Lab 10

Functional Programming (ITI0212)

2023-04-07

This week we are learning about programming with dependent types. We saw how to write expressions that compute the types of other expressions, including the type of the `filter` function for `Vect` types and that of the `printf` function. In order for an expression to compute in a type it must be `total`, and if it is defined in a different module then it must have visibility `public export`.

If we want to perform case analysis on an expression of an inductive type and we need occurrences of that expression to compute in a type then we can use `with`, which allows us to pattern-match on the value of that expression on the *left* of a definition clause.

A useful technique for writing recursive functions is to use an *accumulator*, which is an additional argument that keeps track of how the value computed by a function changes with each recursive call. The value of the accumulator is then used to compute the result when a base case is reached.

Task 1
Write the ternary boolean *majority* function, which returns the `Bool` that occurs most often among its arguments, and whose type can be written using the `ary_op` type constructor from lecture:

```
majority3  :  3 `ary_op` Bool
```

Task 2
There is a similar *majority* function that takes a list of booleans as an argument (let's stipulate that ties go to `True`):

```
list_majority  :  List Bool -> Bool
```

Write this function in such a way that it makes *exactly one* pass over its argument list, which is optimal. Note that functions like `length`, `filter`, `count` (or `accepts`) *each* make one pass over a list, as you can confirm by `:printdef`ing them.

Hint: you can use a helper function that takes an additional *accumulator* argument that keeps track of what you know about the majority so far. When you reach the base case of an empty list you can use this accumulator to decide which boolean wins. As a bonus, your function will most likely be *tail recursive*, which means that it can run in constant space on a stack-based interpreter.

Task 3
Generalize the `ary_op` type constructor so that the argument and result types are arbitrary:

```
infixr 6 >->
(>->)  :  (args : Vect n Type) -> (result : Type) -> Type
```

Here the `infixr` declaration means that we can write this as an infix operator that defaults to right-association. The number specifies the precedence of this operator relative to other infix operators.

The `(>->)` type constructor should take a vector of argument types and a result type and return the type of *curried* functions from the argument types to the result type. For example:

```
Lab10> [] >-> Nat
Nat
Lab10> [Nat] >-> Nat
Nat -> Nat
Lab10> [Nat , Bool] >-> Nat
Nat -> Bool -> Nat
Lab10> [Nat , Bool , String] >-> Nat
Nat -> Bool -> String -> Nat
```

Test your definition by using it to specify the types of some expressions, such as:

```
seven  :  [] >-> Nat
seven  =  7

idty  :  [a] >-> a
idty x  =  x

compose  :  [(a -> b) , (b -> c)] >-> (a -> c)
compose f g x  =  g (f x)
```

Task 4
Rewrite the `ary_op` type constructor as an instance of the `(>->)` type constructor by completing the following definition:

```
ary_opp  :  (n : Nat) -> Type -> Type
n ‘ary_opp‘ a  =  ?args >-> ?result
```

Hint: `:search (n : Nat) -> a -> Vect n a`

Task 5
Write the following function:

```
weakened_by  :  Fin m -> (n : Nat) -> Fin (m + n)
```

which converts an element of type `Fin m` into the corresponding element of type `Fin (m + n)`. Note that this is a *dependent function* because the *type* of the result depends on the *value* of the second argument.

For example:

```
Lab10> the (Fin 3) 2 ‘weakened_by‘ 0
FS (FS FZ) : Fin 3
Lab10> the (Fin 3) 2 ‘weakened_by‘ 1
FS (FS FZ) : Fin 4
Lab10> the (Fin 3) 2 ‘weakened_by‘ 2
FS (FS FZ) : Fin 5
```

Task 6
Write a function `as_fin` that takes two `Nat` arguments and tries to interpret the first as an element of the `Fin` type determined by the second.

For example:

```
Lab10> 2 ‘as_fin‘ 4
Just (FS (FS FZ)) : Maybe (Fin 4)
Lab10> 2 ‘as_fin‘ 3
Just (FS (FS FZ)) : Maybe (Fin 3)
Lab10> 2 ‘as_fin‘ 2
Nothing : Maybe (Fin 2)
```