String matching

July 19, 2015

1 The scan left function

1.1 scan left

`scanl` is a polymorphic function of type $(\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$ for type variables, α and β. It is defined by the following equations.

$$
\begin{aligned}
scanl\ f\ z\ [] &= [z] \\
scanlf\ z\ (x : xs) &= z : scanl\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

In C++, lists are replaced with iterators.

```
template <
  class F, class AccT, class InT, class OutT>
OutT scan_left (F f, AccT z, InT begin, InT end, OutT out) {
  *out++ = z;
  if (begin == end) return out;
  auto const& x = *begin;

  return scan_left (f, f (z, x), std::next (begin), end, out);
}
```

2 The string matching problem

2.1 matches

A string matching problem is one in which one finds all occurrences of a non-empty string (the pattern) in some other string (the text). A specification for the problem can be stated like this:

$$matches\ ws = map\ len \cdot filter\ (endswith\ ws) \cdot inits \tag{1}$$

The function $inits$ returns a list of the prefixes of the text in order of increasing length. The expression $endswith\ ws\ xs$ tests whether the pattern ws is a suffix of xs. Finally, the value $matches\ ws\ xs$ is a list of integers p such that ws is a suffix of $take\ p\ xs$. For example: $matches$ "$abcab$" "$ababcabcab$" is the list $[7, 10]$. That is, $matches\ ws\ xs$ returns a list of integers p such that ws appears in xs ending at position p (counting positions from 1).

2.2 filter

In C++, $filter$ can be written like this.

```
template <class PredT, class RngT, class OutT>
OutT filter (PredT p, RngT xs, OutT out) {
  return std::accumulate (
      std::begin (xs), std::end (xs), out,
      [&p](auto dst, auto const& x) {
           return p (x) ? *dst++ = x : dst;
      }
  );
}
```

2.3 endswith

We define $endswith\ ws = (reverse\ ws \sqsubseteq) \cdot reverse$ where $\sqsubseteq$ is the $prefix$ relation given by the equations

$$\begin{aligned} [\,] &\sqsubseteq us = true \\ (u : us) &\sqsubseteq [\,] = false \\ (u : us) &\sqsubseteq (v : vs) = (u = v \wedge us \sqsubseteq vs) \end{aligned}$$

Here's the $prefix$ function realized in C++.

```
template <class InT1, class InT2>
bool prefix (InT1 lb, InT1 le, InT2 rb, InT2 re) {
  if (lb == le) return true;
  if (rb == re) return false;

  return *lb == *rb &&
         prefix (std::next (lb), le, std::next (rb), re);
}
```

3 The Boyer-Moore solution

3.1 Theory

This identity is called "the scan lemma".

$$map\ (foldl\ op\ e) \cdot inits = scanl\ op\ e \tag{2}$$

This equation is important because the left hand side has complexity $O(N^2)$ whereas the right-hand-side, $O(N)$. It admits restating equation 1 as:

$$\begin{aligned} matches\ ws &= map\ fst \cdot filter((sw \sqsubseteq) \cdot snd)\ scanl\ step\ (0, []) \\ sw &= reverse\ ws \\ step\ (n, sx)\ x &= (n+1, x : sx) \end{aligned} \tag{3}$$

This is the called the basic "Boyer-Moore" algorithm. That's what we'll implement in C++.

```
template <class OutT>
OutT matches (std::string const& ws, std::string const& s, OutT dst) {
  typedef std::pair<int, std::deque<char>> acc_t;

  auto step = [](acc_t p, char x) -> acc_t {
    ++p.first;
    p.second.push_front (x);

    return p;
  };

  std::deque<acc_t> buf;
  scan_left (
      step
    , std::make_pair(0, std::deque<char>())
    , s.begin ()
    , s.end ()
    , std::back_inserter(buf));

  std::string sw(ws.rbegin (), ws.rend ());
  auto pred = [&sw] (auto p) -> bool {
    return prefix (
      sw.begin (), sw.end ()
    , p.second.begin (), p.second.end ());
  };
  std::deque<acc_t> temp;
  filter (pred, buf, std::back_inserter (temp));

  return std::transform (
     temp.begin (), temp.end (), dst,
     [](acc_t const& p) -> int {  return p.first; });
}
```

4 Testing

Here's a little test driver.

```
int main () {

  std::list<int> where;
  matches ("abcab", "ababcabcab", std::back_inserter (where));

  std::for_each (where.begin (), where.end ()
   , [](int i) -> void { std::cout << i << ", "; }
   );

  return 0;
}
```

This program should print “7, 10”.