Lab 4: Streams

October 23, 2016

Contents

Objectives

By the end of this lab, you will:

- understand the two major purposes of functions.
- be able to represent data using functions.
- better understand higher order functions, and have practice writing them.

Note: This lab is adapted from *Programming and Programming Languages (2016): Chapter 9, Functions as Data.*

1 The Functions of Functions

People typically think of a function as serving one purpose: to parameterize an expression. While that is the most common use of a function, it does not justify having a function of no arguments, because that parameterizes over nothing at all.

Yet functions of no argument also have a use, because **functions actually serve two purposes**: to parameterize, and to suspend evaluation of a block of code until the function is applied. While we have covered parameterized expressions so far in this course, in this lab, we will be covering the latter use case: delay without abstraction.

2 Streams from Functions

So far, we've written programs that process and create *finite* sets, trees, and lists. However, there are many lists (or sequences) in nature that have no natural upper bound: from mathematical objects (the sequence of natural numbers) to natural ones (the sequence of hits to a Web site).
How might we represent an infinite list, like the sequence of natural numbers (e.g. 1, 2, 3, 4...)? Certainly, we could attempt to pick an arbitrary number of natural numbers as a representative set to use in our programs. For example, we could pick the first five starting at one:

```
link(1, link(2, link(3, link(4, link(5, empty)))))
```

Surely you can see why this might be a problem, no matter what our upper bound is. Some programs might want *more* values than the arbitrary amount we decide, and it would be a pain to try to guess or change that amount when programs start needing more than we provide. Instead of doing that, let's write a program to compute the sequence of natural numbers, so we don't need to pick an arbitrary end value:

```
    fun nats-from(n):
      link(n, nats-from(n + 1))
    end
```

Question: Does this program have a problem? (*Hint:* Try running it and see what happens!)

While the `nats-from` function represents our intent, it doesn't work: running it creates an infinite loop for every subsequent natural number. In other words, we want to write something very like the above, but that doesn't recur until we want it to, i.e., on demand. In other words, we want the rest of the list to be *lazy*.

This is where our earlier insight into functions comes in. A function, as we have just noted, delays evaluation of its body until it is applied. Therefore, a function would, in principle, defer the invocation of `nats-from(n + 1)` until it's needed.
Except, this creates a type problem: the second argument to link needs to be a list, and cannot be a function. Indeed, because it must be a list, and every value that has been constructed must be finite, every list is finite and eventually terminates in `empty`. Therefore, we need a new data structure to represent the `link`s in these *lazy lists* (also known as *streams*):

```
    data Stream<T>:
      | lz-link(h :: T, t :: ( -> Stream<T>))
    end
```

where the annotation `( -> Stream<T>)` means a function that consumes no arguments (hence the lack of anything before `->`), also known as a *thunk*. Note that the way we have defined streams they *must* be infinite, since we have provided no way to terminate them.
Now, we can construct a stream of constant values (the number 1 in this case):

```
rec ones = lz-link(1, lam(): ones end)
```

Note that the keyword `rec` is new. Since ones is not defined at the point of definition, when Pyret evaluates `lz-link(1, lam(): ones end)`, it complains that `ones` is not defined. However, it is being overly conservative with our former definition: the use of ones is "under a `lam`," and hence won't be needed until after the definition of `ones` is done, at which point `ones` will be defined.
Note that in Pyret, every `fun` *implicitly* has a `rec` beneath it, which is why we can create recursive functions with aplomb.

3 Putting the *fun* in Streams

Because functions are automatically recursive, when we write a function to create a stream, we don't need to use `rec`. Consider this example:

```
    fun nats-from(n): lz-link(n, lam(): nats-from(n + 1) end) end
```

with which we can define the natural numbers:

```
    nats = nats-from(0)
```

Note that the definition of `nats` is not recursive itself – the recursion is inside `nats-from` – so we don't need to use `rec` to define `nats`.

Task: Earlier, we said that every list is finite and hence eventually terminates. How does this remark apply to streams, such as the definitions of `ones` and `nats` above? Discuss your answer with your partner and jot down some notes.

⋆ **Before continuing, call over a TA to check that your answer is right.**

4 Seeing Streams

We know how to create streams, like `nats` and `ones`, but we still don't have an easy way to "see" one.

Task: Using the `Stream<T>` definition shown above in Pyret, write a function that returns the `h` field, or the "head" of the Stream:

```
    fun lz-first<T>(s :: Stream<T>) -> T
```

Task: Using the same data definition, write a function that returns the "rest" of a Stream:

```
    fun lz-rest<T>(s :: Stream<T>) -> Stream<T>
```

The functions you wrote are useful for examining individual values of a stream. It is also useful to extract a finite prefix of it (of a given size) as a (regular) list, which would be especially handy for testing.
Task: Write a function that will return a list of the first n elements as given by a Stream. Don't forget to write tests – see `ones`, `nats`, and `nats-from` above for good Streams to test with!

```
    fun take<T>(n :: Number, s :: Stream<T>) -> List<T>
```

⋆ **Before continuing, call over a TA to check that your `lz-first`, `lz-rest`, and `take` functions are correct.**

5 Streams - More Examples

Task: Define a Stream `tenths` of negative powers of 10. For example, the following test should pass:

```
      take(3, tenths) is [list: 1/10, 1/100, 1/1000]
```

Task: Define a function `pow2-from(n)` that creates a stream of powers of two starting from any value. Use this to create streams with powers of two starting from 1, 2, and 3.

Task: Define a function `onelessdouble-from(n)` that creates a stream of values one less than double the previous value, starting from any value. For example, $(1, 1, 1...)$ would be the values starting from 1, but $(2, 3, 5, 9, 17)$ would be the values starting from 2. Be sure to test with several starting values!

⋆ **Before continuing, call over a TA to check that your work so far.**

6 Higher Order Functions of Streams

Hopefully you're beginning to understand why streams can be useful. However, since they're so similar to lists, wouldn't it make sense to have list-like higher-order functions that we could use for streams?
Below, we show an example of `map2`, a version of `map` that takes two lists and applies

the first argument to them pointwise. Of course, this example is applied to `lz-link`s, not normal lists:

```
fun lz-map2<A, B>(f :: (A, A -> B),
                  s1 :: Stream<A>, s2 :: Stream<A>)
    -> Stream<B>:
  lz-link(f(lz-first(s1), lz-first(s2)),
    lam(): lz-map2(f, lz-rest(s1), lz-rest(s2)) end)
end
```

Task: Define the equivalent of `map` for Streams:

```
fun lz-map<A, B>(f :: (A -> B), s :: Stream<A>) -> Stream<B>
```

Task: Define the equivalent of `filter` for Streams:

```
fun lz-filter<A, B>(f :: (A -> boolean), s :: Stream<A>) -> Stream<A>
```

Task: Define the equivalent of `fold` for Streams:

```
fun lz-fold<A, B>(f :: (A, B -> A), base :: A, s :: Stream<B>) -> Stream<A>
```

Just for fun: Define a Stream `fib` that uses `lz-map2` that lets us get as many Fibonacci numbers as we want.

⋆ **Once a lab TA signs off on your work, you've finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you've just written.**