MAC 0434/5765 — Programação Funcional Contemporânea

Segundo Semestre de 2010

05 de outubro de 2010

Nome do aluno: _____

Assinatura: _____

Nº USP: _____

Instruções:

1. Preencha o cabeçalho acima.
2. Não destaque as folhas deste caderno.
3. A prova tem 6 questões. Antes de começar a trabalhar verifique se o seu caderno de questões está completo.
4. A prova deve ser resolvida individualmente. Não é permitida a consulta a livros, apontamentos ou colegas.
5. Não é permitido o uso de folhas avulsas para rascunho.

Boa prova!

Questão	Valor	Nota
1	2,0	
2	1,0	
3	2,0	
4	1,0	
5	2,0	
6	2,0	
Total	10,0	

Questão 1 (2,0 pontos)

(a) Escreva uma função `map(F, L)` que faça o mesmo que a função `lists:map` da biblioteca de Erlang. Em outras palavras, sua função deve devolver a lista obtida aplicando-se a função `F` a cada elemento da lista `L`.

(b) Reescreva a sua função `map` de modo que a área de memória empregada pela pilha de execução tenha tamanho constante (em vez ser proporcional ao comprimento da lista `L`). Sugestão: Crie uma função auxiliar, que use um acumulador, e faça a nova função `map` chamar a função auxiliar.

Questão 2 (1,0 pontos)

Escreva uma função `foldl(Fun, Acc0, List)` que faça o mesmo que a função `lists:foldl` da biblioteca de Erlang. O primeiro parâmetro de `foldl` é uma função combinadora

```
Fun = fun(ListElem, AccIn) -> AccOut
```

que é usada para acumular o valor dos elementos da lista. O parâmetro `Acc0` é o valor inicial do acumulador. A função `foldl` deve percorrer os elementos da lista da esquerda para a direita (do início para o final — o primeiro elemento da lista deve ser combinado com `Acc0`), incluindo cada elemento no valor acumulado. Ela deve devolver o valor final acumulado.

Questão 3 (2,0 pontos)

(a) Escreva uma função `merge(L1, L2)` que faça a intercalação de duas listas ordenadas. Essa função recebe duas listas em ordem crescente e devolve uma terceira lista, também em ordem crescente, com os mesmos elementos que as duas listas recebidas. (Considere que as listas estão em ordem crescente, mas podem não estar em ordem estritamente crescente. Em outras palavras, elas podem ter elementos repetidos.)

(b) Reescreva a sua função `merge` de modo que a recursão seja de cauda (*tail recursion*). Se necessário, crie uma função auxiliar.

Questão 4 (1,0 pontos)

É dado um servidor `kvs` (que você não precisa escrever) cuja interface é a seguinte:

`kvs:start() -> true`
: Inicia o servidor.

`kvs:store(Key, Value) -> true`
: Associa a chave `Key` ao valor `Value`.

`kvs:lookup(Key) -> {ok, Value} | undefined`
: Procura o valor associado à chave `Key`.

Considere o seguinte exemplo de uso da `lib_chan`:

```
%% Configuration file:
{port, 1234}.
{service, nameServer, password, "ABXy45",
          mfa, mod_name_server, start_me_up, notUsed}.


%% Server code:
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) ->
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! {send, kvs:lookup(K)},
            loop(MM);
        {chan_closed, MM} ->
            true
    end.

%% Client connect example:
%% {ok, Pid} = lib_chan:connect(ServerHost, 1234, nameServer,
%%                              "ABXy45", "").
```

Modifique esse exemplo de modo que, em vez do servico `nameServer`, os clientes vejam dois serviços: um serviço `readOnlyNameServer` e um serviço `updatableNameServer`. Embora esses dois serviços sejam "cascas" em torno do mesmo servidor `kvs`, eles devem ter *passwords* diferentes. O primeiro permite que clientes remotos tenham acesso apenas à função `kvs:lookup`; o segundo permite que clientes remotos chamem também a função `kvs:store`.

Questão 5 (2,0 pontos)

Considere o seguinte arcabouço de servidor genérico:

```
-module(server).
-export([start/1, call/2]).

% Client-server messaging framework.
%
% The callback module must implement the following callbacks:
% init() -> InitialState
% handle_call(Params, State) -> {Reply, NewState}

% Return the pid of a new server with the given callback module.
start(Module) ->
    spawn(fun() -> loop(Module, Module:init()) end).

loop(Module, State) ->
    receive
        {call {Client, Id}, Params} ->
            {Reply, NewState} = Module:handle_call(Params, State),
            Client ! {Id, Reply},
            loop(Module, NewState);
    end.

% Client-side function to call the server and return its reply.
call(Server, Params) ->
    Id = make_ref(),
    Server ! {call, {self(), Id}, Params},
    receive
        {Id, Reply} -> Reply
    end.
```

Escreva um *callback module* para um servidor bancário simples, com as seguintes operações:

`start() -> ok`
: Inicia o servidor bancário.

`new_account() -> Account`
: Cria uma nova conta (com saldo zero) e devolve o número dessa conta.

`get_balance(Account) -> {ok, Balance} | {error, Why}`
: Devolve o saldo da conta `Account`.

`deposit(Account, Amount) -> {ok, NewBalance} | {error, Why}`
: Deposita a quantia `Amount` na conta `Account`.

`withdraw(Account, Amount) -> {ok, NewBalance} | {error, Why}`
: Saca a quantia `Amount` da conta `Account`.

O servidor deve gerar sequencialmente os números das contas: a primeira conta criada deve ter o número 1, a segunda o número 2, etc. As funções `get_balance`, `deposit` e `withdraw` devem devolver erro em caso de número de conta inválido, quantia inválida (ela deve ser positiva) e saldo insuficiente para uma operação de saque.

Questão 6 (2,0 pontos)

Escreva uma função paralela de ordem superior

```
simple_mapreduce(FunMap, FunReduce, Acc0, L) -> Acc
```

que combina as operações *map* e *fold* (também conhecida como *reduce*) da seguinte maneira:

1. Para cada elemento `X` da lista `L`, a chamada `simple_mapreduce(FunMap, FunReduce, Acc0, L)` cria um processo filho que avalia `FunMap(X)` e envia o resultado ao processo que chamou `simple_mapreduce/4`.
2. O processo pai (o que está executando a chamada a `simple_mapreduce/4`) recebe os resultados dos processos filhos. Ao receber cada resultado, ele usa a função

   ```
   FunReduce(MappedListElem, AccIn) -> AccOut
   ```

   para combinar o resultado `MappedListElem` com o valor acumulado dos resultados recebidos anteriormente. O parâmetro `Acc0` é usado como `AccIn` na primeira chamada a `FunReduce/2`.
3. O último valor acumulado (o que inclui o último resultado) é o valor da chamada a `simple_mapreduce/4`.

Note que os elementos da lista mapeada são acumulados numa ordem arbitrária, correspondente à ordem da geração desses elementos pelos processos filhos (o elemento que chegar primeiro é acumulado primeiro).

Você pode supor que não ocorrerão erros ou exceções nas chamadas a `FunMap`. Com essa hipótese, o que diferencia a chamada

```
simple_mapreduce(FunMap, FunReduce, Acc0, L)   % percorre a lista mapeada numa ordem aleatória
```

das chamadas

```
foldl(FunReduce, Acc0, map(FunMap, L))         % percorre a lista mapeada da esquerda para a direita
```

e

```
foldr(FunReduce, Acc0, map(FunMap, L))         % percorre a lista mapeada da direita para a esquerda
```

é a ordem em que os elementos da lista mapeada são acumulados (além da execução paralela, é claro).

Rascunho (não destacar esta página do caderno de questões)