Functional and Logic Programming
Fall 2022
Review of Sets and Functions.

Contents

1 Introduction

These notes are being created in conjunction with the teaching of CPSC 370 in the Fall 2022 term at the University of Northern British Columbia.

Functional programming has far more to do with programming than it does with mathematics. Nevertheless, it is important to understand how functions in the mathematical sense relate to functions (or methods or programs or subroutines) in computer science sense.

2 A Review of Sets and Relations

I assume that you know Chapter 3 of Grimaldi. In particular, you know about sets, set union ($A \cup B$) , set interactions ($A \cap B$) , Cartesian products ($A \times B$) , and power sets ($\mathcal{P}(A) := \{ B : B \subseteq A\}$).

I also assume that you know basic cardinality formulas:

- $|A \cup B| + |A \cap B| = |A| + |B|$.
- $|A \times B| = |A| \cdot |B|$.
- $|\mathcal{P}(A)| = 2^{|A|}$.

Relations A *relation* r is just a subset of a Cartesian product; typically the product is of two sets and we speak of *binary relations*, but products of three sets (*termary relations*), or more are possible. One common application of high arity relations is tables in relational databases.

For instance, the set $q = \{ (a, b) \in \mathbb{R} \times \mathbb{R} : a < b\}$ is a binary relation on $\mathbb{R}$. We normally call this relation "$<$". We often write $a\ q\ b$ as a short form for $(a, b) \in q$.

\+ **Question 1.** How many distinct binary relations are possible between a five-element set and a two-element set?

Partial Functions Partial functions are a new idea that lies part way between relations and functions. Every function is a partial function[1] Every partial function is a relation. However, not every relation is a partial function, and not every partial function is a function.

Definition 2. A relation r where $r \subseteq D \times C$ is a *partial function* from D to C that satisfies the following well-definedness property:

- if $(d, c_1) \in r$ and $(d, c_2) \in r$ then $c_1 = c_2$, in CPSC 141 notation

F1 $\forall d \in D\, \forall c_1, c_2 \in C \qquad (d, c_1) \in r \ \&\ (d, c_2) \in r \rightarrow c_1 = c_2$.

More informally, a relation is a partial function if every question has at most one answer. Note that we translate "at most one" into "if there are two, then they are the same".

[1] Note really carefully that "partial function" means "could be partial", **not** "is not a function".

Functions A function is a partial function where every question has an answer. Every function is a partial function. However, not every partial function is a function.

Definition 3. A relation r where $r \subseteq D \times C$ is a *function* from D to C that satisfies the following two properties:

- if $(d, c_1) \in r$ and $(d, c_2) \in r$ then $c_1 = c_2$, in CPSC 141 notation

F1 $\forall d \in D \, \forall c_1, c_2 \in C \qquad (a, c_1) \in r \;\&\; (a, c_2) \in r \rightarrow c_1 = c_2.$

and

- if ($a \in D$ then there is a $c \in C$ such that $(a, c) \in r$, in CPSC 141 notation

F2 $\forall a \in D \, \exists c \in C$ s.t. $(a, c) \in r$.

More informally, a partial function is a function if every question has at least one answer, or equivalently, a relation is a function if every question has exactly one answer.

In the above situation, the set D is called the *domain*, and the set C is called the *co-domain*. The set $\{ c \in C : \exists d \in D[(d, c) \in r]\}$ is called the *range*.[2]

For f a function from D to C, the notation $f(d)$ means the unique value $c \in C$ such that $(d, c) \in f$. In other words, $f(d) = c$ if and only if $(d, c) \in f$. If f is a partial function, then $f(d)$ may not be defined.

Notation for functions Mathematicians tend to define functions in one of two ways. When giving a name, they often write something like $f(x) = 2 + x.$ which implicitly defines f as a function. Which function? Well, we need to know the domain of f to answer that question; suppose that f is a real-valued function, then f is the set $\{ (r, r + 2) \mid r \in \mathbb{R}\}$.

If they don't want to give the function a name, they'll sometimes write $x \mapsto 2 + x$, which means the same as the above. Note that it actually makes sense to write $f = x \mapsto 2 + x$.

[2]Strictly speaking, we cannot determine the domain or co-domain of a binary relation from its set of pairs alone. For this reason it is more correct to consider a partial function or function to consist of a triple: the domain, the co-domain, and the set of pairs that defines its relation.

Sets of functions The set of functions from the set A to the set B is written B^A. **Note carefully** that the co-domain is the base and the domain is the superscript. Part of the reason for doing so is the cardinality equation:

$$\left|B^A\right| = |B|^{|A|}.$$

The notation $f \in B^A$ means by definition that f is a function from A to B. Mathematicians sometimes write this as "$f : A \to B$" or even "$A \xrightarrow{f} B$"

Reverses, inverses, one-to-one, onto Suppose that $f : A \to B$ is a function. Because f is a function, it is a relation, that is a set of pairs $(a, b) \in A \times B$. Therefore we can define another binary relation f^{rev} from B to A defined by $f^{\text{rev}} = \{\, (b, a) \in B \times A : f(a) = b\}$. A natural question is: when is f^{rev} a function?

It turns out that f must have two properties for f^{rev} to be function.

1. f must be **one-to-one**, which means that for all $a_1, a_2 \in A$ if $f(a_1) = f(a_2)$ then $a_1 = a_2$. Note that one-to-one is *not* the same as being a partial function. Howwever, it is close. If f is one-to-one, then f^{rev} is a partial function.
2. f must be **onto**, which means that for all $b \in B$ there is an $a \in A$ such that $f(a) = b$. In other words, the range of f is the entire co-domain. Again, note that onto is *not* the same as property **F2**. Howwever, if f is onto and f^{rev} is a partial function, then f^{rev} is a function.

When the relation f^{rev} is a function, it is the inverse function of f: $f^{\text{rev}}(f(x)) = x$ and $f(f^{\text{rev}}(y)) = y$.

To summarize: a function is invertible if and only if it is one-to-one and onto. The inverse of an invertible function can be gotten by reversing all of the pairs.

Converting partial functions to functions Mathematicians and computer scientists tend to feel more comfortable dealing with functions than with partial functions.

In mathematics, you can often "fix" a partial function simply by restricting its domain. For instance, $f(x) = 1/x$ is a partial function on $\mathbb{R}$, but a total function on the set $\mathbb{R} \setminus \{\, 0\}$. However, in many programming languages, we don't have a way to name the set $\mathbb{R} \setminus \{\, 0\}$, so we are stuck with

```
public static double f(double x) { return 1.0 / x ; }
```

which is a partial function.

One standard way that any partial function can be converted to a function is to extend the co-domain by one element. If f is a partial function from A to B, we can extend it to a function $f_\perp$from A to $B_\perp = B \cup \{\perp\}$ (where $\perp \notin B$) as follows:

$$f_\perp(x) = \begin{cases} f(x) & \text{when } f(x) \text{ is defined,} \\ \perp & \text{otherwise.} \end{cases} \tag{1}$$

+ **Question 4.** Let $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ be the set of Boolean values. How many *partial* functions are there from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B}$?

+ **Question 5.** Let $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ be the set of Boolean values as above, and let $U = \{\,\mathsf{scissors}, \mathsf{paper}, \mathsf{rock}, \mathsf{Spock}, \mathsf{Lizard},\}$. How many *partial* functions are there from $S \times S$ to $\mathbb{B}$?

In Java, if the co-domain is a class type, then it is natural to extend partial functions by using `null` to extend the co-domain. This leads to ambiguity as to whether the intended co-domain includes the null value.[3]

3 Multi-argument functions and Currying

There's a tension between how mathematicians speak of functions as having one domain, and computer languages like Java talking about functions with multiple (or even zero) arguments. Even mathematicians talk about things like "two argument functions" and "partial derivatives". What's going on?

The answer is that the domain can be a Cartesian product of sets, so

$$h(x, y) = ye^x$$

is likely a function whose domain is $\mathbb{R} \times \mathbb{R}$ and whose co-domain is $\mathbb{R}$. Similarly,

$$g(n, x) = \cos nx$$

[3] See `https://blogs.oracle.com/java/post/java-8s-new-type-annotations`.

might be a function whose domain is $\mathbb{N} \times \mathbb{R}$ and whose co-domain is $\mathbb{R}$.

What is g as a set? Well, it's an infinite set. It contains domain/co-domain pairs like $((0, 4.2), 1)$ and $((2, \pi/3), -\frac{1}{2})$, where each domain element is itself a pair. What happens if we re-bracket the elements? Like $(0, (4.2, 1))$ and $(2, (\pi/3 - \frac{1}{2}))$? The result is not a function, because pairs in g like $((1, 0), 1)$ and $((1, \pi), -1)$ become pairs like $(1, (0, 1))$ and $(1, (\pi, -1))$, which violate the "at most one answer" rule. However there is a function lurking here.

3.1 Currying

Sometimes we want to think of g as a series of cosine functions, like

$$c_n(x) = g(n, x) = \cos nx$$

It's clear what $c_3(\pi/2)$ is. We have $c_3(\pi/2) = g(3, \pi/2) = \cos(3\pi/2) = 0$. What is c_3? It's the function $x \mapsto \cos 3x$. Similarly c_0 is the function $x \mapsto \cos 0x$ (or $x \mapsto 1$).

Here's the big question. What is c? It's a function from $\mathbb{N}$ to the c_ns, that is, it's a function from $\mathbb{N}$ to $\mathbb{R}^{\mathbb{R}}$, that is $c : \mathbb{N} \to (\mathbb{R} \to R)$. Notice that the domain of c is simpler than the domain of g, it's just natural numbers rather than pairs. However, the co-domain is more complicated: it is functions from $\mathbb{R}$ to $\mathbb{R}$, that is sets of pairs from $\mathbb{R} \times \mathbb{R}$. For instance c contains the pair

$$(1, c_1) = (1, \{\, (0, 1), (\pi, -1), \ldots\})$$

This is the function that we hinted at at the end of the previous section.

One of the characteristics that distinguishes functional programming languages like Haskell from C++ is that in Haskell functions can return functions as their value, like c above.

We can write this mathematically as:

$$c : \mathbb{N} \to (\mathbb{R} \to \mathbb{R}), \qquad n \mapsto (x \mapsto \cos nx). \tag{2}$$

We cannot directly write a function like c this in C++ but we will see how we can write this in Haskell. This act of converting a function from the form $\mathbb{N} \times \mathbb{R} \to \mathbb{R}$ to the form $\mathbb{N} \to (\mathbb{R} \to \mathbb{R})$ is called *Currying*.

Currying is in honour of Haskell Curry, who did a lot of early exploration of what we now call the λ-calculus. He wanted a way to talk about multi-argument functions *without* needing to define what pairs and cross-products. He noticed that Cartesian product domains weren't necessary if the functions were "Curried".

Note that the the legitimacy of Currying a function is suggested by standard exponention laws. For positive integers we have $c^{ab} = c^{ba} = (c^b)^a$. Clearly this doesn't work exactly for sets, $A \times B \neq B \times A$, but it does suggest that the function $((n, x) \mapsto \sin nx) \in \mathbb{R}^{\mathbb{N}\times\mathbb{R}}$ is likely intimately related to the function $(n \mapsto (x) \mapsto \sin nx)) \in (\mathbb{R}^{\mathbb{R}})^{\mathbb{N}}$.

4 Function Composition

4.1 Function composition

Computer programmers are quite familiar with the notion of using "functions" to build other "functions", and all main-stream programming languages have mechanisms for designing separate functions and calling one from the other.

In mathematics too functions can be composed, but the form of composition is much stricter. If we write $h = f \circ g$ we mean that $h(x) = f(g(x))$. Note that "$\circ$" is itself a function that takes functions as arguments and returns a function.

5 Functions in Programming Languages

Many programming language methods are *not* functions, often by design because they return multiple answers to the same question. For instance, consider the JAVA static method `System.nanoTime()`.

\+ **Question 6.** Is the JAVA "function" `Math.random` with signature `int random()` a mathematical function?

\+ **Question 7.** Is the JAVA "function" `System.out.println` with signature `void println(?)` a mathematical function?

Note also that most conventional programming languages deal mainly with (partial) functions, and not more general relations. Prolog is a notable exception, as we shall see later in the course.

Haskell Notation for functions In Haskell function definitions looks quite similar to math:

```
f(x) = 2 + x
\ x -> 2+x   -- In math x ↦ 2 + x
f = \ x -> 2 + x ;
```

Note that there is notation for *anonymous* functions, that is functions that have no associated name. This is a common characteristic of functional programming languages. By contrast, in JAVA every function is a named method of class.[4] In C++, functions may exist outside of classes, but they still must be named[5].

5.1 Functions and Types

Haskell is staticly and strongly typed, meaning that the exact type of every expression is known at compile time. Paradoxically, Haskell is so strongly typed that we frequently don't need explicit type declarations.

[4]When these notes were first written, JAVA did not have λ-expressions. The claim is still true, but JAVA now has nice functional syntax for writing nearly anonymous methods that are part of anonymous classes that implement a "functioncal" interface.

[5]No longer true as of C++11. This is because C++ is now explicitly embracing some ideas from function programming languges.

However, we can, and frequently do, explicitly declare the type of functions. For instance, we can write

```
f :: Int -> Int -- math f : ℤ → ℤ
f(x) = 2 + x
```

or

```
g :: (Int,Double) -> Double -- math g : ℤ × ℝ → ℝ
g (n,x) = cos (fromIntegral n * x)
```

We can also Curry g as above and write

```
c :: Int -> Double -> Double -- math c : ℤ → ℝ → ℝ
c n x = cos (fromIntegral n * x)
```

5.1.1 Syntax Notes

There are some syntax points snuck into the examples above.

1. `--` introduces a comment that runs to the end of the line
2. "`->`" is used to be build function types from plain types.
3. "`->`" associates to the right. That is "`c :: Int -> Double -> Double`" means
   "`c :: Int -> (Double -> Double)`" and *not*
   "`c :: (Int -> Double) -> Double`".
4. The math "$x \mapsto \ldots$" translates into the Haskell "`\ x -> ...`".
5. Brackets are sometime necessary, but not for function application. "`fromIntegral n`" is ok, and the same as "`fromIntegral(n)`". However the brackets in "`cos (fromIntegral n * x)`" are necessary in order to force `cos` to be applied to the entire argument, rather than just "`fromIntegral`".
6. Curried functions are easy to write. The follwoing three declarations are equivalent.

   ```
   c n x = cos (fromIntegral n * x)
   c n   = \ x -> cos (fromIntegral n * x)
   c     = \ n -> (\ x -> cos (fromIntegral n * x))
   ```

Converting partial functions to functions in programming languages In C-like languages, some library functions employ this technique of extending the domain by one element in a disguised fashion. They return a pointer to the desired result, and return a null pointer to indicate an input outside of the domain of the function.

In strongly typed functional languages like Standard ML and Haskell there are standard types that add a $\perp$-like value. In particular, Haskell has a `Maybe` type function that adds the value `Nothing` For instance, on can write

```
arcCosH x = if x < 1
   then Nothing
   else Just (log (x + sqrt (x^2 - 1)))
```

To be continued. . .