Lecture 27: 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.
Review: Iterators

• The Python `for` loop

```python
for x in L:
    BODY
```

can use one of two strategies:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>`_ITER = L.<strong>iter</strong>()</td>
<td>`_I, _L = 0, L</td>
</tr>
<tr>
<td>while True:</td>
<td>while True:</td>
</tr>
<tr>
<td>try:</td>
<td>try:</td>
</tr>
<tr>
<td>x = _ITER.<strong>next</strong>()</td>
<td>x = _L[_I]</td>
</tr>
<tr>
<td>BODY</td>
<td>BODY</td>
</tr>
<tr>
<td>except StopIteration:</td>
<td>_I += 1</td>
</tr>
<tr>
<td>break</td>
<td>except IndexError:</td>
</tr>
<tr>
<td></td>
<td>break</td>
</tr>
</tbody>
</table>

• Crucial point: Iterators don’t compute items in a sequence until they are asked to. They are *lazy* (a technical term!).

Last modified: Wed Apr 5 13:47:18 2017
Streams: Another Lazy Structure

We'll define a Stream to look like an rlist (linked list) whose rest is computed lazily.

class Stream(object):
    """A lazily computed recursive list."""
    def __init__(self, first, compute_rest=lambda: Stream.empty):
        """A Stream whose first element is FIRST and whose tail is initialized from COMPUTE_REST() when needed."""
        self.first, self._compute_rest = first, compute_rest

@property
    def rest(self):
        """Return the rest of the stream, computing it once."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest

def __repr__(self):
    return 'Stream({0}, <...>)'.format(repr(self.first))

empty_stream = ... # Some object representing an empty stream
Basic Stream Operations

```python
>>> s1 = Stream(1, lambda: Stream(2))
>>> s1.first
1
>>> s1.rest.first
2
>>> s1.rest.rest
Stream.empty
```
Examples

An infinite stream of the same value.

```python
def make_const_stream(x):
    """An infinite stream of X’s.""
    return Stream(x, lambda: make_const_stream(x))
```

The positive integers (all of them)

```python
def make_integer_stream(first=1):
    """The infinite stream FIRST, FIRST+1, ...""
    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
```
Mapper Streams

Familiar operations on other sequences can be extended to streams:

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

def add_streams(s0, s1):
    """Stream of the sums of respective elements of S0 and S1.""
    def compute_rest():
        return add_streams(s0.rest, s1.rest)
    if s0 is Stream.empty or s1 is Stream.empty:
        return Stream.empty
    else:
        return Stream(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 is Stream.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()
Streams to Lists

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

```python
def stream_to_list(s, n):
    """A list containing the elements of stream S,
    up to a maximum of N.""
    r = []
    while n > 0 and s is not Stream.empty:
        r.append(s.first)
        s = s.rest
        n -= 1
    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.
    >>> p4 = primes(make_integer_stream(4))
    >>> stream_to_list(p4, 9)
    [4, 5, 6, 7, 9, 11, 13, 17, 19]
    >>> p2 = primes(make_integer_stream(2))
    >>> stream_to_list(p2, 9)
    [2, 3, 5, 7, 11, 13, 17, 19, 23]
    """
    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)
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

    def iterator_to_stream(iterator):
        """Returns a stream containing the values returned by ITERATOR."""

    ??
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

```python
def iterator_to_stream(iterator):
    """Returns a stream containing the values returned by ITERATOR.""
    
def compute_rest():
        ??

    return compute_rest()
```
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

```python
def iterator_to_stream(iterator):
    """Returns a stream containing the values returned by ITERATOR.""

    def compute_rest():
        return Stream(??)

    return compute_rest()
```
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

```python
def iterator_to_stream(iterator):
    """Returns a stream containing the values returned by ITERATOR.""
    
    def compute_rest():
        return Stream(next(iterator), ??)
    
    return compute_rest()
```
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

```python
def iterator_to_stream(iterator):
    """Returns a stream containing the values returned by ITERATOR.""
    def compute_rest():
        return Stream(next(iterator), compute_rest)
    return compute_rest()
```
Relationship of Streams to Iterators

- A stream is clearly very much like an iterator.
- The difference is that, in effect, it *remembers* its past values.

```python
def iterator_to_stream(iterator):
    """Returns a stream containing the values returned by ITERATOR."""

    def compute_rest():
        try:
            return Stream(next(iterator), compute_rest)
        except StopIteration:
            return empty_stream
    return compute_rest()
```
Recursive Streams

• Because streams are computed lazily, in a definition such as

\[
aStream = \text{Stream}(\ldots, \text{lambda}: \ldots)
\]

the body of the \texttt{lambda} can refer to \texttt{aStream} (because it will have been initialized by the time the lambda function is called.)

• So what do you suppose we get from these?

\[
c1 = \text{Stream}(1, \text{lambda}: c1)
\]
\[
\text{stream_to_list}(c1, 5)
\]

\[
f1 = \text{add_streams}(c1, \text{Stream}(0, \text{lambda}: f1))
\]
\[
\text{stream_to_list}(f1, 5)
\]

\[
f2 = \text{Stream}(1,
\quad \text{lambda}: \text{Stream}(1,
\quad \quad \text{lambda}: \text{add_streams}(f2, f2.\text{rest})))
\]
\[
\text{stream_to_list}(f2, 6)
\]
Recursive Streams

• Because streams are computed lazily, in a definition such as

\[ \text{aStream} = \text{Stream}(...) \text{, lambda: ...} \]

the body of the \texttt{lambda} can refer to \texttt{aStream} (because it will have been initialized by the time the lambda function is called.)

• So what do you suppose we get from these?

\[
\begin{align*}
c1 &= \text{Stream}(1, \text{\texttt{lambda: c1}}) \\
\text{stream_to_list}(c1, 5) &= [1, 1, 1, 1, 1] \\
\text{f1} &= \text{add_streams}(c1, \text{Stream}(0, \text{\texttt{lambda: f1}})) \\
\text{stream_to_list}(f1, 5) &= \\
\text{f2} &= \text{Stream}(1, \\
& \text{\texttt{lambda: Stream}(1,} \\
& \text{\texttt{lambda: add_streams}(f2, f2.rest))}) \\
\text{stream_to_list}(f2, 6) &=
\end{align*}
\]
Recursive Streams

• Because streams are computed lazily, in a definition such as

   \[
   \text{aStream} = \text{Stream}(..., \text{lambda}: \ldots)
   \]

   the body of the \text{lambda} can refer to \text{aStream} (because it will have been initialized by the time the lambda function is called.)

• So what do you suppose we get from these?

   \[
   \begin{align*}
   \text{c1} &= \text{Stream}(1, \text{lambda}: \text{c1}) \\
   \text{stream_to_list(c1, 5)} &= \{1, 1, 1, 1, 1\}
   \\
   \text{f1} &= \text{add_streams(c1, Stream(0, \text{lambda}: \text{f1}))} \\
   \text{stream_to_list(f1, 5)} &= \{1, 2, 3, 4, 5\}
   \\
   \text{f2} &= \text{Stream}(1, \\
   &\quad \text{lambda}: \text{Stream}(1, \\
   &\quad \quad \text{lambda}: \text{add_streams(f2, f2.rest)))) \\
   \text{stream_to_list(f2, 6)}
   \end{align*}
   \]
Recursive Streams

- Because streams are computed lazily, in a definition such as

  
  \[
  \text{aStream} = \text{Stream}(\ldots, \lambda: \ldots)
  \]

  the body of the \texttt{lambda} can refer to \texttt{aStream} (because it will have been initialized by the time the lambda function is called.)

- So what do you suppose we get from these?

  
  \[
  \begin{align*}
  \text{c1} &= \text{Stream}(1, \lambda: \text{c1}) \\
  \text{stream_to_list}(\text{c1}, 5) &= [1, 1, 1, 1, 1] \\
  \text{f1} &= \text{add_streams}(\text{c1}, \text{Stream}(0, \lambda: \text{f1})) \\
  \text{stream_to_list}(\text{f1}, 5) &= [1, 2, 3, 4, 5] \\
  \text{f2} &= \text{Stream}(1, \\
      \quad \lambda: \text{Stream}(1, \\
          \quad \lambda: \text{add_streams}(\text{f2}, \text{f2}.\text{rest}))) \\
  \text{stream_to_list}(\text{f2}, 6) &= [1, 1, 2, 3, 5, 8]
  \end{align*}
  \]