Lab 11

Functional Programming (ITI0212)

2022-04-08

This week we learned 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 inductively defined type and we need occurrences of that expression to be specialized to a constructor form then we can use a `with` pattern, which is like a `case` expression, except that it occurs on the left side 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 (for concreteness, ties go to `True`).

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

Write this function is 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: Like in the definition of `format_function` from lecture, 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 the state of 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 can be 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 describes the precedence of this operator with respect to other 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:

```
Lab11> [] >-> Nat
Nat
Lab11> [Nat] >-> Nat
Nat -> Nat
Lab11> [Nat , Bool] >-> Nat
Nat -> Bool -> Nat
Lab11> [Nat , Bool , String] >-> Nat
Nat -> Bool -> String -> Nat
```

Test your definition by using it to describe the types of some functions, such as:

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

is_even  :  [Nat] >-> Bool
is_even Z  =  True
is_even (S n)  =  not $ is_even n

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, i.e. complete the following definition:

```
ary_op'  :  (n : Nat) -> Type -> Type
n `ary_op'` a  =  ?args >-> ?result
```

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

Task 5
Modify the `printf` function from lecture so that it accepts `"%f"` as a formatting directive that specifies an argument of type `Double`.

Hint: you need to add only four lines.

This should let you write expressions like

```
Lab11> printf "pi is approximately %f" pi
"pi is approximately 3.141592653589793"
Lab11> printf "the square root of 2 is approximately %f" (sqrt 2)
"the square root of 2 is approximately 1.4142135623730951"
```