Lab 10

Functional Programming (ITI0212)

2022-04-01

This week we talked about data, codata and the notions of totality pertinent to each.

A function producing data is total if it is covering and terminating. Covering means there must be a pattern-match provided for every constructor of the input inductive type. Terminating means the function must finish computing within a finite time.

Idris can mechanically check coverage, but not termination (halting problem) so it uses a (syntactic) approximation to termination which is that recursive calls on inductive arguments must occur on proper subterms of those arguments.

Codata is potentially infinite data. We can build co(inductive) data types by using the `Inf` keyword to mark arguments of constructors as potentially infinite. To construct something of type `Inf a`, we use the `Delay` constructor, which prevents eager evaluation of arguments to functions/constructors which could otherwise cause non-termination. Idris can automatically insert `Delay`s for us.

A function producing codata is total if it is covering and productive. Productive means that the function will produce a non-empty finite prefix of a potentially infinite output in a finite time. Again this is undecidable, so Idris uses a syntactic approximation that can detect some but not all productive functions: recursive calls must be immediate subterms of value constructors for codata (and so guarded by a `Delay`).

A function that is not total (in either sense above) is called partial. Partial functions may cause crashes (non-covering) or hangs (non-productivity or non-termination) at runtime, so it's good to know when your functions are total! However, there are some limitations to Idris' totality checker, and human ingenuity may be required.

Task 1
Write the addition function for `CoNat`s:

```
add : CoNat -> CoNat -> CoNat
```

Make sure that Idris recognizes it as total.

Note: You will need to copy the definition of `CoNat` from Lecture 10.

Task 2
Write the multiplication function for `CoNat`s:

```
mul : CoNat -> CoNat -> CoNat
```

You can assume that $n \times 0 = 0 = 0 \times n$ for any `CoNat` n. Your function does not need to be total, we will return to this in Task 7.

Task 3
Write the bounded subtraction function for `CoNat` (bounded in the sense that subtracting from `Zero` should yield `Zero`)

```
minus : CoNat -> CoNat -> CoNat
```

No definition of the bounded subtraction function for `CoNat` can be total, why not?

Write a term, `minus ?conat1 ?conat2` that will not yield a result in any finite amount of time.

Task 4
Write the length function for `Colists`:

```
length : Colist a -> CoNat
```

Make sure that Idris recognizes it as total.

Task 5
Write the filter function for `Colists`:

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

No definition of filter for `Colist` can be total, why not?

Write a term `filter ?pred ?stream` that will not yield a term in a finite time.

Task 6
The *hailstone function* `h : Integer -> Integer` is $\frac{n}{2}$ if n is even, and $3n + 1$ if n is odd.

```
h : Integer -> Integer
h n = case (n `mod` 2 == 0) of
           True => n `div` 2
           False => (3 * n) + 1
```

The *hailstone sequence* for a given n is the sequence `[n, h n, h (h n), ...]`, terminating if we reach 1 (or 0, in the case $n = 0$).

Implement the hailstone sequence

```
hail : Integer -> Colist Integer
```

It is currently unknown whether this sequence is finite for every n (Collatz conjecture)!

Your function should be total: why does this not mean that it solves the Collatz conjecture?

Task 7
Rewrite the `mul` function from Task 2 to be total. Your function will need at least three cases.

Idris may not recognize it as total, because likely your recursive calls will be not directly guarded by a coinductive value constructor, but rather the coinductive value constructor will guard a total function of the recursive call - but we know that this preserves totality.

If your function is total, `mul infinity infinity` should evaluate in finite time.