Lab 10

Functional Programming (ITI0212)

2021.03.30

This week we are learning about programming with dependent types. Unless you've done this before it is likely to take some getting used to. The solutions to the following tasks are all very short, but they are also quite "dense". As you work on them it is important to have a clear understanding of the type of each expression you encounter. The interactive editing features, especially `:t`, are very helpful for this. Keep in mind that if the type of an expression is `Type` then that expression is itself a type, which in turn may classify other expressions.

Task 1
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 2
We can use dependent pairs indexed by `Bool`s to define a type representing the disjoint union of two given types:

```
DisjointUnion : Type -> Type -> Type
DisjointUnion a b  =  DPair Bool (\ i => if i then b else a)
```

An element of type `DisjointUnion a b` will have an element of type `a` in its second factor if it has `False` in its first factor, and will have an element of type `b` in its second factor if it has `True` in its first factor. Thus, in order to write a function from the type `DisjointUnion a b` to another type `c`, we'll need to know how to turn an `a` into a `c` (because we might find `False` in the first factor) and we'll also need to know how to turn a `b` into a `c` (because we might find `True` in the first factor). With this in mind, write a function for mapping out of `DisjointUnion` types:

```
fromDU  :  (a -> c) -> (b -> c) -> DisjointUnion a b -> c
```

Task 3
In fact, `DisjointUnion` types are *isomorphic* (we'll explain this word later in the course) to `Either` types. Write conversion functions back and forth between these types such that if we compose them in either order we end up with the same thing that we started with.

```
fromEither :  Either a b -> DisjointUnion a b

toEither   :  DisjointUnion a b -> Either a b
```

Note: you can use your `fromDU` function from task 2 to make one of these a point-free one-liner.

Task 4
Write a parameterized *record* type called `DUrec` with fields called `index` and `value` that represents the same information as `DisjointUnion`. Then write the conversion functions:

```
fromDUrec :  DUrec a b -> DisjointUnion a b

toDUrec   :  DisjointUnion a b -> DUrec a b
```

Task 5

In task 2 we defined the disjoint union of two types using a dependent pair type whose first factor is `Bool`, a type with exactly two elements. Generalize this construction by using a dependent pair type to define a disjoint union (also called a "sum") of n types specified by a vector of length n:

```
arySum  :  (n : Nat) -> Vect n Type -> Type
```

For example:

```
> the (3 `arySum` [Unit , Bool , Nat]) (0 ** ())
(0 ** ())
> the (3 `arySum` [Unit , Bool , Nat]) (1 ** True)
(1 ** True)
> the (3 `arySum` [Unit , Bool , Nat]) (2 ** 42)
(2 ** 42)
```

Hint: the function `Data.Vect.index` will be helpful for this.

Task 6

Recall that the type constructor `Vect` is indexed by a `Nat` specifying the length of a sequence of elements. We can generalize this idea by instead using a vector of types as an index, as we did in task 5. This gives us the type constructor of *heterogeneous vectors*:

```
data HVect : Vect n Type -> Type where
  Nil : HVect []
  (::) : t -> HVect ts -> HVect (t :: ts)
```

The type of each element of an `HVect ts` is thus given by the corresponding element of `ts`. For example:

```
> :t [() , True , 42]
[(), True, 42] : HVect [Unit, Bool, Integer]
```

Write the following by-now familiar sequence-type functions for `HVect`s:

- `head` and `tail` for non-empty sequences,
- concatenation `(++)`,
- `index`.

For example:

```
> head [() , True , 42]
()
> tail [() , True , 42]
[True, 42]
> [() , True] ++ [42]
[(), True, 42]
> index 2 [() , True , 42]
42
```

Observe that `HVect` types are expressive enough that once you specify the types of these functions, automatically generate a clause, and case-split on the appropriate arguments, Idris can write the rest of each definition for you using term search.