Lab 9

Functional Programming (ITI0212)

2022-03-25

This week we saw record types, which are types that gather together a bunch of related fields, provide a convenient syntax for manipulating what we might otherwise implement as "types with one constructor" or iterated dependent pairs.

Defining a record gives us field projection functions defined in the namespace of the record, and field "update" functions which allow us to create new records from existing ones using a convenient syntax. This syntax also allows access and "update" of fields of records nested within records.

Task 1
Records could be used to represent posts on a social media site. Write a record `Votes` that can store a count of likes and dislikes, and a record `Post` that has a field for votes (nesting the `Votes` record), along with fields for the title and URL.

Task 2
Write a function `like : Post -> Post` that increases the number of likes of a post by one.

Task 3
Write a function `score : Post -> Integer` that calculates the score of a post as the number of dislikes subtracted from the number of likes.

Task 4
Consider a type of vehicles

```
data Vehicle : Type where
  Bike : Vehicle
  Car : Vehicle
  Plane : Vehicle
```

(extend this however you wish).

Write a function `max_speed : Vehicle -> Nat` that associates a maximum speed (in arbitary units) to each kind of vehicle.

Write a parameterized record type

```
record VehicleSpec (kind : Vehicle) where
```

for specifications of vehicles, parameterized by a type of vehicle. This should include fields for name, speed, year of manufacture, but a vehicle must not be able to go faster than the `max_speed` of its kind: the type of the speed field must *depend* on the `kind` parameter.

Recall: you may use the `Fin` type from `Data.Fin` to represent numbers strictly less than a bound.

Task 5
A finite indexed family of types can be specified by a term of type `Fin n -> Type`: for each index i, such a function gives us a type at that index.

The task is to implement a record that represents the disjoint union of such a family of types. It should have three fields: a finite indexed family of types, an index, and a value of the type at that index.

Such a disjoint union is like a generalization of the `Either` type: where `Either` disjointly unites two types, a disjoint union unites all the types in a finite family.