Advanced Functional Programming, 1DL450 2012

Lecture 2, 2012-11-01

Cons T Åhs

Higher order functions

```
hof() ->
  F = fun(X) -> X * X + 1 end,
  L = lists:map(F, [1, 2, 3],

  G = fun([])     -> nil;
         ([_|_]) -> cons
      end,

  Y = G(L),
  Y == nil.
```

- Syntax for anonymous functions is rather verbose
- Anonymous functions can have several clauses and use pattern matching
- A variable can be bound to a function
- Apply the function by using the variable instead of a function name
  - Erlang got this right!
- What is the value of `hof()`?

Scoping revisited

- The scope of a variable binding is the rest of the function clause
  - An expression can only access variables bound before the expression
  - It is not possible to write a local recursive function in the "ordinary" way

```
no(N) ->
  G = fun(0) -> 1;
         (N) -> N*G(N-1)
      end,

  G(N).
```

- It is possible to write a "local recursive" function using higher order functions
  - Observe that `G` inside is "just" a function variable so it has to be passed to the function
  - This is a good exercise!
  - Write factorial in this way.

Higher order functions

```
make_adder(N) ->
  fun(X) -> X + N end.

inclist(L) ->
  lists:map(make_adder(3), L).

whatlist(L) ->
  lists:map(fun make_adder/1, L).

what(L, V) ->
  lists:map(fun(F) -> F(V) end, L).
```

- A function can be returned
- Notation for passing a named function as an argument
- Describe the functions `inclist/1`, `whatlist/1` and `what/2`

Higher order functions

```
cumbersome(M) ->
  MakeAdder = fun(N) ->
                fun(X) -> X + N end
              end,
  (MakeAdder(3))(M).
```

- Making curried functions suitable for partial application is possible, but quickly becomes a bit difficult to read.
- This is much easier in languages designed for this from the start.

Digression on closures

```
make_adder(N) ->
  fun(X) -> X + N end.

make_what(M) ->
  fun() -> fibonacci(M) end.

do_it(D) ->
  D().
```

- We have the cool feature of being able to return a closure, i.e., a function and the environment it was defined in.
- What does `make_what/1` do?
  - Returns a function of no (?) argument.
  - It delays a computation!
  - The body is evaluated only when we apply the result (of make_what/1) to ().
- We can thus save and represent a computation and do it later.

Variables can hold anything

```
-module(sequences).
-export([plus/2, minus/2]).

plus(X, Y) -> X ++ Y.
minus(X, Y) -> X -- Y.
```

```
-module(numbers).
-export([plus/2, minus/2]).

plus(X, Y) -> X + Y.
minus(X, Y) -> X - Y.
```

```
-module(eval).
-export([eval/4]).

eval(M, F, A1, A2) ->
    M:F(A1, A2).
```

```
10> eval:eval(sequences, plus, [1,2,3], [a,b,c]).
[1,2,3,a,b,c]
11> eval:eval(numbers, plus, 4, 7).
11
12>
```

Variables can hold anything

- A variable can be bound to
  - ordinary values and functions (no surprise)
  - function *names*
  - *modules*
- This means you can send a whole module M as an argument to another function and the receiving function then calls known functions in M.
  - Is this useful?
  - Yes!
- It also means that given a module you can vary the actual function that is called by passing the *name* in a variable.
  - Is this useful?
  - Possibly.
- Both variations lead to the possibility to map, e.g., user input directly to Erlang modules and functions at runtime.
  - Great way to make a really insecure system!

Variables can hold anything

- We had two modules which exported the same function names and arities
  - They thus have the same interface!
  - This concept exists in Erlang, but has the name *behaviour*
  - It can be used in the same way as in, e.g., Java by providing several different implementations of the same (abstract) interface
  - A very commonly used behaviour is the gen_server (for generic server)
  - You provide the details and a generic server takes care of the generic parts.

BIFs (Built In Functions)

- BIFs exist to provide functionality that can’t be done in pure Erlang
  - interface with the real world for things like date, time and low level file system access
  - conversion between primitive types such as
    - atom_to_list (convert an atom to a “string”)
    - list_to_atom (convert a “string” to a (new) atom)
    - etc
- There might also be BIFs for functions that can be implemented in Erlang, but a BIF will do it faster.
- Read documentation!

Standard Libraries

- Erlang comes with a large set of standard libraries, e.g,
  - list function
  - dictionaries of varying representation
  - ets, dets - term storage, either in memory or on disk
  - mnesia - database built on top of dets
  - etc
- Read the documentation

List comprehensions

- Erlang has the standard higher order list functions such map, filter and foldl/r
- Erlang also has *list comprehension* for concise construction of lists
- Very similar to describing sets
- Examples

```
foo(L) ->
  Squares = [X*X || X <-L],
  Squares = lists:map(fun(X) -> X*X end, L),

  Appls = [{X, f(X)} || X <- L, X > 2],
  Appls = lists:map(fun(X) -> {X, f(X)} end,
                    lists:filter(fun(X) -> X > 2 end, L)),
  Appls = lists:foldr(fun(X, S) ->
                        case X > 2 of
                          true  -> [{X, f(X)} | S];
                          false -> S
                        end
                      end,
                      [], L),
  {Squares, Appls}.
```

List comprehensions

- The left hand is an expression for constructing an element (evaluated
- The right hand side consists of
  - generators (`Var <- Expression`)
  - conditions or filters (a boolean expression on a `Var`)
- There can be several generators and conditions

```
map(F, L) -> [F(X) || X <- L].

filter(P, L) -> [X || X <- L, P(X)].

combine(L) -> [{X, Y} || X <- L, Y <- L, X=/=Y].
```

List comprehension

- Generate all permutations of a list
- The result of one generator can be used in another
- Very compact, but it takes some time to understand
- Exercise: write the same function without comprehension

```
perms([]) -> [[]];
perms(L)  ->
  [[X|T] || X <- L, T <- perms(L -- [X])].
```

Concurrent Programming

- Process model used in Erlang
  - No shared memory between processes
    - Problems when you have a *shared* and *mutable* state - Erlang has neither
    - A process that dies does not corrupt the state of another process
  - Communication by message passing; messages are *copied* (even within the same VM)
  - Fast and easy process creation
    - Initial size of a process is 3-400 bytes
  - Easy distribution among
    - cores (within same VM)
    - VMs (on same hardware node)
    - hardware nodes
  - Communication is identical regardless of where the other process lives
  - Processes are identified by PIDs (process identifiers)

What about state?

- Real world computations need state
- State is encoded in a process that reacts to messages
  - init state
  - wait for message
  - compute new state and “loop”

```
start() -> server(init_state()).

server(State) ->
  server(process_message(get_msg(), State)).
```

  - start the server and send messages to it

Managing Processes

- Three basic primitives are used to handle processes
- Create process - returns pid (process id)

```
spawn(Function) or spawn(M, F, Args)
```

- Send a message - returns Msg

```
Pid ! Msg
```

- Receive a message from the message queue (the process will wait if there is no message) - returns value of chosen expression

```
receive
  Pattern1 -> Expr1;
  Pattern2 -> Expr2;
  ...
end
```

Selective receive

- Note that a receive will wait until it finds a message matching the pattern
  - Messages might not be processed in the order they come
  - This can be expensive since the message queue has to be searched

```
receive
  foo -> f(..)
end,
receive
  bar -> g(..)
end
```

Example

```
start() -> server(0).

server(Count) ->
  NewCount = receive
               {report, Pid} ->
                 Pid ! Count,
                 Count;
               _Msg -> Count + 1
        end,
  server(NewCount).

32> P = spawn(fun simple:start/0).
<0.110.0>
33> P!foo.
foo
34> P!foo.
foo
35> P!foo.
foo
36> P!{report, self()}.
{report,<0.88.0>}
37> receive M -> M end.
3
```

Distribution made easy

- Distribute work load among a number of workers
- Input
  - the work to be done, a queue of tasks
  - the workers that performs the work (pids)
- What is specific for each problem?
  - How to get a chunk of work from the queue
  - How to combine results from a single worker with the result from the others

Distribution made easy

- We're done when the queue is empty **and** we have no active workers.
- We wait for a worker to return a result when the queue is empty **or** we have no passive workers
- We activate a worker when the queue is non empty and we have passive workers.
- Initial state is a queue of work, no active workers and a collection of passive workers.

Distribution made easy

```
sequential(L) -> lists:filter(fun is_prime/1, L).

process_work([], [], _, State) -> State;
process_work(Work, Active, Passive, State)
  when Work =:= []; Passive =:= [] ->
  receive {Worker, M} ->
      process_work(Work, lists:delete(Worker, Active),
                   [Worker | Passive], add_result(State, M))
  end;
process_work(Work, Active, [Worker | Passive], State) ->
  {Chunk, Rest} = get_chunk(State, Work),
  Worker ! {self(), Chunk},
  process_work(Rest, [Worker | Active], Passive, State).

worker() ->
  receive {Pid, Work} ->
      Pid ! {self(), sequential(Work)},
      worker()
  end.
```

Simple Message Passing

- Note that you have to set up the actual protocol yourself
- If you want a reply, a sent message should include a return address
- This goes for the reply as well - the original sender might want to know who sent the reply
- This might also apply to request identifiers so a more general request would contain both a return address and an identifier

More on process handling

- Linking processes for error handling and supervision
- Timeouts