only upon the values that are input into it
- The result will always be the same, given the same values
- This is the same as a mathematical function, which is where the name Functional Programming comes from

Haskell Basics

- In Haskell, all functions have higher precidence than operators
- This means that you have to explicity use brackets to ensure the correct order of operations
  ```

Lecture - Intro to Functional Programming II

12:00 22/01/24 Matthew Poole

Tracing a Functional Program

In an imperative program, it is obvious that you trace the program by determining the effect of each statement on the overall state of the program. However, with a functional program, each step of the tracing process is evaluating an expression, known as calculation. For example, we can trace the following program

```
twiceSum :: Int -> Int -> Int
twiceSum x y = 2 * (x + y)

twiceSum 4 (2 + 6)
```

by replacing each of the parameters of `twiceSum` as below

```
twiceSum 4 (2 + 6)
~> 2 * (4 + (2 + 6))
~> 2 * (4 + 8)
~> 2 * 12
~> 24
```

As above, in Haskell, the arguments are passed into the function verbatim, so the first step of executing a function is usually evaluating the arguments. This means that Haskell uses Non-Strict Computation - The arguments are passed into the function before being evaluated. Some functional programming languages use Strict Computation, meaning that the arguments are evaluated before being passed into the function. It doesn't really make a difference, other than that you may be asked to use a specific method in questions on exams, etc.

Guards

Guards are boolean expressions which can be used when defining a function to give different results, depending upon the input or a property thereof. This is especially useful for **Guarding** against invalid inputs. The syntax in Haskell is as below

```
maxVal :: Int -> Int -> Int
maxVal x y
  | x >= y    = x
  | otherwise = y
```

If the first guard is true, then the corresponding result is returned. If it's false, the next guard is evaluated, and corresponding value returned. You can also create a "default" case which is used if none of the other guards are true, which uses the keyword `otherwise`. Guards can also be used instead of a chain of if statements, which is easier to understand, and simpler to create in the first place.

Local Definitions

If your function uses a very complex mathematical calculation, you may want to break the calculation into several steps. In Haskell, you can do this using Local Definitions. Using the `where` keyword, you can define what are effectively local variables. This is useful as you can break down complex calculations into multiple, less complex and easier to understand ones. For example, the following function,

```
distance :: Float -> Float -> Float -> Float
distance x1 y1 x2 y2 = sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

could also be written as

```
distance x1 y1 x2 y2 = sqrt (dxSq + dySq)
  where
  dxSq = dx^2
  dySq = dy^2
  dx = x1 - x2
  dy = y1 - y2
```

The order of local definitions is irrelevant, and will still work if you reference a definition before it is actually defined. Local definitions are only usable within the function they are defined, hence “local”, but are able to reference each other, and the parameters of the function. As well as “variables”, you can also define “functions” as local definitions. This is useful to simplify repetitive code, which is used only within the function itself. Local definitions can also be used in conjunction with guards, in which case they are defined after the guards.