Lab 14

Functional Programming (ITI0212)

2022-04-29

Task 1
As a warm-up, prove congruence for functions of 2 arguments:

```
congruence2 : (0 f : a -> b -> c)
            -> (p : x = y)
            -> (q : u = v)
            -> f x u = f y v
```

Hint: As always, *when in doubt, case-split the arguments and see how the goal changes.* The proof is a one-liner, but you can reuse `congruence` from the lecture, if you like.

Both lemmata can be found in `Prelude` as `cong` and `cong2`, respectively.

More properties of addition

Please fill in the holes in the propositions given below. You can make use of all propositions proved during the lecture, in particular `symmetry`, `congruence`, `transitivity` (or their `Prelude`-versions `sym, cong, trans`) and

```
%hint
plusZeroRightNeutral : (n : Nat) -> n + 0 = n
plusZeroRightNeutral 0 = Refl
plusZeroRightNeutral (S k) = congruence S $
    plusZeroRightNeutral k
```

If you annotate a definition with `%hint`, *expression search* in your editor will attempt to use that definition to fill holes.

Task 2 (Addition and successors)
Show that $1 + (m + n) = m + (1 + n)$. Here's the code; fill in the holes:

```
%hint
plusSuccRightSucc : (m , n : Nat) -> S (m + n) = m + S n
plusSuccRightSucc 0 n = ?base_case
plusSuccRightSucc (S k) n = ind_step where
  ind_hyp : S (k + n) = k + (S n)
  ind_hyp = ?ind_hyp_proof

  ind_step : S (S (k + n)) = S (k + (S n))
  ind_step = ?ind_step_proof
```

Although this sounds trivial, it is not a proof by `Refl`. The reason is again that addition is defined by induction on the first argument, and `m + (S n)` does not immediately match a case in that definition.

Hint: An application of congruence will turn one of the goals into a one-liner.

Task 3 (Commutativity of addition)
Show that addition of natural numbers is commutative.

```
%hint
plusComm : (m , n : Nat) -> m + n = n + m
plusComm 0 n = ?base_case'
plusComm (S k) n = goal where
  ind_hyp : k + n = n + k
  ind_hyp = ?ind_hyp_proof'

  step1 : S (k + n) = S (n + k)
  step1 = ?step1_proof

  step2 : S (n + k) = n + (S k)
  step2 = ?step2_proof

  goal : S (k + n) = n + (S k)
  goal = ?goal_proof
```

Hint: Looking at its type declaration, `goal` is a combination of `step1` and `step2`. Transitivity of equality can help you to chain the two statements together.

Proving equality with *preorder reasoning*

Writing proofs is a matter of style, just like programming is a matter of style. While there are many style guides for code, what constitutes a "good style" of formalized proofs is (arguably) less well explored. At least when proving equalities, the style of *preorder reasoning* has been developed as a reasonable compromise between succinctness, repetition and readability.

Let us look again at the proof of `plusComm` in task 3. We notice that in a *pen-and-paper* proof, the combination of steps 1 and 2 would probably be written as

$$\begin{aligned} \mathrm{S}(k+n) &= \mathrm{S}(n+k) && (1)\\ &= n + \mathrm{S}(k), && (2) \end{aligned}$$

where step (1) and (2) would receive some written justification.

Preorder reasoning tries to emulate this style of proof. To use it, import the module `Syntax.PreorderReasoning` from the package `contrib`, which is distributed along with Idris. To use it in the REPL, either tell Idris on the command line:

```
$ idris2 --package contrib /path/to/Lab14.idr
```

...or add a package file to your project.

The module provides operators `(|~)`, `(~~)` and `(...)` to build a well-typed *derivations*, which can be converted into a proof of equality using the function `Calc`. Using those, `plusComm` turns into

```
plusComm' : (m , n : Nat) -> m + n = n + m
plusComm' 0 n = Calc $
  |~ 0 + n ~~ n      ...(Refl)
           ~~ n + 0  ...(?base_case_again)
plusComm' (S k) n = Calc $
  |~ S (k + n) ~~ S (n + k)  ...(?step1)
               ~~ n + (S k)  ...(?step2)
```

Task 4 (Working with preorder reasoning)
Copy the above definition and fill in the holes. When inspecting the context, you should see that the intermediate steps remain the same, but there is no need anymore for the last "combining" step.

Try searching for proofs using your editor integration. Since we marked some propositions with `%hint`, Idris should find proofs for all of the holes by itself.

Prefixes of lists, revisited

During lecture 13, we showed that being a prefix of a list is a reflexive (`isPrefixRefl`) and transitive (`isPrefixTrans`) relation. A relation with these two properties is called a preorder

Task 5 (Lists are ordered by prefixes)
Proof that `isPrefix` is anti-symmetric:

```
isPrefixAntisymm : IsPrefix xs ys -> IsPrefix ys xs
                -> xs = ys
```

With this third property, being the type of lists becomes *partially ordered* by the prefix-relation.

```
%hint
consCong : {0 xs, ys : List a} -> xs = ys
        -> x :: xs = x :: ys
consCong Refl = Refl
```

Hint: When in doubt, case-split the arguments. If your stuck in the induction step, the following lemma might be helpful:

Task 6
Have have a look at the documentation for generic preorder reasoning. Can you make `isPrefix` work with that? You will need to implement the interface `Preorder`, which can be found in the module `Control.Order`. Can you come up with some property of `isPrefix`, and then prove it using this style of reasoning?