Exam in Advanced Programming, 2012

Sebastian Paaske Tørholm

October 30, 2012

1 IVars in Erlang

The IVars are modelled using Erlang processes. Each IVar starts in an empty state, changing to a set state once a value has been specified.

Immutability is ensured by throwing away `put`-requests when in the set state, marking the ivar as compromised if required by the specification.

For the princess variables, robustness with regards to malformed predicates in handled by evaluating the predicate inside a `try` clause; ignoring the `put` request if anything but `true` is returned, or an exception of any kind is thrown.

The functions in `pmuse` have been solved using a simple map-reduce implementation. Each mapper is spawned with a mapping function, which is given pairs of data and an IVar to store the result in. This makes it possible to use the same mapper for both `pmmap` and `treeforall`. Distribution of problems is done in the same way that the map reduce framework from assignment 3 does it. Reduction is done in the original thread, and is thus not parallellized in any way.

Order is maintained in `pmmap` by keeping a list of `IVars` in the correct order. These can then be queried for the computed values upon reducing. Values are only `put` into each IVar once in one mapping task, so no IVar will be compromised.

Using princess predicates for `treeforall` is slightly awkward. If we perform a `get` on an `IVar` for which the predicate rejected the entry, we block the thread. This problem is resolved by using a custom predicate, which uses the supplied predicate on the data contained in values of the form `{ok, D}`, and always accepting values of the form `notok`. We then put the value we wish to check, wrapped in a `{ok,D}` first, followed by putting a `notok`. This guarantees us that a value will eventually be accepted. Since the order of messages is maintained when sending from one process to another, we know that the `ok`-message will be processed first. This presents no problem with regards to compromising the IVar, as princess IVars cannot be compromised.

An exception is used to break out of the reducer as soon as a falsy value is found. As such, we terminate our reduction at the earliest possible point.

1.1 Assessment

The code tries to be indented well, using meaningful variable- and function names. Pattern matching is used whenever it makes sense. The supplied unit testing provides a good indication that the code in all likelihood satisfies the specification.

Using `pmmap` or `treeforall` on large lists (of 100,000 elements, for instance) may be problematic, since a process is spawned for each IVar, and an IVar is created for each element in the list.

A Code

A.1 IVars in Erlang

A.1.1 pm.erl

```
%% IVars for Erlang
%% Advanced Programming exam 2012, DIKU
%% Sebastian Paaske Tørholm <sebbe@diku.dk>

-module(pm).
-export([newVanilla/0, newPrincess/1, get/1, put/2, compromised/1]).

%% Interface

newVanilla() -> spawn(fun vanilla_loop/0).

newPrincess(Pred) -> spawn(fun() -> princess_loop(Pred) end).

get(IVar) -> rpc(IVar, get).

put(IVar, Term) -> put_async(IVar, Term).

compromised(IVar) -> rpc(IVar, compromised).

%% Asynchronous communication
info(Pid, Msg) ->
    Pid ! Msg.

put_async(Pid, Term) ->
    info(Pid, {put, Term}).

%% Synchronous communication
rpc(Pid, Request) ->
    Pid ! {self(), Request},
```

```
    receive
    {Pid, Response} ->
        Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.

%% Vanilla IVars
vanilla_loop() ->
    receive
        {put, Term} ->
            vanilla_loop_set(Term, false);
        {From, compromised} ->
            reply(From, false),
            vanilla_loop()
    end.

vanilla_loop_set(Val, Comp) ->
    receive
        {put, _} ->
            vanilla_loop_set(Val, true);
        {From, get} ->
            reply(From, Val),
            vanilla_loop_set(Val, Comp);
        {From, compromised} ->
            reply(From, Comp),
            vanilla_loop_set(Val, Comp)
    end.

%% Princess IVars
princess_loop(Pred) ->
    receive
        {put, Term} ->
            try Pred(Term) of
                true -> princess_loop_set(Pred, Term);
                _    -> princess_loop(Pred)
            catch
                _:_  -> princess_loop(Pred)
            end;
        {From, compromised} ->
            reply(From, false),
            princess_loop(Pred)
    end.
```

```
princess_loop_set(Pred, Val) ->
    receive
        {put, _} ->
            princess_loop_set(Pred, Val);
        {From, get} ->
            reply(From, Val),
            princess_loop_set(Pred, Val);
        {From, compromised} ->
            reply(From, false),
            princess_loop_set(Pred, Val)
    end.
```

A.1.2 pmuse.erl

```
%% Utility functions working with IVars
%% Advanced Programming exam 2012, DIKU
%% Sebastian Paaske Tørholm <sebbe@diku.dk>

-module(pmuse).
-export([pmmap/2, treeforall/2]).

%% Interface

pmmap(Fun, List) ->
    Mappers = init_mappers(20, fun (D, IV) -> pm:put(IV, Fun(D)) end),
    Data    = lists:map(fun(D) -> {D, pm:newVanilla()} end, List),
    send_data(Mappers, Data),
    Results = lists:map(fun({_,IV}) -> pm:get(IV) end, Data),
    lists:foreach(fun stop_async/1, Mappers),

    Results.

treeforall(Tree, Pred) ->
    Mappers = init_mappers(20, fun(D, IV) ->
        pm:put(IV, {ok, D}),
        pm:put(IV, notok)
    end),
    List    = preorder_treewalk(Tree),
    OurPred = fun({ok, D}) -> Pred(D);
                 (notok)   -> true
              end,
    Data    = lists:map(fun(D) -> {D, pm:newPrincess(OurPred)} end, List),
    send_data(Mappers, Data),
```

```
    Result = try treeforall_reduce(Data) of
                 _ -> true
             catch
                 throw:false -> false
             end,
    lists:foreach(fun stop_async/1, Mappers),

    Result.

%% asynchronous communication

info(Pid, Msg) ->
    Pid ! Msg.

data_async(Pid, D) ->
    info(Pid, {data, D}).

stop_async(Pid) ->
    info(Pid, stop).

%% Implementation

init_mappers(0, _) -> [];
init_mappers(N, Fun) ->
    Mapper = spawn(fun() -> mapper_loop(Fun) end),
    [Mapper | init_mappers(N-1, Fun)].

send_data(Mappers, Data) ->
    send_loop(Mappers, Mappers, Data).

send_loop(Mappers, [Mid|Queue], [D|Data]) ->
    data_async(Mid, D),
    send_loop(Mappers, Queue, Data);
send_loop(_, _, []) -> ok;
send_loop(Mappers, [], Data) ->
    send_loop(Mappers, Mappers, Data).

preorder_treewalk(leaf) -> [];
preorder_treewalk({node, E, L, R}) ->
    [E | preorder_treewalk(L) ++ preorder_treewalk(R)].

mapper_loop(Fun) ->
    receive
        {data, {Data, IVar}} ->
```

```
            Fun(Data, IVar),
            mapper_loop(Fun);
        stop ->
            ok
    end.

treeforall_reduce(Data) ->
    lists:foreach(fun({_,IV}) ->
        case pm:get(IV) of
            {ok, _} -> true;
            notok   -> throw(false)
        end
    end, Data).
```

A.1.3 pmtest.erl

```
%% Unit tests for pm module
%% Advanced Programming exam 2012, DIKU
%% Sebastian Paaske Tørholm <sebbe@diku.dk>

-module(pmtest).
-include_lib("eunit/include/eunit.hrl").

%% Vanilla IVars {{{
% Putting and getting gives the same element back
vanilla_put_get_test() ->
    IV = pm:newVanilla(),
    pm:put(IV, 5),
    ?assert(5 =:= pm:get(IV)).

get_block_getter() ->
    receive
        {From, {get, IV}} ->
            V = pm:get(IV),
            From ! {got, V}
    end.

% Getting blocks until data is put into IVar
vanilla_get_block_test() ->
    IV = pm:newVanilla(),
    P = spawn(fun() -> get_block_getter() end),
    P ! {self(), {get, IV}},

    timer:sleep(453),
```

```
    pm:put(IV, data_packet),
    receive
        {got, data_packet} -> ?assert(true);
        _                  -> ?assert(false)
    end.

% compromised acts according to spec
vanilla_compromised_test() ->
    IV = pm:newVanilla(),
    ?assert(not(pm:compromised(IV))),
    pm:put(IV, some_data),
    ?assert(not(pm:compromised(IV))),
    pm:put(IV, some_more_data),
    ?assert(pm:compromised(IV)).

% cannot overwrite value
vanilla_immutable_test() ->
    IV = pm:newVanilla(),
    pm:put(IV, something),
    pm:put(IV, something_else),
    ?assert(something =:= pm:get(IV)).
%% }}}
%% Princess IVars {{{

accept(_) -> true.

% Putting and getting gives the same element back
princess_put_get_test() ->
    IV = pm:newPrincess(fun accept/1),
    pm:put(IV, 5),
    ?assert(5 =:= pm:get(IV)).

% Getting blocks until data is put into IVar
princess_get_block_test() ->
    IV = pm:newPrincess(fun accept/1),
    P = spawn(fun() -> get_block_getter() end),
    P ! {self(), {get, IV}},

    timer:sleep(453),

    pm:put(IV, data_packet),
    receive
        {got, data_packet} -> ?assert(true);
```

```
        _                   -> ?assert(false)
    end.

% compromised acts according to spec
princess_compromised_test() ->
    IV = pm:newPrincess(fun accept/1),
    ?assert(not(pm:compromised(IV))),
    pm:put(IV, some_data),
    ?assert(not(pm:compromised(IV))),
    pm:put(IV, some_more_data),
    ?assert(not(pm:compromised(IV))).

% cannot overwrite value
princess_immutable_test() ->
    IV = pm:newPrincess(fun accept/1),
    pm:put(IV, something),
    pm:put(IV, something_else),
    ?assert(something =:= pm:get(IV)).

% Getting doesn't get anything until predicate is satisfied
princess_get_block_pred_test() ->
    IV = pm:newPrincess(fun(V) -> V > 5 end),
    P = spawn(fun() -> get_block_getter() end),
    P ! {self(), {get, IV}},

    pm:put(IV, 4),
    timer:sleep(123),
    pm:put(IV, 2),
    timer:sleep(234),
    pm:put(IV, 23),
    timer:sleep(113),
    pm:put(IV, 43),

    receive
        {got, 23} -> ?assert(true);
        _         -> ?assert(false)
    end,
    ?assert(23 =:= pm:get(IV)).

% Princess predicates handles non-boolean return values and exceptions correctly
bad_predicate(1) -> 5;
bad_predicate(2) -> throw(true);
bad_predicate(3) -> erlang:error(wrong_stuff);
bad_predicate(4) -> exit(stuff);
```

```
bad_predicate(_) -> true.

princess_bad_pred_test() ->
    IV = pm:newPrincess(fun bad_predicate/1),
    P = spawn(fun() -> get_block_getter() end),
    P ! {self(), {get, IV}},

    pm:put(IV, 1),
    pm:put(IV, 2),
    pm:put(IV, 3),
    pm:put(IV, 4),
    pm:put(IV, 5),

    receive
        {got, 5} -> ?assert(true);
        _        -> ?assert(false)
    end.

%% }}}
```

A.1.4 pmusetest.erl

```
%% Unit tests for pmuse module
%% Advanced Programming exam 2012, DIKU
%% Sebastian Paaske Tørholm <sebbe@diku.dk>

-module(pmusetest).
-include_lib("eunit/include/eunit.hrl").

%% pmmap {{{
% Compare pmmap to lists:map
pmmap_0_test() ->
    F = fun(E) -> E*E end,
    L = lists:seq(1, 1000),
    ?assert( lists:map(F, L) =:= pmuse:pmmap(F, L) ).

pmmap_1_test() ->
    F = fun(E) -> E+5 end,
    L = [],
    ?assert( lists:map(F, L) =:= pmuse:pmmap(F, L) ).

pmmap_2_test() ->
    F = fun(E) -> {ok, E} end,
    L = [65, 3, 7, 3, 87, 32, 5, 2, 6],
```

```
    ?assert( lists:map(F, L) =:= pmuse:pmmap(F, L) ).
%% }}}
%% treeforall {{{
% true on empty tree
reject(_) -> false.

tfa_accept_empty_test() ->
    ?assert(pmuse:treeforall(leaf, fun reject/1)).

% treeforall doesn't evaluate
% this test uses that the tree is evaluated in preorder.
retimm_pred(3) -> retimm_pred(3);
retimm_pred(1) -> false;
retimm_pred(_) -> true.

tfa_returns_immediately_test() ->
    Tree = {node, 2, {node, 1, leaf, leaf},
                     {node, 3, leaf, leaf}},
    ?assert(not(pmuse:treeforall(Tree, fun retimm_pred/1))).

% treeforall on some regular trees
tfa_0_test() ->
    Tree = {node, 5, {node, 3, {node, 2, {node, 1, leaf, leaf}, leaf},
                               {node, 4, leaf, leaf}},
                     {node, 8, leaf, leaf}},
    Pred = fun(V) -> V < 10 end,
    ?assert(pmuse:treeforall(Tree, Pred)).

tfa_1_test() ->
    Tree = {node, 5, {node, 3, {node, 2, {node, 1, leaf, leaf}, leaf},
                               {node, 4, leaf, leaf}},
                     {node, 8, leaf, leaf}},
    Pred = fun(V) -> V < 8 end,
    ?assert(not(pmuse:treeforall(Tree, Pred))).
%% }}}
```