Lecture 34: Streams and Lazy Evaluation

Some of the most interesting real-world problems in computer science center around sequential data.

- DNA sequences.
- Web and cell-phone traffic streams.
- The social data stream.
- Series of measurements from instruments on a robot.
- Stock prices, weather patterns.

Finite to Infinite

Currently, all our sequence data structures share common limitations:

- Each item must be explicitly represented, even if all can be generated by a common formula or function
- Sequence must be complete before we start iterating over it.
- Can't be infinite. Who cares?
  - "Infinite" in practical terms means "having an unknown bound".
  - Such things are everywhere.
  - Internet and cell phone traffic.
  - Instrument measurement feeds, real-time data.
  - Mathematical sequences.

Iterators

- We've already seen an alternative to lists and tuples: iterators.
- Review: In Python, an iterator is an object that defines:
  - `I.__next__()`, which yields the next element in sequence `I` or raises `StopIteration`
  - `I.__iter__()` (optionally), which simply returns `I`. (This is so that either lists or iterators can be used in **for**).
- Anything that wants to be iterated over by **for** can supply an `__iter__` method to create an iterator.
- Crucial point: Iterators don't compute items in a sequence until they are asked to. They are *lazy* (a technical term!).

Generators: Another Kind of Iterator

- Generators provide a concise and elegant way to write iterators.
- Example: *generator returning lists* [0], [0, 1], [0, 1, 2], . . .

```
def triangle(n):
    """Generates all lists of the form [0], [0,1], ...,
    up to [0,...n-1]."""
    L = []
    for i in range(0, n):
        L += [i]
        yield L

>>> for p in triangle(3):
...     print(p)
[0]
[0, 1]
[0, 1, 2]
```

Generators, explained

- A generator function is one that contains a **yield** statement.
- When called, a generator function returns a generator object.
- The generator object defines `__next__`, and acts like an iterator.
- When called, this `__next__` function executes the body of the generator up to the next call to **yield** and then returns the result.
- On each subsequent call, starts from after the **yield** statement.
- Stops iterating on exit from the generator function.

Streams: Another Lazy Structure

We'll define a *Stream* to look like an rlist whose `rest` is computed lazily.

```
class Stream(object):
    """A lazily computed recursive list."""

    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

    @property
    def rest(self):
        assert not self.empty, 'Empty streams have no rest.'
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
        return self._rest
empty_stream = Stream(None, None, True)
```

Example: The positive integers (all of them)

```
def make_integer_stream(first=1):
    """An infinite stream of increasing integers, starting at FIRST.
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)

>>> ints = make_integer_stream(1)
>>> ints.first
1
>>> ints.rest.first
2
```

Mapping Streams

Familiar operations on other sequences can be extended to streams:

```
def map_stream(fn, s):
    """Stream of values of FN applied to the elements of stream S.
    if s.empty:
        return s
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first), compute_rest)

def combine_streams(fn, s0, s1):
    """Stream of the elements of S0 and S1 combined in pairs with
    two-argument function FN."""
    def compute_rest():
        return combine_streams(f, s0.rest, s1.rest)
    if s0.empty or s1.empty:
        return empty_stream
    else:
        return Stream(f(s0.first, s1.first), compute_rest)
```

Filtering Streams

Another example:

```
def filter_stream(fn, s):
    """Return a stream of the elements of S for which FN is true."""
    if s.empty:
        return s
    def compute_rest():
        return filter_stream(fn, s.rest)
    if fn(s.first):
        return Stream(s.first, compute_rest)
    return compute_rest()
```

A Few Conveniences

To look at streams a bit more conveniently, let's also define:

```
def truncate_stream(s, k):
    """A stream of the first K elements of stream S."""
    if s.empty or k == 0:
        return empty_stream
    def compute_rest():
        return truncate_stream(s.rest, k-1)
    return Stream(s.first, compute_rest)

def stream_to_list(s):
    """A list containing the elements of (finite) stream S."""
    r = []
    while not s.empty:
        r.append(s.first)
        s = s.rest
    return r
```

Finding Primes

```
def primes(pos_stream):
    """Return a stream of members of POS_STREAM that are not
    evenly divisible by any previous members of POS_STREAM.
    POS_STREAM is a stream of increasing positive integers.

    >>> p1 = primes(make_integer_stream(2))
    >>> stream_to_list(truncate_stream(p1, 7))
    [2, 3, 5, 7, 11, 13, 17]
    >>> p2 = primes(iterator_to_stream(positives()).rest)
    >>> stream_to_list(truncate_stream(p2, 7))
    [2, 3, 5, 7, 11, 13, 17]
    """
    def not_divisible(x):
        return x % pos_stream.first != 0
    def compute_rest():
        return primes(filter_stream(not_divisible, pos_stream.rest))
    return Stream(pos_stream.first, compute_rest)
```

Recursive Streams

What do you suppose we get from this?

```
f = Stream(1,
           lambda: Stream(1,
                          lambda: combine_streams(add, f, f.rest)))
stream_to_list(truncate_stream(f, 20))
```