Lecture 2: DATA TYPES AND FUNCTIONS [*]

10-606 MATHEMATICAL FOUNDATIONS FOR MACHINE LEARNING

1 Data types

In many programming languages, variables have types: e.g., `int32`, `char *`, or `dict`. The type of a variable tells us what kind of values we can assign to it: e.g, a variable of type `int32` can hold a signed integer that fits in 32 bits, using two's complement representation to handle negative numbers.

More importantly, a data type tells us an *interface*, a list of operations we can perform on values of this type. For example, there is an operation + that takes two `int32` values and produces another `int32` value (handling numerical overflow in a specified way).

The important thing about an interface is that it provides abstraction: it doesn't matter how an `int32` is represented under the hood, and in fact we rarely, if ever, have to think about this. We could even swap between two different implementations of `int32`; so long as they provide the same interface, our surrounding program shouldn't even notice.

1.1 Types as a formal system

A list of data types and their interfaces is a good example of a formal system: it tells us a mechanical way to generate and combine values of different types. If we follow the rules of the formal system, we're guaranteed to get values that make sense: e.g., we won't try to divide a `char *` by a `dict`. This guarantee applies no matter what specific implementations we use for the individual types.

1.2 Combining types

Because a data type is based on a set of allowed values, we can use a lot of the set operations we previously discussed to combine data types and make new ones. The difference is that when types are involved, we need to say how the interfaces combine as well.

An example is unions. In C, the expression

```
union {int a, char *b}
```

defines a type that we can interpret as the mathematical union between two simpler types: the type of values that are *either* integers or C strings. The interpretation is a bit subtle, though:

[*]Compiled on Friday 9th June, 2023 at 15:10

Knowledge check 1

1. **Short answer**: Where do we have to be careful when describing a C union type this way?
   - **Hint**: Frame your answer in terms of the interfaces associated with each data type in the union; are all operations you could perform on an `int` applicable to a `char *`?
   - **Answer**: Whenever we use a value of the above C type, we have to know ahead of time whether we want to interpret it as an `int` or a `char *`. By allowing us to define such a type, C is effectively violating our description of an interface, i.e., C is providing a loophole that lets us use the `char *` interface on values of type `int`. There have been many heated arguments over whether this is a good idea. But certainly if we do it unintentionally, bad things can happen: e.g., we can get a program with a bad security vulnerability.

1.3 Disjoint unions

Let's return to the C union type above. What we might expect, instead of C's default behavior, is that we know whether every value is intended as an integer or a string. This kind of combination of types is called a *tagged union* or a *disjoint union*: the same bit string (the internal representation of some value) is treated differently depending on whether it arrived as an `int` or a `char *`. Disjoint unions prevent us from (accidentally or on purpose) treating an object of one type as if it were an object of another type.

We can achieve the same effect in a programming language by requiring the types in a union to be disjoint. While C doesn't do this, some languages use the first few bits of a value to indicate its type. Or, we can achieve a similar effect in C by manually tagging the elements, e.g.,

```
struct {int tag, union {int a, char *b}}
```

It should be clear at this point that it's important to be explicit about which convention we're using: we have to pick a single formal system and stick with it if we want to get results that make sense. Either we do or we don't automatically remember what type of value we're currently storing, but if we think we do when we don't, bad stuff can happen.

2 Functions

One of the most interesting ways to combine types is to define a function: e.g.,

$$f(x, y) = 3x^2 + xy$$

The function f here is an object of a new type, which we can write as

$$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

This expression stands for the set of functions that take two real-valued arguments and return a single real-valued output.

In the above expression the new type is implicit: we infer it automatically based on the assumed types of the arguments.

Some programming languages are like this as well, e.g., Python doesn't require us to annotate function types, and will infer types automatically. (In fact, Python is quite aggressive about guessing types; this can be convenient but sometimes leads to unexpected behavior.) In other languages, like C, we have to annotate function definitions explicitly with input and output types, e.g.,

```
float f(float x, float y) { ... }
```

If we have a function $f \in X \to Y$, then for every value $x \in X$, there is a unique value $f(x) \in Y$. That is, functions are *complete* (defined for every input) and *single-valued* (never have ambiguous outputs).

An alternate notation for a function type $X \to Y$ is Y^X. This is by analogy to the set product notation Y^n: in Y^n we assign a unique value of type Y to every integer in $1 \dots n$. Similarly, in Y^X we assign a unique value of type Y to every input value of type X.

Knowledge check 2

1. **Select one**: Consider a function g that takes as input a function $f \in \mathbb{R} \to \mathbb{R}$ and returns another function of the same type that computes $f(x) + 1$. What is the type of this function?
   a) $\mathbb{R} \to \mathbb{R}$
   b) $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$
   c) $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$
   d) $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$
   - **Answer**: d, g takes as input a function and returns a function.
2. **Select one**: Consider the `floor` function that takes as input a real number and returns the largest integer not greater than the input. What is the type of this function?
   a) $\mathbb{R}^{\mathbb{R}}$
   b) $\mathbb{R}^{\mathbb{Z}}$
   c) $\mathbb{Z}^{\mathbb{R}}$
   d) $\mathbb{Z}^{\mathbb{Z}}$
   - **Answer**: c, the `floor` function maps the reals to integers and in this notation, the output space is the base and the input space is the exponent.