Sequence Processing

Rob Miller
Fall 2007
Review

Last time: higher-order functions

- A higher-order function takes a function as an argument or returns a function as its result
- Functional objects like UnaryFunction<T,U> and BinaryFunction<T,U,V>

Today: higher-order functions over sequences

- **map** applies a unary function to every element of a sequence
- **filter** removes elements that don’t satisfy a predicate
- **accumulate** combines elements using a binary function
- **generate** creates a sequence by repeatedly applying a unary function to a seed element
- The map/filter/accumulate programming pattern leads to high-level code that omits details about how the sequence is iterated
- It also enables easy concurrent and distributed evaluation

© Robert Miller 2007
Sequences

Seq<E> is a sequence of elements ∈ E

e.g. [1, 2, 3, 4] ∈ Seq<int>

Operations

[ ] denotes the empty sequence
add: E × Seq<E> → Seq<E> adds an element to the front of a sequence

e.g. add(a, [b, c, d]) = [a, b, c, d]
first : Seq<E> → E
rest : Seq<E> → Seq<E>

first ( [a, b, c] ) = a

rest ( [a, b, c] ) = [b,c]

Sequences are an abstract type

➢ List<E> is a canonical (and generic) implementation of Seq<E>
➢ But so is String (representing Seq<char>)
Map

Map applies a unary function to each element

- Returns a new sequence containing the results

\[
\text{map} : \text{Seq}<E> \times (E \rightarrow F) \rightarrow \text{Seq}<F>
\]

\[\text{e.g. map}( \ [1,2,3], \text{incr} ) = [\text{incr}(1), \text{incr}(2), \text{incr}(3)] = [2, 3, 4]\]

\[\text{map}( \text{“gLoB”}, \text{toLowerCase} ) = \text{“glob”}\]

- Let’s write \text{map()} as if it were an instance method of \text{Seq}, since that will make nested expressions easier to read from left to right:

\[\text{[1,2,3].map(incr).map(square)} = [4, 9, 16]\]

\[\text{map( seq, f ) = if (seq == [ ])} \text{ return [ ]}\]

\[\text{else return add( f( seq.first() ), map( seq.rest(), f ) )}\]
Filter tests each element with a unary predicate

- Elements that satisfy predicate are kept; those that don’t are removed

\[
\text{filter} : \text{Seq}\langle E \rangle \times (E \to \text{boolean}) \to \text{Seq}\langle E \rangle
\]

e.g. \([1,2,3].\text{filter}(\text{isOdd}) = [1,3]\)

“xy23ab9”.filter( isLetter ) = “xyab”

\[
\text{filter}(\text{seq}, p) = \\
\text{if } (\text{seq} == \text{[]}) \text{ return } \text{[]} \\
\text{else if } (p(\text{seq.first()})) \text{ return add( seq.first(), filter(seq.rest(), p))} \\
\text{else return filter(seq.rest(), p)}
\]
Accumulate

Accumulate combines elements using a binary function

- Also called reduce or fold
  
  \[ [1, 2, 3] \text{ . } \text{ accum }(+, 0) = 1 + 2 + 3 + 0 = 6 \]

- The result type doesn’t have to be the same as the element type
  
  \[ [1, 2, 4, 1] \text{ . } \text{ accum } (\text{ add }, \{\}) = \{1, 2, 4\} \]

- In general:

  \[ \text{ accum } : \text{ Seq} <E> \times (E \times F \rightarrow F) \times F \rightarrow F \]

**Two ways to accumulate: from the left or the right**

- Let’s assume \text{ accum } folds from the right

  - \text{ foldRight } : \text{ Seq} <E> \times (E \times F \rightarrow F) \times F \rightarrow F

  \[ \text{ foldRight}([1,2,3], -, 0) = 2 \]

- \text{ foldLeft } : F \times (F \times E \rightarrow F) \times \text{ Seq} <E> \rightarrow F

  \[ \text{ foldLeft}(0, -, [1,2,3]) = -6 \]
Map/Filter/Accumulate in Java

For immutable lists List<E>

interface List<E> {
    ...
    List<F> map (UnaryFunction<E,F> f);
    List<E> filter (UnaryFunction<E, Boolean> p);
    F  accum (BinaryFunction<E,F, F> op, F init);
}

For Strings

class StringSequences {
    static String mapToString (String s, UnaryFn<Character,Character> f) { ... }
    static List<F> mapToList (String s, UnaryFn<Character,F> f) { ... }
    static String filter (String s, UnaryFn<Character, Boolean> p);
    static F accum (String s, BinaryFn<Character,F, F> op, F init);
}
Other Kinds of Sequences

- The general pattern of map/filter/accumulate computation can be applied to other collections of elements as well.

Consumable sequences

- Iterator<E> is like a Seq<E>
  ```java
  static Iterator<F> map (Iterator<E> iterator, UnaryFn<E,F> f) { ... }
  ```
- A stream can be regarded as Seq<byte> or Seq<Line>
- Note that streams and iterators are state machines, which consume the sequence when it is read.

Sets

- A Set<E> can be put into an arbitrary order and treated as Seq<E>
- map(f) is set comprehension \{ f(x) | x in S \}
- filter(p) is subset \{x | x in S && p(x) \}
- accum(op,init) requires op to be insensitive to sequence order (so fold-left and fold-right are equivalent)
Real World Example

Searching for digital cameras

Camera data type has these observers:

- price : Camera → int
- brand : Camera → Brand
- pixels : Camera → int

Brand = {Kodak, Nikon, ...}

cameras : Seq<Camera> is the list of cameras for sale on Amazon

What’s the highest resolution found in a Nikon camera?

\[
\text{cameras.filter(brandIs(Nikon))}
\text{.map(pixels)}
\text{.accumulate(max, 0)}
\]


\[
\text{brandIs : Brand → (Camera → boolean)}
\text{brandIs(b) = \lambda c \mid c.brand() == b}
\]

max: int × int → int
sequence processing style

```java
import cameras.filter(brandIs(Nikon))
    .map(pixels)
    .accum(max, 0)
```

...as long as functionals are available and well-named

imperative style

```java
int result = 0;
for (Camera c : cameras) {
    if (brand(c) == Nikon) {
        result = max(c.pixels(), result)
    }
}
```

shows low-level details of control flow (loops and branches) and iteration of the sequence
Databases Also Use M/F/A Pattern

Most of the world’s data is probably stored in relational databases

- SQL (Structured Query Language) is the de facto standard language for querying relational databases
- A typical SQL query:

  ```sql
  select max(pixels) from cameras where brand = "Nikon"
  ```
Generate

**map/filter/accumulate seem incomplete**

> map and filter are producers, accum is an observer – where’s the creator?

**From an example**

> Recall the series() function from last lecture, which computed

\[ x \ op \ f(x) \ op \ f^2(x) \ op \ ... \ op \ f^{n-1}(x) \]

\[= [x, f(x), f^2(x), ..., f^{n-1}(x)] \ . \text{accum}(\text{op}, \text{init}) \]

> So here’s our creator:

\[
generate : E \times (E \rightarrow E) \times \text{int} \rightarrow \text{Seq}<E> \\
generate(x, f, n) = \text{if } (n == 0) \text{ return } [ ] \]

\[
\text{else return add}(x, generate(f(x), f, n-1))
\]

\[
generate(1, incr, 3) = [1, incr(1), incr(incr(1))] = [1, 2, 3] \\
generate(1, identity, 3) = [1, 1, 1]
\]
Extended M/F/A Example

Map/filter/accumulate is very useful for web-scale problems

- Enormous sequences stored on many computers
- Maps and filters can be run concurrently in parallel

Suppose we want to find palindromes on the Web

- A palindrome reads the same both forwards and backwards (ignoring punctuation, spaces, and capitalization)
  - A man, a plan, a canal – Panama!
  - A Toyota's a Toyota.
  - "Deer gas!" I disagreed.
  - Don't nod.
Palindromes

isPalindrome : String → boolean

isPalindrome(s) =
    String t = s.filter(isLetter).map(toLowerCase);
    return t.equals(t.reverse())

isPalindrome(“Mom”) == true
isPalindrome(“Don’t nod”) == true
isPalindrome(“blob”) == false

Suppose we have a sequence of words represented as Seq<String>
[“Hi”, “Mom!”, “Don’t”, “nod.”]

How do we find all the palindromic phrases that appear in the sequence?
- A palindromic phrase is a sequence of words that (when concatenated) is a palindrome
- e.g., “Mom!” “Don’t nod.”
Getting Phrases from a Sequence

We need to produce all possible phrases of a sequence

\[
\text{abc} \Rightarrow \text{a \ ab \ abc \ b \ bc \ c}
\]

Step 1: produce all the **suffixes** of the sequence (including the seq itself)

\[
\text{abc} \Rightarrow \text{abc \ bc \ c}
\]

Step 2: map over those suffixes, producing the **prefixes** of each suffix

\[
\text{abc \ bc \ c} \Rightarrow [\text{a \ ab \ abc}] \ [\text{b \ bc}] \ [\text{c}]
\]

Step 3: flatten Seq<Seq<Seq<E>>> to Seq<Seq<E>>

\[
[\text{a \ ab \ abc}] \ [\text{b \ bc}] \ [\text{c}] \Rightarrow \text{a \ ab \ abc \ b \ bc \ c}
\]

**phrases** : Seq<E> → Seq<Seq<E>>

\[
\text{phrases(s) = s.suffixes()}
\]

\[
\ .\text{map(prefixes)}
\]

\[
\ .\text{accum(append, [ ])}
\]

© Robert Miller 2007
Suffixes and Prefixes of a Sequence

suffixes : Seq<E> → Seq<Seq<E>>

\[ s.\text{suffixes}() = s.\text{accum}( (\lambda x, \text{suffs} | \text{add}(\text{add}(x, \text{suffs}.\text{first}()), \text{suffs})), [ [ ] ]) \]

\[ \text{e.g. } x = a \quad \text{suffs} = [ \text{bc} \quad \text{c} \quad [ ] ] \quad => \quad [\text{abc} \quad \text{bc} \quad \text{c} \quad [ ] ] \]

prefixes : Seq<E> → Seq<Seq<E>>

\[ s.\text{prefixes}() = s.\text{accum}( (\lambda x, \text{prefs} | \text{add}([x], \text{prefs}.\text{map}(\lambda t | \text{add}(x, t)))) , [ ] ) \]

\[ \text{e.g. } x = a \quad \text{prefs} = [ \text{b} \quad \text{bc} ] \quad => \quad [\text{a} \quad \text{ab} \quad \text{abc}] \]
Finding Palindromes on a Web Page

Page
text : Page → String

palindromes : Page → Seq<String>

palindromes(page) =
  page.text()  e.g. “Hi Mom! Don’t nod.”
     .split(‘ ‘)  e.g. [“Hi”, “Mom!”], “Don’t”, “nod.”]
     .phrases()  [ [“Hi”], [“Hi”, “Mom!”], …, [“Don’t”, “nod.”], [“nod.”]]
     .map( λ seq | seq e.g. [“Don’t”, “nod.”]
         .accum(join(' ',"")) “Don’t nod.”
     ) [“Hi”, “Hi Mom!”, …, “Don’t nod.”, “nod.”]
     .filter(isPalindrome) [“Mom!”, “Don’t nod.”]
Palindrome Finding Over Many Pages

web : Seq<Page>

web  
est [ p1, p2, p3, ... ]  
.map(palindromes)  
[ [ ], [ "Mom", "l" ], [ "Wow!", "a", "l" ], ... ]  
.accum(append, [ ] )  
[ "Mom", "l", "Wow!", "a", "l", ... ]  
.filter(longerThan(3))  
[ "Don’t nod.", ..., "Don’t nod.", ... ]  
.accum (add, { } )  
{ "Don’t nod.", ... }

This pattern (mapping then flattening) is so common that it’s often defined as
mapflat: Seq<E> x (E → Seq<F>) → Seq<F>

map and filter are special cases of mapflat. (And mapflat is actually a special case of accum!)

© Robert Miller 2007
Web Scale Palindrome Finding

Web is much too big for one computer to store and process

- $|\text{web}| = 10 \text{ billion pages} = 100 \text{ terabytes of data}$

Google does these kinds of computations using a cluster of computers

- A typical cluster consists of 1000 PCs
- web is divided into 1000 chunks of a few million pages (100GB) each
- Map & filter code is distributed to each PC and run independently

  ```
  \text{web1.mapflat(palindromes).filter(longerThan(3))}
  \text{web2.mapflat(palindromes).filter(longerThan(3))}
  ...
  ```

- Accumulate code can also be run in distributed fashion
- Further reading: Dean & Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004 (search for it in Google)

© Robert Miller 2007
Key Ideas of Symbolic Programming

Immutability: values that never change
- Makes programs simpler and safer (safe to share, to use concurrently, to copy over network)

Abstract data types
- type = set of abstract values
- operation = function over the type
- hidden representation specified by rep invariant and abstraction function

Common patterns of data types
- Tuples, options, lists, trees, functionals

Code as first-class data objects
- Interpreter: data represents a language expression
- Visitor & functional: data represents a function
- Higher-order functions and sequence processing (map, filter, accumulate)