# Lab 6

Functional Programming (ITI0212)

2022-03-04

This week we met *indexed types*, also known as *dependent types*. These are type constructors that produce types that are *indexed by* or *dependend on* elements of another type. We met the bounded natural number types `Fin n` and the lengthed-list types `Vect n a`, both of which are indexed by natural numbers. We also met the dependent pair types `DPair a b`, in which the second factor `b` is indexed by the first factor `a`.

We saw that dependent types let us write more precise specifications for functions, which can be used to eliminate run-time errors such as index-out-of-bounds for finite sequences. We also saw that the use of dependent types can sometimes make functions easier to write by reducing the space of well-typed programs, sometimes to such an extent that Idris can write the function definitions for us.

Despite this, dependent types are not a panacea. While they make some programming tasks easier, they make others harder. Dealing with this new complexity is a topic that we will return to throughout the rest of the course. This week we will content ourselves with gaining familiarity with some basic indexed types and dependent functions.

For this lab you will need to import `Data.Fin` and `Data.Vect`.

**Task 1**
Write a *type isomorphism* between the types `Bool` and `Fin 2`; that is, write functions

```
bool_2_fin  :  Bool -> Fin 2
```

and

```
fin_2_bool  :  Fin 2 -> Bool
```

such that composing them in either order does the same thing as the identity function:

```
Lab6> (fin_2_bool . bool_2_fin) False
False
Lab6> (fin_2_bool . bool_2_fin) True
True
Lab6> (bool_2_fin . fin_2_bool) 0
FZ
Lab6> (bool_2_fin . fin_2_bool) 1
FS FZ
```

*Challenge:* how many such type isomorphisms are there?

**Task 2**
Write the `map` function for `Vect` types,

```
map_vect  :  (a -> b) -> Vect n a -> Vect n b
```

or better yet, let Idris write it for you.

**Task 3**
Sometimes we encounter *partial functions*, that is, functions that are undefined for some inputs. For example, division on the rational numbers is not defined when the second argument is 0. There are two canonical strategies for totalizing a partial function.

**expand the codomain:** replace the result type of the function with a "bigger" one that contains new elements to receive arguments on which the partial function is undefined. For example, some languages add a `NAN` (not-a-number) value to numerical types to accommodate zero-division.

**restrict the domain:** replace the source type of the function with a "smaller" one that does not contain the elements on which the partial function is undefined. For example, we could create a type of non-zero numbers and use that for the second argument to division.

The first element of the sequence is called its `head`. However, `head` is generally a partial function because there is no first element of an empty sequence.

(i) Define a total `head` function for `Vect` types by expanding the codomain:

```
head_e  :  Vect n a -> ?G1 a
```

(ii) Define a total `head` function for `Vect` types by restricting the domain:

```
head_r  :  Vect ?G2 a -> a
```

**Task 4**
Write a function called `indPair` that converts a (non-dependent) `Pair` to the `DPair` with the same factors. For example:

```
> indPair (True , 1)
(True ** 1)
> indPair (3.14 , ())
(3.14 ** ())
```

**Task 5**
`List` types and `Vect` types are both *finite sequence types*, made from constructors called `Nil` and `(::)`, and sharing the same syntactic sugar in the form of bracket notation, `[x, y, ...]`. Informally, we can think of a `Vect` as a `List` that knows its length.

Forgetting things is usually pretty easy. Write a function that converts a `Vect` into the `List` containing the same elements in the same order.

```
forget_length  :  Vect n a -> List a
```

Learning things is often a little harder than forgetting them. Write a function that converts a `List` into the `Vect` containing the same elements in the same order.

```
learn_length  :  (xs : List a) -> Vect ?n a
```

*Hint:* `learn_length` will need to be a *dependent function*. Recall that Idris' function type constructor lets us use the *value* of the argument when determining the *type* of the result. It may be useful to `:set showtypes` in the REPL to help distinguish `Vect`s and `List`s.