Functional Programming for the Object Oriented

Øystein Kolsrud
19 November, 2015
Case study: The For-Loop

• Using direct indexing:

```c
for (int i=0; i<COUNT_OF(customerNames); i++)
{
    printf("%s\n", customerNames[i]);
}
```

• Using iterators:

```c
for (vector<int>::iterator it = customerNames.begin();
    it!=customerNames.end(); ++it)
{
    cout << *it << std::endl;
}
```

• Using foreach:

```c
foreach (var name in customerNames)
{
    Console.WriteLine(name);
}
```

• Using higher order function:

```c
customerNames.ForEach(Console.WriteLine);
```
Functions in Haskell

• Summarize elements of a list:

  ```haskell
  sumInts :: [Int] -> Int
  sumInts [] = 0
  sumInts (n:ns) = n + (sumInts ns)
  ```

• Concatenation of strings:

  ```haskell
  concatenate :: [String] -> String
  concatenate [] = ""
  concatenate (x:xs) = x ++ (concatenate xs)
  ```

• Common aspects of “concatenate” and “sumInts”:
  – There is a function to combine two values into one.
  – There is a base case.
  – There is a list of values.
The Fold Pattern

• Template for the fold pattern:
  
  ```haskell
  concatenate :: [String] -> String
  concatenate [] = ""
  concatenate (x:xs) = x ++ (concatenate xs)
  ```

• Implementing “fold”
  
  ```haskell
  fold :: (a -> a -> a) -> a -> [a] -> a
  fold _ base [] = base
  fold f base (x:xs) = f x (fold f base xs)
  ```
Using the Fold Pattern

• Concatenate using fold
  
  ```haskell
  concatenate :: [String] -> String
  concatenate xs = fold (++) "" xs
  ```

• Other applications of fold:
  
  ```haskell
  sum, product :: [Int] -> Int
  sum     ns = fold (+) 0 ns
  product ns = fold (*) 1 ns
  ```

  ```haskell
  and, or :: [Bool] -> Bool
  and bs = fold (&&) True bs
  or  bs = fold (||) False bs
  ```
Functional Patterns in Haskell

- Most iterative patterns already have predefined functions

- Common polymorphic higher order functions:

  ```
  map       :: (a -> b) -> [a] -> [b]
  concatMap :: (a -> [b]) -> [a] -> [b]
  filter    :: (a -> Bool) -> [a] -> [a]
  takeWhile :: (a -> Bool) -> [a] -> [a]
  all, any  :: (a -> Bool) -> [a] -> Bool
  zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
  iterate   :: (a -> a) -> a -> [a]
  foldr     :: (a -> b -> b) -> b -> [a] -> b
  foldl     :: (a -> b -> a) -> a -> [b] -> a
  ```
# Higher Order Functions in C# and Java

- **C#**: LINQ Introduced in .NET Framework 3.5 in 2007
- **Java**: Class `java.util.stream` Introduced in Java SE 8 in 2014

<table>
<thead>
<tr>
<th>Haskell</th>
<th>C#</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map f xs</code></td>
<td><code>xs.Select(f)</code></td>
<td><code>xs.map(f)</code></td>
</tr>
<tr>
<td><code>concatMap f xs</code></td>
<td><code>xs.SelectMany(f)</code></td>
<td><code>xs.flatMap(f)</code></td>
</tr>
<tr>
<td><code>filter p xs</code></td>
<td><code>xs.Filter(p)</code></td>
<td><code>xs.filter(p)</code></td>
</tr>
<tr>
<td><code>takeWhile p xs</code></td>
<td><code>xs.TakeWhile(p)</code></td>
<td></td>
</tr>
<tr>
<td><code>all p xs</code></td>
<td><code>xs.All(p)</code></td>
<td><code>xs.allMatch(p)</code></td>
</tr>
<tr>
<td><code>any p xs</code></td>
<td><code>xs.Any(p)</code></td>
<td><code>xs.anyMatch(p)</code></td>
</tr>
<tr>
<td><code>zipWith f xs ys</code></td>
<td><code>xs.Zip(f, ys)</code></td>
<td></td>
</tr>
<tr>
<td><code>foldl f x xs</code></td>
<td><code>xs.Aggregate(f, x)</code></td>
<td><code>xs.reduce(x, f)</code></td>
</tr>
<tr>
<td><code>iterate f x</code></td>
<td></td>
<td><code>Stream.iterate(x, f)</code></td>
</tr>
</tbody>
</table>
Case Study:
The Eight Queens Problem
Problem Definition

- Place 8 queens on a chessboard so that no two queens threaten each other.
Modelling the Problem

• What should a solution look like?

Solution: A list of row positions.

– Signature: solution :: [Int]
– Example: solution => [3,1,6,2,5,7,4,0]
Solution Strategy

• Strategy: Greedy Algorithm
  – Create all solutions and take the first.

• Where do we start?
  – Empty solution: []

• Where do we end?
  – A solution with eight queens has been found.

• What functions will we need to solve the problem?
  – We need to be able to extend an incomplete solution.
Extend Solution

• For a given partial solution, create all possible extensions.
• Current partial solution: [7,4,0]

- Possible extensions: [1,7,4,0], [5,7,4,0]
okToAdd

• Function signature:
  
  \[
  \begin{align*}
  \text{okToAdd} \; &:: \; \text{Int} \to [\text{Int}] \to \text{Bool} \\
  \end{align*}
  \]

• Examples:
  
  \[
  \begin{align*}
  \text{okToAdd} \; 0 \; [7,4,0] \; => \; \text{False} \\
  \text{okToAdd} \; 1 \; [7,4,0] \; => \; \text{True} \\
  \end{align*}
  \]

• Three cases:
  
  \[
  \begin{align*}
  \text{okToAddUp} \; &:: \; \text{Int} \to [\text{Int}] \to \text{Bool} \\
  \text{okToAddDown} \; &:: \; \text{Int} \to [\text{Int}] \to \text{Bool} \\
  \text{okToAddLevel} \; &:: \; \text{Int} \to [\text{Int}] \to \text{Bool} \\
  \end{align*}
  \]

• Function implementation:
  
  \[
  \begin{align*}
  \text{okToAdd} \; q \; qs \; = \; & (\text{okToAddUp} \; \; q \; qs) \; && \\
  & (\text{okToAddDown} \; \; q \; qs) \; && \\
  & (\text{okToAddLevel} \; \; q \; qs) \\
  \end{align*}
  \]
okToAddUp

- Strategy:
  - Create the diagonal set of numbers from row position.
  - Perform pairwise comparison between diagonal set and partial solution.
  - okToAddUp 1 [7,4,0]: Compare [2,3,4] with [7,4,0]
Implement `okToAddUp`

• Create the diagonal up:
  – Pattern:      iterate
  – Signature:    iterate :: (a -> a) -> a -> [a]
  – Example:      iterate succ 1 => [1,2,3,4,5,6...

• Perform pairwise comparison:
  – Pattern:      zipWith
  – Signature:    zipwith :: (a -> b -> c) -> [a] -> [b] -> [c]
  – Example:      zipWith (=/=) [2,3,4,5,6...] [7,4,0] => [True,True,True]

• Implementing `okToAddUp`:

  `okToAddUp :: Int -> [Int] -> Bool`
  
  `okToAddUp q qs = and $ zipWith (=/=) (tail $ iterate succ q) qs`
okToAddDown

• Follow a diagonal down and see if we hit a queen
  
  \[
  \text{okToAddUp} \ q \ qs = \text{and} \ (\text{zipWith} \ (=)) \ (\text{tail} \ (\text{iterate} \ \text{succ} \ q) \ qs)
  \]
  
  \[
  \text{okToAddDown} \ q \ qs = \text{and} \ (\text{zipWith} \ (=)) \ (\text{tail} \ (\text{iterate} \ \text{pred} \ q) \ qs)
  \]

• Higher order version
  
  \[
  \text{okToAddGivenDir} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{(Int} \rightarrow \text{Int}) \rightarrow \text{Bool}
  \]
  
  \[
  \text{okToAddGivenDir} \ q \ qs \ f = \text{and} \ (\text{zipWith} \ (=)) \ (\text{tail} \ (\text{iterate} \ f \ q) \ qs)
  \]

• New versions of up, and down:
  
  \[
  \text{okToAddUp} \ q \ qs = \text{okToAddGivenDir} \ q \ qs \ \text{succ}
  \]
  
  \[
  \text{okToAddDown} \ q \ qs = \text{okToAddGivenDir} \ q \ qs \ \text{pred}
  \]

• Implementing okToAddLevel
  
  \[
  \text{okToAddLevel} \ q \ qs = \text{okToAddGivenDir} \ q \ qs \ \text{<which function?>}
  \]
  
  \[
  \text{okToAddLevel} \ q \ qs = \text{okToAddGivenDir} \ q \ qs \ \text{id}
  \]
okToAdd – Revisited

• Old version of okToAdd:

\[
\text{okToAdd } q \, q_s = \text{okToAddUp } q \, q_s \&\& \\
\text{okToAddDown } q \, q_s \&\& \\
\text{okToAddLevel } q \, q_s
\]

• New version of okToAdd:

\[
\text{okToAdd } q \, q_s = \text{okToAddGivenDir } q \, q_s \, \text{succ} \&\& \\
\text{okToAddGivenDir } q \, q_s \, \text{pred} \&\& \\
\text{okToAddGivenDir } q \, q_s \, \text{id}
\]
okToAdd – Revisited

• New version of okToAdd:
  
  \[
  \text{okToAdd } q \ q s = (\text{okToAddGivenDir } q \ q s \ \text{succ}) \ \&\& \\
  (\text{okToAddGivenDir } q \ q s \ \text{pred}) \ \&\& \\
  (\text{okToAddGivenDir } q \ q s \ \text{id})
  \]

• Extract functions to list: \([\text{succ}, \text{pred}, \text{id}]\)

• Check if all elements of list satisfies predicate:
  – Pattern: all
  – Signature: all :: (a -> Bool) -> [a] -> Bool
  – Example: all even [2,4,6] => True

• Using a list:
  
  \[
  \text{okToAdd } q \ q s = \text{all} (\text{okToAddGivenDir } q \ q s) [\text{succ}, \text{pred}, \text{id}]
  \]
Current Solution

• Summary so far:

\[
\text{okToAdd} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Bool}
\]

\[
\text{okToAdd} \ q \ qs = \text{all} (\text{okToAddGivenDir} \ q \ qs) \ [\text{succ, pred, id}]
\]

\[
\text{okToAddGivenDir} :: \text{Int} \rightarrow [\text{Int}] \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Bool}
\]

\[
\text{okToAddGivenDir} \ q0 \ qs \ f = \text{and} \ (\text{zipWith} \ (/=)) \ (\text{tail} \ (\text{iterate} \ f \ q0)) \ qs
\]
Next step: Extend a solution

• Take a partial solution and find all possible ways to extend that solution with one queen.

• Type signature:
  
  extendSolution :: [Int] -> [[Int]]

• Steps for extendSolution:
  1. Create all eight possible queen positions for new column.
  2. Take only the queens that are OK to add.
  3. Create extensions based on queens that are OK to add.
Patterns for extendSolution

• Create all eight possible queen positions: [0..7]
• Take only the queens that are OK to add.
  – Pattern: filter
  – Signature: \( \text{filter :: (}a \to \text{Bool}) \to [a] \to [a]\)
  – Example: \( \text{filter even } [0..7] \Rightarrow [0,2,4,6]\)

• Create extensions based on queens that are OK to add.
  – Pattern: map
  – Signature: \( \text{map :: (}a \to b) \to [a] \to [b]\)
  – Example: \( \text{map } (*2) [1,2,3] \Rightarrow [2,4,6]\)

• Solution:

\[
\text{extendSolution :: [Int]} \to [[\text{Int}]]
\]
\[
\text{extendSolution } qs = \text{map } (:qs) \$ \text{filter } (\text{`okToAdd` qs}) [0..7]
\]
Next step: Create All Solutions

• Step 1: Apply “extendSolution” to the empty solution.

  extendSolution [] => [[0],[1],[2],[3],[4],[5],[6],[7]]

• Step 2: Apply “extendSolution” to all partial solutions.

  extendSolution [0] => [[2,0],[3,0],...,[7,0]]
  extendSolution [1] => [[3,1],[4,1],...,[7,1]]
  ...
  extendSolution [7] => [[0,7],[1,7],...,[5,7]]

• Step 3: Join all extension lists into one.

  [[2,0],[3,0],...,[7,0],[3,1],...,[5,7]]

• Repeat step 2 and 3 until done.
Create All Solutions, Implementation

• Recursive implementation: All solutions with n queens
  
  allSolutions :: Int -> [[Int]]
  allSolutions 0 = [[]]

• Recursive case: Create all solutions with n-1 queens, and extend all of them.

• Create all solutions with n-1 queens:
  
  allSolutions (n-1)

• Extend all solutions:
  
  map extendSolution (allSolutions (n-1))

• Merge it all into one list of partial solutions:
  
  concat $ map extendSolution (allSolutions (n-1))

• Or simply
  
  concatMap extendSolution (allSolutions (n-1))
allSolutions

• Full version

\[
\text{allSolutions} :: \text{Int} \rightarrow [[\text{Int}]]
\]
\[
\text{allSolutions} \ 0 = [[]]
\]
\[
\text{allSolutions} \ n = \text{concatMap} \ \text{extendSolution} \ (\text{allSolutions} \ (n-1))
\]

• Hmm... Apply function multiple times... What pattern?

\[
\text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]
\]

• Iterate with extendSolution:

\[
\text{iterate} \ (\text{concatMap} \ \text{extendSolution}) \ [[]]
\]

• Take the n’th entry:

\[
(\text{iterate} \ (\text{concatMap} \ \text{extendSolution}) \ [[]]) \ !! \ n
\]

• Final version:

\[
\text{allSolutions} :: \text{Int} \rightarrow [[\text{Int}]]
\]
\[
\text{allSolutions} \ n = (\text{iterate} \ (\text{concatMap} \ \text{extendSolution}) \ [[]]) \ !! \ n
\]
Wrapping It All Up

• We have a function that returns all solutions for n queens.
• Generate all solutions for 8 queens and take the first.

    solution :: [Int]
    solution = head (allSolutions 8)
Complete Solution

solution :: [Int]
solution = head (allSolutions !! 8)

allSolutions :: [[[Int]]]
allSolutions = iterate (concatMap extendSolution) [[]]

extendSolution :: [Int] -> [[[Int]]
extendSolution qs = map (:qs) $ filter (`okToAdd` qs) [0..7]

okToAdd :: Int -> [Int] -> Bool
okToAdd q qs = all (okToAddGivenDir q qs) [succ, pred, id]

okToAddGivenDir :: Int -> [Int] -> (Int -> Int) -> Bool
okToAddGivenDir q0 qs f = and $ zipWith (/=) (tail $ iterate f q0) qs
Thank You