COSC441 Concurrent Programming

Higher Order Functions and a Case Study

Richard A. O'Keefe

September 12, 2017

Outline

- Higher Order Functions
- The *Chameneos* Case Study

Higher Order Functions

- The phrase comes from logic.
- In first-order logic, the arguments of functions and predicates can be data only.
- In higher-order logic, the arguments of functions and predicates can also be functions and predicates.
- A higher-order function has a function as an argument or a function as a result.

Practical Use of Higher Order Functions 1

- Fortran (1957), Lisp (1958), and Algol (1960) allowed procedures to have procedures as arguments.
- This allowed numerical differentiation, integration, and optimisation of functions and solution of equations, also general sorting and searching algorithms.
- Fortran, Algol 60, PL/I (1965), Pascal (1970), and other languages of that time only allowed existing procedures to be passed. They could not be returned or constructed.
- Lisp and Algol 68 (1968) allowed anonymous functions.

Practical Use .. 2

- It was realised that higher order functions provide *abstraction over control structures*.
- Functional languages have a common stock of "iteration" functions: *all*, *any*, *for-each*, *map*, *foldl*, *foldr*, *filter*.
- They are what you use instead of loops.

Example: foreach/2 (A)

Suppose we have a list $[X_1,\ldots,X_n]$ and want to call $f(X_1)$, $\ldots$, $f(X_n)$, discarding the results. We can write

```
do_f([X|Xs]) -> f(X), do_f(Xs);
do_f([]) -> ok.
```

Example: foreach/2 (B)

That just works for f. We don't want to keep writing the same code over and over. So take f out and make it a parameter.

foreach(F, [X|Xs]) $\rightarrow$ F(X), foreach(F, Xs);
foreach(_, []) $\rightarrow$ ok.

do_f(Xs) $\rightarrow$ foreach(**fun** f/1, Xs).

The (tail-)recursive control structure is now hidden inside a re-usable function.

Example: map/2 (A)

Suppose we have a list $[X_1,\ldots,X_n]$ and want to compute the list $[X_1+2,\ldots,X_n+2]$. We can write

add_2([X|Xs]) $\rightarrow$ [X+2 | add_2(Xs)];
add_2([]) $\rightarrow$ [].

As before, we want to split this into "how to do something to every element and collect the results" and "what to do to each element".

Example: map/2 (B)

Here the function is $x \mapsto x + 2$.

map(F, [X|Xs]) $\rightarrow$ [F(X) | map(F, Xs)];
map(_, []) $\rightarrow$ [].

add_2(Xs) $\rightarrow$ map(**fun** (X) $\rightarrow$ X+2 **end**, Xs).

This is not the only way to compute that result.

Example: map/2 (C)

We can use tail recursion, and then reverse the result.

map(F, Xs) $\rightarrow$ map_loop(F, Xs, []).

map_loop(F, [X|Xs], R) $\rightarrow$ map_loop(F, Xs, [F(X)|R]);
map_loop(F, [], R) $\rightarrow$ lists:reverse(R).

Writing such a loop ourselves, we have to choose the approach. Using map/2, we don't know how it is done and don't care.

Example: foldl/3 and foldr/3 (A)

Suppose we want to add up a list of numbers. We can do it from right to list or left to right.

sum_rtl([X|Xs]) → X + sum_rtl(Xs);
sum_rtl([]) → 0.

sum_ltr(Xs) → sum_ltr_loop(Xs, 0).

sum_ltr_loop([X|Xs], Acc) → sum_ltr_loop(Xs, Acc+X);
sum_ltr_loop([], Acc) → Acc.

Example: foldl/3 and foldr/3 (B)

We abstract out 0 and _+_.

foldr(F, A, [X|Xs]) $\rightarrow$ F(X, foldr(F, A, Xs));
foldr(_, A, []) $\rightarrow$ A.

foldl(F, A, [X|Xs]) $\rightarrow$ foldl(F, F(X, A), Xs);
foldl(_, A, []) $\rightarrow$ A.

Example: foldl/3 and foldr/3 (C)

For adding integers, we don't care which order it is done. We may care that foldl is tail recursive (uses a fixed amount of stack) and foldr is body recursive (stack use is linear in list size).
For adding floats, the two orders give different results.

Example: filter/2 (A)

Suppose we want to select the elements of a list that satisfy some predicate p. We can write

```
pick_p([X|Xs]) →
    case p(X) of
        true → [X | pick_p(Xs)];
        false → pick_p(Xs)
    end;
pick_p([]) → [].
```

This too can be done using tail recursion followed by reverse.

Example: filter/2 (B)

We abstract out p.

```
filter(P, [X|Xs]) →
    case P(X) of
        true → [X | filter(P, Xs)];
        false → filter(P, Xs)
    end;
filter(_, []) → [].

pick_p(Xs) → filter(fun p/1, Xs).
```

- These functions are already in the library.
- When we implement abstract data types such as sets and dictionaries we can (should!) provide similar functions to hide the implementation.
- When we need something similar but different it is not hard to add it.
- "Pure" function languages can use laws like map(F,map(G,Xs)) = map(F.G, Xs) to do loop fusion.

Example: foldl2/3 (A)

Let us generalise dot product.

```
foldl2(F, A, [X|Xs], [Y|Ys]) ->
    foldl2(F, F(A, X, Y), Xs, Ys);
foldl2(_, A, [], []) -> A.

dot(Xs, Ys) ->
    foldl2(fun (A, X, Y) -> A+X*Y end, 0, Xs, Ys).
```

This is *not* in the lists library.

Example: foldl2/3 (B)

Euclidean distance

```
math:sqrt(
    foldl2(fun (A, X, Y) → A+(X-Y)*(X-Y) end,
        Xs, Ys))
```

Count matching elements

```
inc_if_eq(N, X, X) → N+1;
inc_if_eq(N, _, _) → N.
foldl2(fun inc_if_eq/3, 0, Xs, Ys)
```

The Case Study

- I want this to be a dialogue.
- Yes I have a solution to the problem.
- But I want us to work towards it *together*.

Chameneos-redux

- Source: "The Computer Language Benchmarks Game", benchmarksgame.alioth.debug.org
- A chameneos is a sapient creature which can change colours between red, blue, and yellow. They normally live alone in the forest, but occasionally go to the mall to play cards with the first chameneos they meet, then go back to the forest.
- When two chameneos meet, if they are the same colour, they stay that colour, otherwise they both change to the third colour.
- See the web page for details.

Sample sizes

C	448 lines
Ada	435 lines
C++	288 lines
C#	242 lines
Java	236 lines
Go	178 lines
Smalltalk	95 lines
Erlang	88 lines

More precisely

- The Erlang version is 1 page of application code
- and half a page of iteration functions.
- The C version is the fastest.
- I tried to get someone else's Smalltalk version going first. It was 468 lines and I could *not* understand it.
- My Smalltalk version was written by thinking in Erlang, then hacking a bit to reduce dynamic allocation.
- The Erlang version was spectacularly easy to think up by comparison and worked first time.

Questions

- What are the actors (processes)?
- What do they know?
- What do they say to each other?