Lab 9

Functional Programming

2024-04-12

This week we are learning about indexed type families and dependent functions. An *indexed type family* (or "indexed type" or "dependent type (family)") is a type constructor where the resulting types are indexed by (or "depend on") elements of another type.

We met the indexed family of *finite types*, `Fin n`, whose elements are finite prefixes of the natural numbers bounded by `n`. We also met the *vector types*, `Vect n a`, whose elements are length `n` sequences of `a`s. Both of these type families are indexed by the type `Nat`.

We saw how we can define *tuple types* of arbitrary arity, `Tuple ts`, which are indexed by vectors of types. We also saw how the *dependent pair types*, `DPair a b`, generalize the ordinary pair types, `Pair a b`, in that the type of the second factor can depend on the value of the first factor.

A *dependent function* is one where the type of the result can depend on not only the type, but also the value, of the argument. The type constructor for dependent functions is built into Idris and is what makes its type system so expressive.

For this lab you should import `Data.Fin` and `Data.Vect`. It may also be helpful to `:set showtypes` in the REPL. Because the finite sequence types `List a`, `Vect n a`, and `Tuple ts` overload the "`[ ... ]`" notation, you may need to disambiguate their types at the REPL using the function `the`.

Task 1
Write a *type isomorphism* between the types `Bool` and `Fin 2`,

```
bool_2_fin  :  Bool -> Fin 2

fin_2_bool  :  Fin 2 -> Bool
```

so that:

```
Lab9> (fin_2_bool . bool_2_fin) True
True : Bool
Lab9> (fin_2_bool . bool_2_fin) False
False : Bool
Lab9> (bool_2_fin . fin_2_bool) 0
FZ : Fin 2
Lab9> (bool_2_fin . fin_2_bool) 1
FS FZ : Fin 2
```

Question: how many isomorphisms are there between these two types?

Task 2
Write the *map* function for vectors:

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

Once you specify the type you should be able to do this using only interactive editing commands, without typing a single expression into your editor. However, it is worth taking the time to examine each goal and its context to understand why Idris is able to write this function automatically.

In fact, each type constructor `Vect n : Type -> Type` is a `Functor` instance, so you can use the `map` method with vectors.

Task 3
Write the following function, which returns the element of a `Fin` type having the same "size" as its argument `Nat`, and that is the "largest" element of its type:

```
as_top  :  (n : Nat) -> Fin (S n)
```

Note that this is a *dependent function* because the result *type* depends on the argument *value*.

For example:

```
Lab9> as_top 0
FZ : Fin 1
Lab9> as_top 1
FS FZ : Fin 2
Lab9> as_top 2
FS (FS FZ) : Fin 3
```

Task 4
Write a function `two_tuple` that converts a `Pair` into a `Tuple` with the same elements in the same order. (You can use the definition of `Tuple` from lecture 9.)

For example:

```
Lab9> two_tuple ("hello" , 42)
["hello", 42]
Lab9> two_tuple (True , ())
[True, ()]
```

Defining this function is easy, indeed, Idris can do it for you once you figure out its type.

Task 5
Write a function `ind_pair` that converts a `Pair` into a `DPair` with the same elements in the same order.

For example:

```
Lab9> ind_pair ("hello" , 42)
("hello" ** 42)
Lab9> ind_pair (True , ())
(True ** ())
```

Defining this function is easy, indeed, Idris can do it for you once you figure out its type.

Task 6
Consider the following type:

```
Tricky  :  Type
Tricky  =  DPair Integer (\ n => if n `mod` 2 == 0 then String else Tricky)
```

Complete the following terms any way you like, so long as they are well-typed:

```
the Tricky (0 ** ?goal0)
the Tricky (1 ** ?goal1)
the Tricky (3 ** 5 ** ?goal2)
```

Note that `(**)` is right-associative.

Task 7

`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 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* because the result *type* depends on the argument *value*.