Lab 6

Functional Programming (ITI0212)

2023-03-10

This week we are learning about totality for data and codata. An expression that is *total* is safe in the sense that trying to evaluate it won't cause our program to crash or hang.

Total expressions must be *covering*, in the sense that they must be able to handle all possible cases that can arise. If a function is not covering then it will cause our program to crash if we ever use one of the missing cases. Idris checks coverage algorithmically, and considers coverage failure to be a type error. In addition to coverage, totality requires one additional condition to ensures that our programs won't hang.

For inductive types this condition is *termination*: evaluation must finish in finite time. Unlike coverage, termination cannot be decided algorithmically due to the unsolvability of the halting problem. So Idris makes the following conservative approximation: it accepts as terminating a function whose argument type is inductive if every recursive call is on a proper subterm.

For coinductive types the additional condition is *productivity*: evaluation must reach a constructor form in finite time. Unlike with inductive types, we don't evaluate the recursive arguments to constructors of coinductive types because they may be infinite. Like termination for inductive types, productivity for coinductive types is not algorithmically decidable. So Idris makes the following conservative approximation: it accepts as productive an expression whose result type is coinductive if every recursive occurrence is guarded by a constructor (and thus by an implicit or explicit `Delay`).

Task 1
Write a function that returns a colist containing the same elements in the same order as the argument list.

```
coL  :  List a -> Colist a
```

Write your definition so that you and Idris agree that it is total.

Note: The type constructor `Colist` is in the standard library, but you need to import the module `Data.Colist` in order to use it.

Task 2
Write a function that returns a list containing the same elements in the same order as the argument colist.

```
uncoL  :  Colist a -> List a
```

No function that satisfies this specification can be total—why not? Write an expression `uncoL ?sequence` that does not produce a result in finite time.

Task 3
Write the following function that computes the length of a colist.

```
length  :  Colist a -> Conat
```

Write your definition so that you and Idris agree that it is total. Why does the result type need to be `Conat` rather than `Nat`?

Note: The coinductive type `Conat` is not (yet) in the standard library, but you can copy it from lecture 6.

Task 4
Write the `filter` function for colists, which keeps only those elements of the argument sequence that satisfy the given predicate:

```
filter  :  (a -> Bool) -> Colist a -> Colist a
```

No function that satisfies this specification can be total—why not? Write an expression `filter ?predicate ?sequence` that does not produce a result in finite time.

Task 5
Unlike for inductive types, the collection of constructors for a coinductive type need not have a "base case", i.e. a constructor without arguments of the type in question. The following coinductive type of infinite sequences, or "streams" is in the standard library.

```
data  Stream : (a : Type) -> Type  where
  (::)  :  a -> Inf (Stream a) -> Stream a
```

Write the function

```
unroll  :  (a -> a) -> a -> Stream a
```

so that `unroll f x` generates the infinite sequence `[x , f x , f (f x), ...]`.

For example:

```
Lab6> take 5 (unroll S 1)
[1, 2, 3, 4, 5]
```

where the finite prefix function `take : Nat -> Stream a -> List a` is in the standard library.

Write your definition so that you and Idris agree that it is total.

Task 6
Write a function that zips a stream with a list.

```
zipSL  :  (a -> b -> c) -> Stream a -> List b -> List c
```

For example:

```
Lab6> zipSL MkPair (unroll S 1) ['a' , 'b' , 'c' , 'd' , 'e']
[(1,'a') , (2,'b') , (3,'c') , (4,'d') , (5,'e')]
```

Task 7 (optional challenge)
The goal of this task is to define multiplication for conatural numbers:

```
mul  :  Conat -> Conat -> Conat
```

to complete the `Num` implementation from lecture 6:

```
implementation  Num Conat  where
  (+)  =  add
  (*)  =  mul
  fromInteger  =  coN . fromInteger
```

We will need to think carefully in order to do this in a way that is total.

First, let's stipulate that the following property of `Nat` multiplication should remain true for `Conat` multiplication:

$$m * n = 0 \quad \text{if} \quad m = 0 \text{ or } n = 0$$

This gives us two clauses of the definition of `mul`. In the remaining clause m and n are both successors. If you use the same strategy as in lab 2, task 5 you will discover that the definition is not total. You can test this by evaluating `infinity * infinity`. Nevertheless, this is pretty close to the right idea.

You will need to use some basic facts about algebra to rewrite this clause so that the recursive call to `mul` occurs within the scope the `Conat` constructor `Succ`. Don't worry if it's not an immediate subexpression: even though this is the syntactic condition that Idris uses to recognize totality, it is stronger than necessary. In fact, it suffices for the recursive call to occur within a total expression that is guarded by the constructor.

If you figure it out then you should be able to evaluate expressions like:

```
Lab6> uncoN (infinity * 0)
0
Lab6> uncoN (0 * infinity)
0
Lab6> uncoN (3 * 4)
12
Lab6> infinity * 2
Succ (Delay (add (mul (Succ (Delay infinity)) (Succ (Delay (coN 1))))
    (Succ (Delay (coN 0)))))
Lab6> infinity * infinity
Succ (Delay (add (mul (Succ (Delay infinity)) (Succ (Delay
    infinity))) (Succ (Delay infinity))))
```

where the expressions that you get for multiplying a successor by infinity will depend on the details of your definition, but should in any case be infinite (so applying `uncoN` to them will hang your interpreter).