String matching

July 24, 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.

$$scanl\ f\ z\ [] = [z]$$
$$scanlf\ z\ (x : xs) = z : scanl\ f\ (f\ z\ x)\ xs$$

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

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

So, here's $prefix$.

```
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. It is not hard to see why this formula works. Translating to code, here's one implementation.

```
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

This code can be quickly tested with a program like the following.

```
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;
}
```

The program should print "7, 10" for this input.

5 War and Peace

5.1 Stack overflow

We can try our program in earnest by trying to find all occurence of the string "people" in the text of the book "War and Peace" by Tolstoy. When we do, we'll immediately realize the shortcomings of our program in the face of "large" inputs. the first problem is stack overflow. It would seem that the `scan_left` is not getting the tail-call optimization applied to it. We take matters into our own hands with this alternative implmentation.

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

5.2 Memory consumption

The stack overflow problem fixed then next challenge is to address impossible memory consumption. The solution here is to avoid copying data by referencing character ranges in the source string using iterators instead.

```
template <class OutT>
OutT matches (std::string const& ws, std::string const& s, OutT dst) {
  typedef std::string::const_reverse_iterator it;
```

```
  typedef std::pair<it, it> iterator_range;
  typedef std::pair<int, iterator_range> acc_t;

  std::size_t num_chars=s.size();

  auto step = [num_chars,&s](acc_t p, char x) -> acc_t {
    ++p.first;
    std::string::const_reverse_iterator rbegin = s.rbegin ();
    std::advance (rbegin, num_chars - p.first);
    p.second = std::make_pair (rbegin, s.rend ());

    return p;
  };

  std::deque<acc_t> buf1;
  scan_left (
      step
    , std::make_pair (0, std::make_pair (s.rend (), s.rend()))
    , s.begin ()
    , s.end ()
    , std::back_inserter (buf1));

  std::string sw(ws.rbegin (), ws.rend ());
  auto pred = [num_chars, &sw, &s] (auto p) -> bool {
    return prefix (sw.begin (), sw.end (), p.second.first, p.second.second);
  };

  std::deque<acc_t> buf2;
  filter (pred, buf1, std::back_inserter (buf2));
  buf2.swap (buf1);

  return std::transform (buf1.begin (), buf1.end (),
            dst, [](acc_t const& p) -> int {  return p.first; });
}
```

5.3 The "final" version

Now as we've realized references to the character ranges via iterators, we observe that this such references can be produced on demand. Accordingly the entire computation as a simple loop eliminates `scan_left`, `filter`, `transform` and lambda functions entirely!

```
template <class OutT>
OutT matches2 (std::string const& ws, std::string const& s, OutT dst) {
  std::string sw (ws.rbegin (), ws.rend ());
  std::size_t num_chars=s.size ();
```

```
  for (std::size_t i = 0; i < num_chars; ++i) {
    std::string::const_reverse_iterator rbegin = s.rbegin ();
    std::advance (rbegin, num_chars - i);
    if (prefix (sw.begin (), sw.end (), rbegin, s.rend ()))  {
      *dst++ = i;
    }
  }

  return dst;
}
```

This program is able to find the 582 occurrences of "people" in "War and Peace" in times typical of around $\frac{6}{100}$ of a second.

References

[1] Richard Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.