Lab 13

Functional Programming (ITI0212)

2023-04-28

This week we discussed how equality can be represented in the Idris type system. In this lab session we aim at giving you some familiarity with it and how it interacts with some constructs we saw previously.

Note: just like last week's lab, do not forget to add `%default total` to the beginning of your script. This will tell Idris that all the functions we are defining should be total, so that we do not accidentally fall into the case in which we can prove any statement by simply giving a function that does not terminate.

Warmup

Let's begin with some simple proofs.

Task 1 (Triple transitivity)
Prove the following:

```
triple_transt : {x, y, z, w : a}
    -> x = y -> y = z -> z = w -> x = w
```

Task 2 (Triple transitivity, with a twist)
We presented transitivity as a fact regarding the equality taken as a binary relation. But in the propositions-as-types paradigm, facts are types and proofs are objects. Therefore, you can regard transitivity as a binary operation, that takes a proof that `x = y`, a proof that `y = z` and combines them to produce a proof that `x = z`. To emphasize this, we can define it as an infix operator:

```
infixl 4 -*-
(-*-) : {x, y, z : a} -> x = y -> y = z -> x = z
(-*-) Refl Refl = Refl

-- Usage:
-- if p : x = y, q : y = z, then p -*- q : x = z
```

The task is the following: prove the same thing as the task above, without case splitting, using `-*-` instead.

```
triple_transt' : {x, y, z, w : a}
    -> x = y -> y = z -> z = w -> x = w
```

Hint: Think of it as "How do I get from `x` to `w`?"

Task 3 (Congruence across multiple functions)
If applying a function takes equals to equals, why shouldn't applying two functions do the same? Prove the following. Please try and do that using `cong`.

```
cong_mult_functs : (f : a -> b) -> (g : b -> c) -> {x, y : a}
    -> x = y -> g (f x) = g (f y)
```

Heavy lifting

We will now see how to use equality together with other inductively defined types we saw in the previous weeks. The behaviour of equality with these often has nice characterizations. We will present pairs and dependent pairs.

Task 4 (Equality between pairs)
A characterization of equality in pair types is the following: two pairs are equal if and only if they are equal component-wise. Prove it by giving two functions with the signatures presented below. All of this can be done by case splitting on the equality proofs, but you are encouraged to instead try and manipulate the terms in scope. You will also probably need the function `cong2` defined below. This can be seen as congruence for functions taking two arguments. You don't have to copy `cong2` in your script, it is already provided in the prelude.

```
-- cong2 : (f : a -> b -> c) -> {x, x' : a} -> {y, y' : b}
--      -> x = x' -> y = y' -> f x y = f x' y'
-- cong2 f {x} {x' = x} Refl = cong (f x)


And = Pair

eq_pair : x = x' -> y = y' -> (x, y) = (x', y')

eq_pair_iff : {p1, p2 : Pair a b}
    -> p1 = p2 -> (fst p1 = fst p2) `And` (snd p1 = snd p2)
```

Task 5 (Associativity of addition for naturals)
And now, a proof by induction. Let us prove that addition is associative for naturals. You are encouraged to experiment and see that going by induction on different arguments results in proofs that are more or less straightforward.

```
plus_assoc : (m, n, p : Nat) -> (m + n) + p = m + (n + p)
```

Task 6 (Length of lists)
Induction is not only useful to prove facts about natural numbers, but also about other inductive structures, like lists. Let us prove that the length of the concatenation of two lists is the sum of the lengths of the two lists.

```
len_concat : (l1, l2 : List a)
    -> length (l1 ++ l2) = length l1 + length l2
```

Task 7 (Optional: Preorder reasoning workout)
Recall our proof of commutativity of addition for the naturals:

```
plus_0 : (n : Nat) -> n = n + 0
plus_0 0 = Refl
plus_0 (S n) = cong S $ plus_0 n

plus_S : (m, n : Nat) -> S (m + n) = m + (S n)
plus_S 0 n = Refl
plus_S (S m) n = cong S $ plus_S m n

plus_comm : (m, n : Nat) -> m + n = n + m
plus_comm 0 n = plus_0 n
plus_comm (S m) n =
    let
```

```
        IHm = plus_comm m n
    in Calc $
        |~ S (m + n)
        ~~ S (n + m) ...( cong S IHm )
        ~~ n + S m   ...( plus_S n m )
```

Now let's use preorder reasoning to prove that multiplication distributes over addition on the left. The skeleton of the proof is already there, just provide the steps.

```
plus_mult_distrib_l : (m, n, p : Nat)
    -> m * (n + p) = m * n + m * p
plus_mult_distrib_l 0 n p = Calc $
    |~  0 * (n + p)
    ~~  0       ...( ?step0Z )
    ~~  0 + 0     ...( ?step1Z )
    ~~  0 * n + 0 * p ...( ?step2Z )
plus_mult_distrib_l (S m) n p = Calc $
    |~  S m * (n + p)
    ~~  (n + p) + m * (n + p)   ...( ?step0S )
    ~~  (n + p) + (m * n + m * p) ...( ?step1S )
    ~~  n + (p + (m * n + m * p)) ...( ?step2S )
    ~~  n + ((p + m * n) + m * p) ...( ?step3S )
    ~~  n + ((m * n + p) + m * p) ...( ?step4S )
    ~~  n + (m * n + (p + m * p)) ...( ?step5S )
    ~~  (n + m * n) + (p + m * p) ...( ?step6S )
    ~~  S m * n + S m * p     ...( ?step7S )
```

Task 8 (Guided: `List a` is a monoid)
The `Monoid` interface we presented in lecture 8 does not include the laws that characterise monoids in abstract algebra. That is, associativity of `<+>` and neutrality of `neutral` with respect to `<+>`. Having equality types now allows us to state and prove these laws in Idris.

The standard library provides a stricter interface for monoid, called `MonoidV`, which can be imported from module `Control.Algebra`. This module is not in the `base` package of the

standard library, it can be found in the `contrib` package: in order to import it, we must tell Idris that we want to access this package by passing the `-p contrib` flag to the compiler. This can also be set up in VS Code extension settings, to make it work with the IDE integration.

We will see together how to instantiate the interface for `List a`, therefore proving that the laws are respected:

```
-- import Control.Algebra
MonoidV (List a) where
  -- provide an instance
```