Lab 7

Functional Programming

2024-03-29

This week we are learning about totality for data and codata. An expression that is *total* is safe to evaluate, in the sense that trying to do so won't cause our runtime 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 an error. In addition to coverage, totality requires another condition to ensures that our programs won't hang.

For inductive types the additional condition is *termination*: evaluation must finish in finite time. Unlike coverage, termination cannot be decided algorithmically, even in principle, due to the undecidability 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, Idris doesn't evaluate the 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 explicit or implicit `Delay`).

You can ask Idris to confirm that an expression is total according to its algorithm using the REPL command `:total`.

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 defined in the standard library module `Data.Colist`.

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? Inspect your definition to identify the point where totality can fail. Then write an expression `uncoL ?sequence` that does not produce a result in finite time. You can interrupt Idris's evaluation with `Control-C`.

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 type `Conat` is not (yet) in the standard library, but you can copy it from the lecture notes.

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? Inspect your definition to identify the point where totality can fail. Then write an expression `filter ?predicate ?sequence` that does not produce a result in finite time.

Task 5
Unlike for inductive types, the collection of element constructors for a coinductive type need not have a *base case* (constructor with no arguments of the type being defined) in order for the type to be inhabited. 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) , ...]`.

Make sure that Idris can see that your definition is total. Next, use this function to define the stream of natural numbers so that Idris can see that it is total too.

```
nats  :  Stream Nat
```

For example:

```
Lab7> take 10 nats
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

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

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

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

Python's standard library includes a function that zips together a given list with the list of its indices:

```
enumerate  :  List a -> List (Pair Nat a)
```

```
Lab7> enumerate ['a','b','c']
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Use the functions above to define this as a point-free one-liner:

```
enumerate  =  zipSL ?function ?stream
```

Task 7
The goal of this task is to define multiplication for the conatural numbers,

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

to complete the `Num` implementation from lecture:

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

As a first attempt, try defining multiplication for `Conat`s using the same structurally recursive strategy that you used to define multiplication for `Nat`s in lab 2.

This seems plausible, but unfortunately this definition will not be total, as you can see by evaluating `infinity * infinity`. So we will need to think carefully in order to come up with a definition that is total.

Let's stipulate that the following facts about `Nat` arithmetic should remain true for `Conat` arithmetic:

$$0 * n = 0 \qquad \text{and} \qquad m * 0 = 0$$

This gives us two base cases for a recursive definition of `mul`. In the remaining case both m and n are successors.

You will need to use some basic algebra to write this clause in such a way that the recursive call to `mul` occurs within the scope the `Conat` constructor `Succ`, and thus of a(n explicit or implicit) `Delay`. For now, don't worry if it's not an immediate subexpression: even though that is the syntactic condition that Idris uses to recognize totality, it is stricter 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:

```
Lab7> uncoN (infinity * 0)
0
Lab7> uncoN (0 * infinity)
0
Lab7> uncoN (3 * 4)
12
Lab7> 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 and an infinite number will depend on the details of your definition, but should in any case be infinite (so applying `uncoN` to them will hang your interpreter).