Lab 15

Functional Programming (ITI0212)

2023-05-12

This week we are learning about decidability and automation in Idris programming.

A *decision procedure* for a predicate is an algorithm that for each possible argument either produces a *proof* that the predicate holds or else a *refutation* proving that it does not. In Idris the type constructor for decidability is called `Dec` with constructors `Yes` and `No`. Additionally, there is an interface for types with decidable equality called `DecEq` in the standard library module `Decidable.Equality`.

A *constraint argument* (also called an *auto-implicit argument*) is an implicit argument used to ensure that some validity condition is satisfied. It is written using the double-shafted arrow `=>` and is intended to be found by Idris's *expression search* mechanism, which is the same tool that is invoked for expression search by the editor integration. By default, expression search will try using context variables, constructors, function literals, recursion, and interface implementations in order to find a term of the desired type. You can augment this list by specifying additional terms for it to try using the `%hint` directive.

The first few tasks will illustrate why constraint arguments are useful and guide you through the process of using them.

Task 1
Suppose that we are hired by a bank to write the software for withdrawing money from an automat. As part of this program we need to subtract the amount withdrawn from the customer's bank balance.

Write a subtraction function to perform this task:

```
subtract_uf  :  Nat -> Nat -> Nat
```

Task 2
What happens if a customer tries to withdraw more money than they have? We could change the type to allow a negative balance, but if the bank doesn't trust the customer it may not want to let them withdraw more money than they have in their account. Another option is to use `Maybe` to only allow a withdrawal if it does not cause an underflow.

Rewrite the subtraction function using a `Maybe` type so that only valid withdrawals are allowed:

```
subtract_maybe  :  Nat -> Nat -> Maybe Nat
```

Task 3
This is certainly a valid solution. But it means that the rest of our banking program will have to use `case` or `map` to deal with our `Maybe` subtraction. Another option is to use `Fin` in the argument instead of `Maybe` in the result.

Rewrite the subtraction function using a `Fin` type so that only valid withdrawals are allowed:

```
subtract_fin  :  (m : Nat) -> (n : Fin ?what) -> Nat
```

Task 4
Again, this is a perfectly valid solution, but now we must deal with `Fin` types, which is less convenient than `Nat`. We could do our subtraction using two ordinary `Nat`s, provided that we could *prove* that the second one was less than or equal to the first. We can do this using `Data.Nat.LTE`.

Rewrite the subtraction function using `LTE` so that only valid withdrawals are allowed:

```
subtract_lte  :  (m : Nat) -> (n : Nat) -> (inbounds : n `LTE` m) -> Nat
```

Task 5
It is annoying to have to have to provide a proof of $n \leq m$ every time we want to do a subtraction $m - n$. We can ask Idris to try to find this proof for us using its search mechanism.

Rewrite the subtraction function using a *constraint argument* to ensure only valid withdrawals are allowed:

```
subtract_cst  :  (m : Nat) -> (n : Nat) -> (inbounds : n `LTE` m) => Nat
```

For example:

```
Lab15> 2 `subtract_cst` 0
2
Lab15> 2 `subtract_cst` 1
1
Lab15> 2 `subtract_cst` 2
0
Lab15> 2 `subtract_cst` 3
Error: Can't find an implementation for LTE 3 2.
```

Hint: There are (at least) two ways to write this function. One involves bringing the constraint argument into scope by explicitly binding it. Another involves writing and `%hint`ing a lemma to prove that the constraint is satisfied.

Task 6
This less than or equal to predicate looks pretty decidable.

Write a *decision procedure* for `LTE`:

```
decide_lte  :  (n , m : Nat) -> Dec (n `LTE` m)
```

Task 7
Show that if types `a` and `b` each have decidable equality then so does the type `Pair a b`:

```
implementation  DecEq a => DecEq b => DecEq (Pair a b)  where
```

Task 8 (optional challenge)
We learned that in intuitionistic logic the law of the excluded middle, `p `Or` Not p`, is *not* provable. But in homework 4 you showed that its double negation, `Not (Not (p `Or` Not p))`, *is* provable. This implies that `Not (p `Or` Not p)` is not provable either, because otherwise we would have a contradiction. Since we can prove neither `p `Or` Not p` nor `Not (p `Or` Not p)` we can't prove that `p `Or` Not p` is decidable. However, we *can* prove that it is not not decidable!

Prove that it is not the case that the law of the excluded middle is not decidable:

```
nndec_lem  :  Not (Not (Dec (p `Or` Not p)))
```

Hint: this proof is very similar to the one the one for `not_not_lem` in homework 4. It is not long or hard, but it is rather counterintuitive.