Exam in Advanced Programming, 2012

Sebastian Paaske Tørholm

November 2, 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.

2 Parsing Soil

The parsing is done using `ReadP`.

The structure of the parser construction mostly follows the structure suggested by the grammar. Instead of having two rules each for parameter lists and argument lists, these rules have been simplified using `sepBy`.

Prim had a problem with left recursion in the rule for `concat`. This problem was resolved by special-casing `concat` using `chainl1`, and moving the remaining rules into the nonterminal *PrimBasic*.

In order to allow for arbitrary whitespace, `skipSpaces` is used whereever whitespace is allowed. In general, the parsers only handle whitespace in front of them, leaving the `program` parser to deal with trailing whitespace. To simplify this process, helper parsers `schar` and `sstring` have been introduced, as as `char` and `string`, that ignores arbitrary preceding whitespace.

I have chosen to use a datatype to represent parse errors. The datatype is capable of representing the two types of error that can occur from parsing:

- There are multiple ways to interpret a given source text.
- The source text doesn't represent a valid program.

I'm not quite sure if the first of these can occur. It may be possible due to arbitrary[1] whitespace being allowed between tokens.

2.1 Assessment

The parser has been segmented into sub-parsers, each one of a managable size, making it easier to inspect them for correctness. Combinators of varying types are used from `ReadP`, rather than reinventing them. This yields shorter, more readable code. The code attempts to embrace Haskell's style, and has been run through HLint and ghc-mod check without any notices.

The code comes with unit tests. These unit tests verify the given example programs against their expected parse tree. The also try to verify correct

[1]And thus, also 0.

response to failure, and proper associativity of `concat`. The tests would suggest that the parser works correctly.

Further correctness checking could be done using quickcheck, by generating an AST, serializing it to source code, re-parsing it and comparing the result to the original AST.

3 Interpreting Soil

The name and function environments are modelled using simple lists of tuples. This means we get a $O(n)$ lookup time, which could be improved by changing the datatype if necessary.

The process type has been expanded to contain a process identifier, making the process queue a simple list. The laziness of the lists provides for asymptotically constant concatenation, making them perform well for the necessary operations.

The program is evaluated in the `ProgramEvaluation` monad, which maintains a program state consisting of the function environment and process queue. The function environment is immutable, which is reflected in the type of `runPE`.

Several helpful monadic values have been made, making it possible to write relatively clean monadic code.

I have chosen to handle static errors using the `error` function. This makes it easy to verify that no such error occurs in testing, and is a simple, lightweight and local solution to handling those error conditions cleanly. In addition to the static errors provided in the problem text, a few extra ones have been added. `primToName` throws an error if a non-name primitive occur in the syntax tree, where such primitives shouldn't be allowed. In addition, if there at any point is no process in the process queue when doing a round robin, [2] an error is also thrown. All other errors are handled by sending a suitable message to the `#errorlog` process, and ignoring whatever operation fails. This gives the required

The processes `#println` and `#errorlog` are special-cased, but still exist in the process table. All values except for their mailbox and process id are dummy values.

Evaluating all possible outcomes is done without using a monad transformer, though it seems like it'd likely candidate to be rewritten to use one. In the same vein, the `ProgramEvaluation` monad could have been written using the state monad, as that essentially is what it is.

[2] Which should be an impossibility, as there always should exist at least two processes, `#println` and `#errorlog`.

3.1 Assessment

The code style could most likely be improved; a lot of places checking has to be done to ensure that the input is correct. This leads to much unpacking of option types, verification of length of lists and so on. The monadic style does help simplify the transfer and maintenance of state, making it it simpler to work with.

The code runs through `ghc-mod check` and `hlint` without any notices or warnings.

Basic correctness checks are done by running the interpreter on the sample programs provided, as well as doing manual evaluation. Further correctness checking is a good idea, as the interpreter is rather complex, and subtle bugs easily could hide in the corners. To improve correctness checking, further programs could be written. In addition, quickcheck could be used to compare output to a reference implementation to further catch any bugs.

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.
```

```erlang
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))).
%% }}}
```

A.2 Parsing Soil

A.2.1 SoilParser.hs

```haskell
{-# OPTIONS_GHC -fno-warn-orphans #-}
{-# LANGUAGE TupleSections #-}
--
-- Soil parser
-- Exam for Advanced Programming, B1-2012
-- Sebastian Paaske Tørholm <sebbe@diku.dk>
--

module SoilParser
  ( parseString
  , parseFile
  , Error(..)
  ) where

import SoilAst
import Control.Monad (ap, MonadPlus(mzero, mplus), liftM)
import Control.Applicative ((<$>), Applicative(..),
                            Alternative(empty, (<|>)))
import Text.ParserCombinators.ReadP
import Data.Char (isAlphaNum, isAlpha)

-- Instances for Applicative ReadP and Alternative ReadP aren't in base-4.5
instance Applicative ReadP where
    pure = return
    (<*>) = ap

instance Alternative ReadP where
    empty = mzero
    (<|>) = mplus

{- AmbiguousParseTree - there are multiple possible interpretations of the
                        code.
   InvalidProgram     - there's a syntax error. -}

data Error = AmbiguousParseTree
           | InvalidProgram
    deriving (Show, Eq)

parenthesized :: ReadP a -> ReadP a
parenthesized = between (schar '(') (schar ')')
```

```haskell
-- Char preceded by whitespace
schar :: Char -> ReadP Char
schar c = skipSpaces >> char c

-- String preceded by whitespace
sstring :: String -> ReadP String
sstring s = skipSpaces >> string s

keyword :: String -> Bool
keyword = flip elem ["let", "from", "end", "case", "of", "if", "then", "else",
                     "send", "create", "become", "self"]

program :: ReadP Program
program = do d <- defops
             skipSpaces
             eof
             return d

defops :: ReadP Program
defops = (do f <- defundef
             (fs, as) <- defops
             return (f:fs, as))
     <|> do as <- actops
            return ([], as)

fundef :: ReadP Func
fundef = do sstring "let"
            i    <- ident
            ps   <- parenthesized pars
            sstring "from"
            n    <- name
            schar '='
            ex   <- expr
            sstring "end"
            return $ Func i ps n ex

pars :: ReadP [Name]
pars = sepBy name (schar ',')

expr :: ReadP Expr
expr = fmap Acts actops
   <|> (do sstring "case"
           p <- prim
```

```haskell
           sstring "of"
           (cs, def) <- cases
           sstring "end"
           return $ CaseOf p cs def)
   <|> do sstring "if"
          p1 <- prim
          sstring "=="
          p2 <- prim
          sstring "then"
          ethen <- expr
          sstring "else"
          eelse <- expr
          sstring "end"
          return $ IfEq p1 p2 ethen eelse

cases :: ReadP ([([Name], Expr)], Expr)
cases = (do ps <- parenthesized pars
            schar ':'
            e  <- expr
            (cs, def) <- cases
            return ((ps, e) : cs, def))
    <|> ([],) <$> (schar '_' >> schar ':' >> expr)

actops :: ReadP [ActOp]
actops = many actop

actop :: ReadP ActOp
actop = (do sstring "send"
            as <- parenthesized args
            sstring "to"
            p  <- prim
            return $ SendTo as p)
    <|> (do sstring "create"
            p  <- prim
            sstring "with"
            (fcp, fca) <- fcall
            return $ Create p fcp fca)
    <|> do sstring "become"
           (fcp, fca) <- fcall
           return $ Become fcp fca

fcall :: ReadP (Prim, [Prim])
fcall = do p <- prim
           a <- parenthesized args
```

```
           return (p, a)

args :: ReadP [Prim]
args = skipSpaces >> sepBy prim (schar ',')

prim :: ReadP Prim
prim = chainl1 primBasic (sstring "concat" >> return Concat)

primBasic :: ReadP Prim
primBasic = fmap Id ident
        <|> fmap Par name
        <|> (sstring "self" >> return Self)

ident :: ReadP Ident
ident = schar '#' >> munch1((||) <$> isAlphaNum <*> (==) '_')

name :: ReadP Name
name = do skipSpaces
          n  <- satisfy isAlpha
          ns <- munch((||) <$> isAlphaNum <*> (==) '_')
          case n:ns of
            name' | keyword name' -> pfail
                  | otherwise     -> return name'

parseString :: String -> Either Error Program
parseString s = case readP_to_S program s of
                  []        -> Left InvalidProgram
                  [(p,"")] -> Right p
                  _         -> Left AmbiguousParseTree

parseFile :: FilePath -> IO (Either Error Program)
parseFile = liftM parseString . readFile
```

A.2.2 SoilParserTest.hs

```
import SoilParser
import SoilAst
import Test.HUnit
import System.FilePath (replaceExtension, replaceBaseName)

-- ASTs {{{
helloWorldAST :: Program
helloWorldAST =
    (
```

```
        [Func {
            funcname = "print",
            params = [],
            receive = "message",
            body = Acts [SendTo [Par "message"] (Id "println")]}
        ],
        [Create (Id "hw1") (Id "print") []
        ,Create (Id "hw2") (Id "print") []
        ,SendTo [Id "Hello"] (Id "hw1")
        ,SendTo [Id "World"] (Id "hw2")
        ]
    )

cleanUpAST :: Program
cleanUpAST =
    (
        [Func {
            funcname = "dub",
            params = [],
            receive = "message",
            body = CaseOf (Par "message") [
                (["sender", "msg"]
                    ,Acts [
                        SendTo [Self, Par "msg"] (Par "sender"),
                        SendTo [Self, Par "msg"] (Par "sender")]
                )]
                (Acts [SendTo [Id "FaultyMessage"] (Id "println")])
            }
        ,Func {
            funcname = "half",
            params = ["state"],
            receive = "message",
            body = IfEq (Par "state") (Id "skip")
                (Acts [
                        Become (Id "half") [Id "return"],
                        SendTo [Id "SkippingMessage"] (Id "println")]
                )
                (CaseOf (Par "message") [
                    (["sender", "msg"]
                        ,Acts [
                            Become (Id "half") [Id "skip"],
                            SendTo [Self, Par "msg"] (Par "sender")]
                    )]
                    (Acts [SendTo [Id "FaultyMessage"] (Id "println")])
```

```
            )
        }
    ],
    [Create (Id "dubproc") (Id "dub") []
    ,Create (Id "halfproc") (Id "half") [Id "return"]
    ,SendTo [Id "halfproc", Id "foo"] (Id "dubproc")
    ]
)

gateKeeperAST :: Program
gateKeeperAST =
    (
        [Func {
            funcname = "printer",
            params = [],
            receive = "message",
            body = Acts [SendTo [Par "message"] (Id "println")]
            }
        ,Func {
            funcname = "gate",
            params = ["fst", "fstmsg"],
            receive = "message",
            body = CaseOf (Par "message") [
                (["snd", "sndmsg"]
                    , IfEq (Par "fst") (Id "none")
                        (Acts [Become (Id "gate") [Par "snd", Par "sndmsg"]])
                        (Acts [
                            SendTo [Par "fstmsg", Par "sndmsg"]
                                (Concat (Par "fst") (Par "snd"))
                            ,
                            Become (Id "gate") [Id "none", Id "none"]]
                        )
                )]
                (Acts [])
            }
        ,Func {
            funcname = "repeat",
            params = ["other"],
            receive = "message",
            body = Acts [
                SendTo [Par "message"] (Id "gatekeeper"),
                SendTo [Par "message"] (Par "other")]
            }
        ],
```

```haskell
        [Create (Id "foobar") (Id "printer") []
        ,Create (Id "gatekeeper") (Id "gate") [Id "none", Id "none"]
        ,Create (Id "repeater1") (Id "repeat") [Id "repeater2"]
        ,Create (Id "repeater2") (Id "repeat") [Id "repeater1"]
        ,SendTo [Id "foo", Id "Hello"] (Id "repeater1")
        ,SendTo [Id "bar", Id "World"] (Id "repeater2")
        ,SendTo [Id "foo", Id "Bye"] (Id "repeater1")
        ]
    )
-- }}} end of ASTs

-- Test cases that were handed out
fileTestCases :: [(String, Program)]
fileTestCases = [("helloWorld", helloWorldAST),
             ("cleanUp", cleanUpAST),
             ("gateKeeper", gateKeeperAST)]

testWithFile :: (String, Program) -> [IO Test]
testWithFile (f, ast) =
    [ do got <- parseFile filePath
         return $ TestCase $ assertEqual ("parseFile (" ++ f ++ ")")
                                         (Right ast) got
    , do src <- readFile filePath
         return $ TestCase $ assertEqual ("parseString (" ++ f ++ ")")
                                         (Right ast) $ parseString src
    ]
    where file = replaceExtension f ".soil"
          filePath = replaceBaseName "examples/" file

-- Does an invalid program give the correct Error?
invalidProgramTest :: IO Test
invalidProgramTest = return $ TestCase
                            $ assertEqual "invalid program"
                              (Left InvalidProgram)
                            $ parseString "case foo of _ : end"

-- Is concat left-associative?
concatAssocTest :: IO Test
concatAssocTest = return $ TestCase
                         $ assertEqual "concat associativity"
                           (Right ([], [SendTo [] (Concat (Concat (Par "a") (Par "b")
                         $ parseString "send () to a concat b concat c"

fileTests :: IO Test
```

```haskell
fileTests = fmap TestList $ sequence $ concatMap testWithFile fileTestCases

tests :: IO Test
tests = fmap TestList $ sequence [fileTests, invalidProgramTest, concatAssocTest]

main :: IO Counts
main = do t <- tests
          runTestTT t
```

A.3 Interpreting Soil

A.3.1 SoilInterp.hs

```haskell
--
-- Soil interpreter
-- Exam for Advanced Programming, B1-2012
-- Sebastian Paaske Tørholm <sebbe@diku.dk>
--

module SoilInterp
  ( runProgRR
  , runProgAll
  ) where

import SoilAst
import Control.Monad (void, liftM)
import Data.Maybe (fromMaybe)
import Data.List (find, intercalate, nub, partition)
import Control.Arrow ((***))

--
-- Part 1: Define a name environment
--
data NameEnv = NameEnv [(Name, [Ident])]
    deriving (Show)

-- Functions for insert and lookup

insertName :: NameEnv -> (Name, [Ident]) -> NameEnv
insertName (NameEnv ns) (n,i) =
    case lookup n ns of
        Nothing -> NameEnv $ (n,i) : ns
        Just _  -> error $ "name " ++ n ++ " already exists"
```

```haskell
lookupName :: NameEnv -> Name -> [Ident]
lookupName (NameEnv ns) n =
    fromMaybe (error $ "name " ++ n ++ " undefined") (lookup n ns)

--
-- Part 2: Define a function environment
--
data FuncEnv = FuncEnv [(Ident, Func)]
        deriving (Show)

-- Functions for insert and lookup

insertFunc :: FuncEnv -> (Ident, Func) -> FuncEnv
insertFunc (FuncEnv fs) (i,f) =
    case lookup i fs of
        Nothing -> FuncEnv $ (i,f) : fs
        Just _  -> error $ "function " ++ i ++ " already exists"

lookupFunc :: FuncEnv -> Ident -> Maybe Func
lookupFunc (FuncEnv fs) = flip lookup fs


--
-- Part 3: Degine a process and process queue
--
type Message = [Ident]

data Process = Process { procid    :: Ident
                       , function  :: Ident
                       , arguments :: [Ident]
                       , mailbox   :: [Message]
                       }
    deriving (Show)

data ProcessQueue = PQ [Process]
    deriving (Show)

-- Function for process modification
popMessage :: Process -> ProgramEvaluation (Maybe Message)
popMessage p = case mailbox p of
                  []     -> return Nothing
                  (m:mb) ->
                     do replaceProc (procid p) $ p { mailbox = mb }
                        return (Just m)
```

```haskell
hasMessages :: Process -> Bool
hasMessages = not . null . mailbox

changeProcFunc :: Process -> Ident -> [Ident] -> Process
changeProcFunc p f args = p { function = f , arguments = args }

getProcFunc :: Process -> (Ident, [Ident])
getProcFunc p = (function p, arguments p)

data ProgramState = PS { funcs :: FuncEnv
                       , procs :: ProcessQueue
                       }
    deriving (Show)

data ProgramEvaluation a = PE {
        runPE :: ProgramState -> (a, ProcessQueue)
    }

instance Monad ProgramEvaluation where
    return a = PE $ \ps -> (a, procs ps)
    pe >>= f = PE $ \ps -> let (a, pq) = runPE pe ps in
                           runPE (f a) $ ps { procs = pq }

instance Functor ProgramEvaluation where
    fmap f = (>>= return . f)

getProcQueue :: ProgramEvaluation ProcessQueue
getProcQueue = PE $ \ps -> let pq = procs ps in (pq, pq)

putProcQueue :: ProcessQueue -> ProgramEvaluation ()
putProcQueue pq = PE $ \_ -> ((), pq)

getFuncEnv :: ProgramEvaluation FuncEnv
getFuncEnv = fmap funcs getProgramState

getProc :: Ident -> ProgramEvaluation (Maybe Process)
getProc pid = do PQ pq <- getProcQueue
                 return $ find ((== pid) . procid) pq

getCurrentProc :: ProcessState -> ProgramEvaluation Process
getCurrentProc ps = let pidm = curPid ps in
                    case pidm of
                       Nothing  -> error "Current process doesn't exist."
                       Just pid ->
```

```haskell
                          do pm <- getProc pid
                             case pm of
                                Nothing -> error "Current process doesn't exist."
                                Just p  -> return p

replaceProc :: Ident -> Process -> ProgramEvaluation ()
replaceProc pid p = do PQ pq <- getProcQueue
                       putProcQueue $ PQ $ map (\p' -> if procid p' == pid
                                                        then p
                                                        else p') pq

pushProc :: Process -> ProgramEvaluation ()
pushProc p = do PQ pq <- getProcQueue
                putProcQueue $ PQ $ pq ++ [p]

popProc :: ProgramEvaluation Process
popProc = do PQ pq <- getProcQueue
             case pq of
                []   -> error "Process queue is empty!"
                p:ps -> do putProcQueue $ PQ ps
                           return p

getProgramState :: ProgramEvaluation ProgramState
getProgramState = PE $ \ps -> (ps, procs ps)

sendMessage :: Ident -> Message -> ProgramEvaluation ()
sendMessage pid m =
    do p <- getProc pid
       case p of
           Nothing -> sendMessage "errorlog"
                         ["Cannot send message to nonexistent process #" ++ pid]
           Just proc -> replaceProc pid $ proc { mailbox = mailbox proc ++ [m] }

--
-- Part 4: Define and implement a process step
--
data ProcessState = ProS { nameEnv :: NameEnv
                         , curPid  :: Maybe Ident
                         }
    deriving (Show)

-- use only on values that *must* be names
primToName :: Prim -> Name
primToName Self         = error "Invalid use of self where name expected."
```

```haskell
primToName (Id _)       = error "Invalid use of identifier where name expected."
primToName (Concat _ _) = error "Invalid use of concat where name expected."
primToName (Par n)      = n

-- gives [] if one of the identifiers in a concat resolves to 0 or 2+ values
resolveIdent :: ProcessState -> Prim -> [Ident]
resolveIdent _ (Id i)          = [i]
resolveIdent ps Self           = case curPid ps of
                                     Just pid -> [pid]
                                     Nothing  -> error "Self used outside of process."
resolveIdent ps (Par n)        = lookupName (nameEnv ps) n
resolveIdent ps (Concat p1 p2) =
    case (resolveIdent ps p1, resolveIdent ps p2) of
        ([i1], [i2]) -> [i1 ++ i2]
        _          -> []

runActOp :: ProcessState -> ActOp -> ProgramEvaluation ProcessState
runActOp ps aop =
    case aop of
        SendTo msg tgt       ->
            case resolveIdent ps tgt of
                [tpid] -> sendMessage tpid $ concatMap (resolveIdent ps) msg
                _      ->
                   sendMessage "errorlog"
                      [show tgt ++ " resolves to more than one ident in SendTo"]
        Create pidp fidp args ->
            do fe <- getFuncEnv
               case (resolveIdent ps fidp, resolveIdent ps pidp) of
                  ([fid], [pid]) ->
                    case lookupFunc fe fid of
                        Nothing ->
                          sendMessage "errorlog"
                            ["Cannot spawn process for nonexistent function " ++ fid]
                        Just f  ->
                            if length args /= length (params f)
                            then sendMessage "errorlog"
                                  ["Inccorect number of arguments passed to " ++ fid]
                            else let p = Process pid fid
                                           (concatMap (resolveIdent ps) args) [] in
                                 pushProc p
                  _      ->
                    sendMessage "errorlog"
                      ["A name resolves to more than one ident in Create"]
        Become fidp argsp ->
```

```
            do p <- getCurrentProc ps
               fe <- getFuncEnv
               case resolveIdent ps fidp of
                [fid] ->
                    let args = concatMap (resolveIdent ps) argsp in
                    case lookupFunc fe fid of
                        Nothing ->
                          sendMessage "errorlog"
                            ["Cannot become nonexistent function " ++ fid]
                        Just f  ->
                            if length args /= length (params f)
                            then sendMessage "errorlog"
                                 ["Inccorect number of arguments passed to " ++ fid]
                            else let p' = changeProcFunc p fid args in
                                 replaceProc (procid p) p'
                _ ->
                    sendMessage "errorlog"
                      ["Function name resolves to more than one ident in Become"]
    >> return ps

runExp :: ProcessState -> Expr -> ProgramEvaluation ()
runExp ps ex =
    let ne = nameEnv ps in
    case ex of
        CaseOf n pts def ->
            let v = lookupName ne $ primToName n in
            case find (\(p,_) -> length p == length v) pts of
                Nothing    -> runExp ps def
                Just (pt,e) ->
                    let ne' = foldl insertName ne $ zip pt $ map (:[]) v in
                    runExp (ps { nameEnv = ne' }) e
        IfEq p1 p2 thexp elexp ->
            let rp1 = resolveIdent ps p1
                rp2 = resolveIdent ps p2 in
            if length rp1 == 1 && length rp2 == 1 && rp1 == rp2
            then runExp ps thexp
            else runExp ps elexp
        Acts aops ->
            void $ foldl (\s a -> s >>= flip runActOp a) (return ps) aops

runFunc :: ProcessState -> Func -> ProgramEvaluation ()
runFunc ps f = runExp ps (body f)

evalFunc :: Ident -> [Ident] -> -- function id, arguments
```

```
             Ident -> [Ident] -> -- process id, message contents
             ProgramEvaluation ([String], [String])
evalFunc _ _ "println"  msg = return ([intercalate ":" msg], [])
evalFunc _ _ "errorlog" msg = return ([], [intercalate ":" msg])
evalFunc fid args pid msg =
    do fe <- getFuncEnv
       case lookupFunc fe fid of
           Nothing -> sendMessage "errorlog" ["No such function: " ++ fid] >>
                      return ([], [])
           Just f  -> let ne = NameEnv $ zip (receive f : params f)
                                             (msg : map (:[]) args)
                          ps = ProS ne (Just pid) in
                      runFunc ps f >> return ([],[])

processStep :: Ident -> ProgramEvaluation ([String], [String])
processStep pid =
    do p  <- getProc pid
       case p of
            Nothing -> do sendMessage "errorlog"
                             ["Cannot execute nonexistent process #" ++ pid]
                          return ([], [])
            Just p' -> do mm <- popMessage p'
                          case mm of
                             Nothing -> return ([], [])
                             Just m  -> uncurry evalFunc (getProcFunc p') pid m


--
-- Part 5: Define and implement the roind-robin algorithm
--

-- Not used, though it works fine; nextNonemptyProcessRR used instead
--nextProcessRR :: ProgramEvaluation Ident
--nextProcessRR = do p <- popProc
--                   pushProc p
--                   return $ procid p

nextNonemptyProcessRR :: ProgramEvaluation (Maybe Ident)
nextNonemptyProcessRR =
    do fp <- popProc
       pushProc fp
       if hasMessages fp
       then return $ Just $ procid fp
       else loop $ procid fp
```

```haskell
    where loop fpid = do p <- popProc
                         pushProc p
                         if procid p == fpid
                         then return Nothing
                         else if hasMessages p
                         then return $ Just $ procid p
                         else loop fpid

--
-- Part 6: Implement the round-robin evaluator
--
emptyProcessState :: ProcessState
emptyProcessState = ProS (NameEnv []) Nothing

initialProcessQueue :: ProcessQueue
initialProcessQueue = PQ [
        Process "println"  [] [] [],
        Process "errorlog" [] [] []
    ]

compileProgRR :: Int -> ProgramEvaluation ([String], [String])
compileProgRR 0 = return ([], [])
compileProgRR n =
    do pidm <- nextNonemptyProcessRR
       case pidm of
            Nothing  -> return ([], [])
            Just pid -> do (so, se)   <- processStep pid
                           (sos, ses) <- compileProgRR (n-1)
                           return (so ++ sos, se ++ ses)

initialPS :: [Func] -> ProgramState
initialPS fs = let fe = foldl (\e -> insertFunc e . \f -> (funcname f, f))
                              (FuncEnv []) fs in
               PS fe initialProcessQueue

evalInitialActOps :: [ActOp] -> ProgramEvaluation ()
evalInitialActOps aops = runExp emptyProcessState (Acts aops)

runProgRR :: Int -> Program -> ([String], [String])
runProgRR n (fs, aops) =
     fst $ runPE (evalInitialActOps aops >> compileProgRR n)
               $ initialPS fs

--
```

```haskell
-- Part 7: Implement a find all possible executions evaluator
--
nextProcAll :: ProgramEvaluation [Ident]
nextProcAll = do PQ pq <- getProcQueue
                 let (nep, ep) = partition hasMessages pq
                 return $ map procid $ case ep of
                    []     -> nep
                    (e:_) -> e:nep


runPid :: Int -> Ident -> ProgramEvaluation [([String], [String])]
runPid n pid = do (so, se) <- processStep pid
                  rems     <- compileProgAll n
                  return $ map ((so ++) *** (se ++)) rems

compileProgAll :: Int -> ProgramEvaluation [([String], [String])]
compileProgAll 0 = return [([],[])]
compileProgAll n = do pids <- nextProcAll
                      pq   <- getProcQueue
                      let curState = putProcQueue pq
                      liftM concat $ mapM ((curState >>) . runPid (n-1)) pids

runProgAll :: Int -> Program -> [([String], [String])]
runProgAll n (fs, aops) =
    nub $ fst $ runPE (evalInitialActOps aops >> compileProgAll n)
                    $ initialPS fs
```

A.3.2 SoilInterpTest.hs

```haskell
import SoilParser
import SoilInterp
import Test.HUnit
import Data.List (isPrefixOf)

helloWorldTest :: IO Test
helloWorldTest =
    do Right program <- parseFile "examples/helloWorld.soil"
       return $ TestList $ map TestCase [
           assertEqual "hw: run 0 steps = empty output" ([], [])
                     $ runProgRR 0 program,
           assertEqual "hw: run 2 steps = empty output" ([], [])
                     $ runProgRR 2 program,
           assertEqual "hw: run 3 steps = hello printed" (["Hello"], [])
                     $ runProgRR 3 program,
```

```
            assertEqual "hw: run 4 step = hello world printed" (["Hello", "World"], []
                      $ runProgRR 4 program,
            assertEqual "hw: run all 0 steps = empty output" [([],[])]
                      $ runProgAll 0 program,
            assertEqual "hw: run all 1 step = empty output" [([],[])]
                      $ runProgAll 1 program,
            assertEqual "hw: run all 2 steps" [([],[]), (["Hello"],[]), (["World"],[])
                      $ runProgAll 2 program,
            assertEqual "hw: run all 3 steps" [([],[]), (["Hello"],[]), (["World"],[])
                      $ runProgAll 3 program,
            assertEqual "hw: run all 4 steps" [([],[]), (["Hello"],[]), (["World"],[])
                                               (["Hello", "World"],[]),(["World", "Hel
                      $ runProgAll 4 program,
            assertEqual "hw: run all 7 steps" [([],[]), (["Hello"],[]), (["World"],[])
                                               (["Hello", "World"],[]),(["World", "Hel
                      $ runProgAll 7 program
          ]

gateKeeperTest :: IO Test
gateKeeperTest =
    do Right program <- parseFile "examples/gateKeeper.soil"
       return $ TestList $ map TestCase [
             assertEqual "gk: run 0 steps = empty output" ([], [])
                 $ runProgRR 0 program,
             assertEqual "gk: run 20 steps (output provided by Troels the oracle)"
                 (["Hello:World"], ["Cannot send message to nonexistent process #foofo
                 $ runProgRR 20 program,
             assertBool "gk: run 250 steps"
                 (let (o, e) = runProgRR 250 program in
                   all (== "Hello:World") o &&
                   all ("Cannot send message to nonexistent process" `isPrefixOf`) e)
          ]

cleanUpTest :: IO Test
cleanUpTest =
    do Right program <- parseFile "examples/cleanUp.soil"
       return $ TestList $ map TestCase [
             assertEqual "cleanUp: run 0 steps = empty output" ([], [])
                 $ runProgRR 0 program,
             assertBool "gk: run 250 steps"
                 (let (o, e) = runProgRR 250 program in
                   all (== "SkippingMessage") o && null e)
          ]
```

```haskell
tests :: IO Test
tests = fmap TestList $ sequence [helloWorldTest, gateKeeperTest, cleanUpTest]

main :: IO Counts
main = do t <- tests
          runTestTT t
```