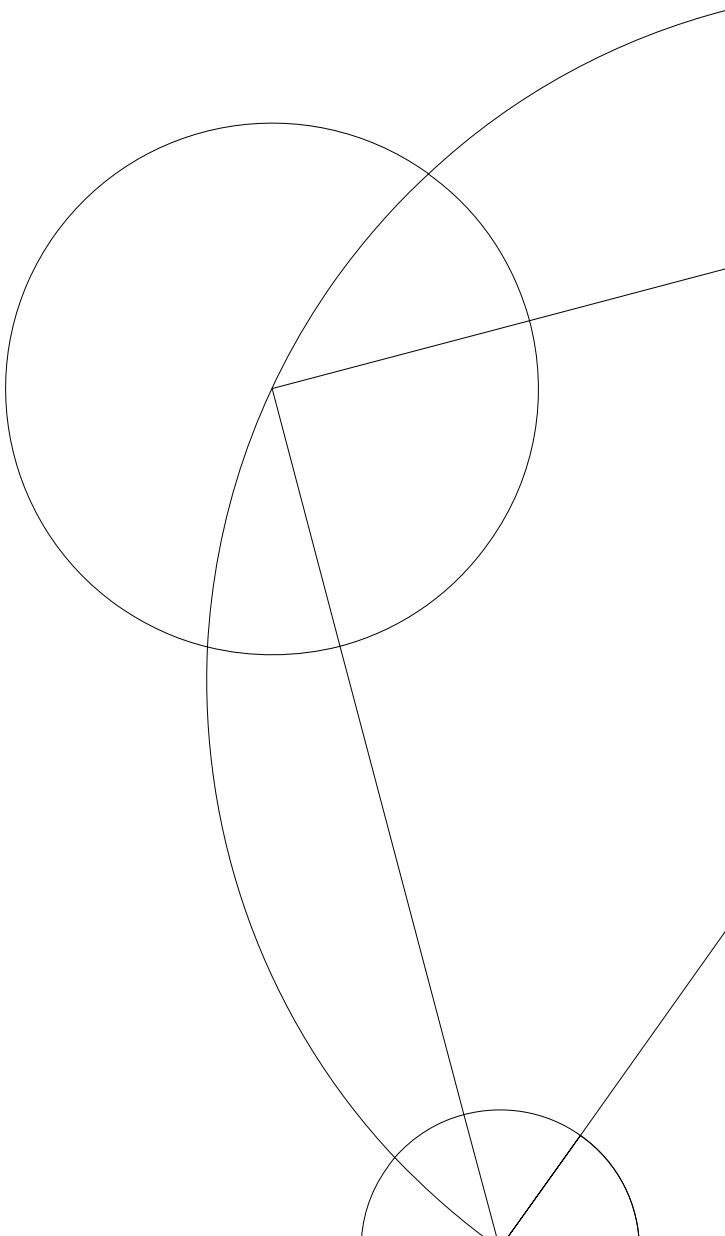
KØBENHAVNS UNIVERSITET

Advanced Programming Exam 2020

exam number 106



Datalogisk Institut6. november 2020

1 APQL: A Perplexing Query Language

1.1 Parser

In the parser part, we use Text.ParserCombinators.ReadP has the framework for I have a bit experience on it.

We implemented some basic functions like pComment, pConstant and pName to parse the Comment, Data and PName/VName, and we also implemented two functions, lexeme and lexeme1, to skip the spaces and comments in the string. The function lexeme1 is especially for skipping comments used as separators. Then we get functions like pTerm, pAtom, pCond, pRule and pProgram to parse terms, atoms, conds, rules and the whole program from the input.

There are some left recursion in the gramma of Cond, and we have to remove it. So we get the following refined gramma for the original Cond:

Cond ::= Cond1
| Cond1 Cond0

Cond0 ::= 'and' Cond1
| 'or' Cond1
| 'implies' Cond

Cond1 ::= Atom
| Term 'is' Term
| Term 'is not' Term
| 'true'
| 'false'
| 'not' Cond
| '(' Cond ')'

Here in our parser, if we cannot fully parse the string, then an user error would occur, saying the input is illegal.

1.2 Preprocessor

1.2.1 clausify

In this part we tranform Rules to Clauses. We separate the process to two parts, first, get [Atom]s and [Test]s from the Atom and Cond part of the Rule expressions, and then wrap them to be Clauses. Here we used a new type Tmp ::= ([Atom], [Test]) for convenience.

To get [Atom]s and [Test]s, we have a getTmp function, in which we listed all the possible situations of Cond, especially for 'CNot' expressions. When we would get a list of Tmp where there are only atoms, negated atoms, (possibly negated) is-comparisons, and trues in them. During the process we have to merge two Tmps into one Tmp when there is a CAnd, so we implemented a merge Tmp function to merge Tmps. If there is a (CNot CTure) in the expression, we will just return a Left "false"for this Clause since Clause with such expression would never be satisfied, we return Right [Tmp] if there is some satisfying Clauses.

To wrap Tmps to Clauses, We just use a wrapTmp function to format each Tmp with its corresponding atom to get a Clause, and we return a list of Clause.

After transforming, we have to do a variable verification. We check that in each Clause, the variables occurring in the clause head, and/or in one or more of the tests must appear in the list of non-negated atoms in the body with function checkClauses.

1.2.2 stratify

To help with partition all intensional predicates into strata such that the strata can be evaluated consecutively, we introduce a new data type ReferP ::= Neg PSpec.

For each PSpec in IDB, we look into all the clauses and if it is the head of some clause, we get all its negatively referred PSpecs with function lookForPos. That is, if there is some TNot expressions in the tests, we record all the referred atoms with its PSpec. After getting all the negatively referred PSpecs, we check if all of them have appeared in the lower strata with function canStayNeg, if not, then it cannot be in the current strata. We can get a list of PSpec from this process.

Getting the list if PSpec, we are going to check if all the positive references are in the current or lower strata. The function findStabilize is implemented for this purpose. We check if some PSpec should be removed from current strata for its positive reference is not in the current or lower strata with function formNewList2, if there is nothing to be removed, then the current strata is stable. If the current strata is empty, an user error would occur.

If all of the PSpecs have been put into some strata, then we have done the stratification.

1.3 Engine

We do the execution by strata.

First we do a initialization for the level of strata we work on. Then we look into this level of strata, for each PSpec, we do execution on each Clause to get a new EDB. We recursively do this until EDB does not change anymore.

If the Clause does not have any positive references, then the terms of it's head must only include TData expressions because of the variable verification. Then we just add it to the ETable.

Otherwise, we execute the non-negated atoms in the body of Clause. Here we introduce the data type ([VName], [Row]), we call it "variable table", and we use this to store all the variable names and their possible values. For each positively referred atom, we fetch the rows for it that satisfy the restriction of terms of itself from current EDB, that is, for an Atom like "Atom p [TVar x, Tvar x, TData "a"]", we remove the rows whose first and second element are not the same, or the third element is not "a". We do this in function getTab. Then we do conjunction on them. For each row from EDB, we check for each row in current variable table, whether the same variable have the same value, if not, this row should be removed from variable table, else, if there are new variables in the row from EDB, we add them to the variable table.

variables	x	y
row1	1	2
row2	3	5

Tabel 1: An example of variable table

Then we apply the tests to the variable table. For each row in variable table, for TNot expression, we take satisfying rows for the atom and check each row of it and each row of variable table, if they have the same variable with the same value, then remove the line of variable table. For TEq expressions, we just check if two terms are not the same, then we remove the corresponding row in variable table. And for TNeq expressions, if two terms are the same, then we remove the corresponding row in variable table.

After doing all the tests, the rows of the variable table is just the rows to be added to ETable. We just add them and update the EDB.

The length of the list of VName (or list of Term) and any corrsponding list of data (a row) should always be the same by definition. If we refer to some PSpec which is not in EDB, an user error would occur.

1.4 Test

We wrote a test file to test our code. For each possible significant form of input, we wrote a test case for it. We mix some of them together to make the test tree lightweight, like, the test case named "parse Rule=Atom if Cond, Cond=Atom"is a case whose input has the Rule in the form of Atom if Cond, and the Cond is just an Atom. The test case of execute has a multi-level strata. Also, we test all the possible user errors of our code.

In some cases, there are more than one correct answer. Such as, in stratify, the order of the PSpec in a certain strata level does not matter, so we implemented checkPSpec and checkSta function, to check whether the result strata has the same length with that of a correct one, and to check in each level of strata, the set of PSpec in result is the same as the set in a correct answer, and we use Data.Set.fromList to implement it.

I have meet all the requests and I have passed all the tests I generate, I think I worth a full score.

2 Mail Filter

2.1 Function Implementation

2.1.1 The structure of the solution

We build a Mail Filter Server with gen_server behaviour. We use the same module for the server and the filters, I found we can separate them into two modules, but I don't have so much time to rewrite my code. So here we have the same callback functions for both server and filter, to better distinguish them, the handle_call or handle_cast functions with state variable name "State0"is only used by filters.

We start the server by calling start(Cap), now we only support infinite capacity, in fact, the capacity will be set to infinite no matter what Cap is. The server, as described in Table 2, keeps track of an increasing number Count, which would be the id (which is also MR) of any mail we add to the server, and two maps, one for storing the label of filter, the filter, the initial data and whether it is default, and one for storing the id of the mail and the corresponding configuration.

items in State	explanation
Count	id (MR) for next Mail
Filters	#{Label of a Filter => {Filter, Initial Data, is Default}}
Mails	#{id of a Mail (MR) => {Mail, [{Label of a Filter, Result of the Filter}]}}

Tabel 2: the State

The start(Cap) API would return a pid of the server as MS.

The server can be stopped by calling stop(MS). This will terminate all the existing filters and then the server itself, and returns the state of all the mails. If a filter is still under calculation (during the Fun(Mail,Data) process), it would terminate after the calculation finishes.

When we call default(MS, Label, Filt, Data), we modify the Filters in the State of the server. We add a mapping from the new Label to the {Filt, Data, 1} if the Label is not a key in Filters.

The get_config(MR) API just ask the server to look for the configuration from the Mails.

When we call add_mail(MS, Mail), we give an id (MR) Count to the Mail, then increase Count by 1, we add the information about this Mail, including the default filters, to the Mails of the State in server. Then for all the filters the Mail has, we start a filter concurrently for each of them.

When we call enough(MR), we will stop all the running filters for this mail and then remove the mail from the server, that is, MR will be removed from Mails.

And if we call add_filter(MR, Label, Filt, Data), we will modify Mails and Filters, and start a filter for it if Label is not a key in Filters.

2.2 Filters

We only implemented simple and chain filters. The filter functions would take a mail and an initial data as input. I tried to implement timeout, but I cannot deal with the situation that it gets a timeout.

When starting a filter, we would check if it is already running by checking if its global name has been registered, if so, we would stop the former one and start a new one. The global name is {MR, Label} for a filter.

Each time we start a labelled Filter for some mail, we pass the current State to the Filter to facilitate calculation, although the needed part is only the corresponding filters, mail and initial data.

We run the simple filter function to get a result, or run the chain filter functions in a row to get the final result, then write it back to the State in server.

When a filter terminates (both normally and killed by other process), its global name would be unregistered. If a filter changes the Mail, all other filters associated with this mail would restart concurrently.

2.3 Communication

In this module we have a mail filter server and several filters, the filter write its result back to the server by calling gen_server:cast(?MODULE, update, MR, Label, Mail, Res) to update the State in server.

When a labelled filter is started, the server would pass the current State on to the filter, for the filter needs the filter information and mail information in State.

There is no communication between filters.

If a filter changes the mail, server would know it and then restart all the other filters of the mail. When there are more than one filters function in a filter that change the mail, which would result in a non-termination, we can still go on with other filters, we can call get_config or stop and get corresponding return value.

2.4 Robustness

We make the function enough to return ok even if the MR we want to remove from the server doesn't exist, so it would return ok in any case and never crash the server.

The function get_config would return an error message if the corresponding mail doesn't exist in our mail state list, and the server would not crash.

The function add_filter would return an info message to inform the user if the Label has been registered, or the MR doesn't exist, and would not do such add filter operation, the server would not crash.

2.5 Testing

2.5.1 quick check

Run quick check by running "**c(test_mailfilter), eqc:module(test_mailfilter).**". I think I implemented wellbehaved_filter_fun/0 and filter(FunGen) functions correctly. I also implemented a prop_filter to show all the filters I generated. However I cannot use generator to implement prop_mail_is_sacred() and prop_consistency(), so I just wrote a fixed case for them. I test such properties in our unit test, too.

2.5.2 unit test

We implemented an unit test in file test_unit.erl. We can run the test by "**c(mailfilter), c(test_unit), test_unit:test_all().**".

We first test the start and stop functions, and then default function. For default function, since we only implemented simple and chain filters, we only run such tests and make sure they work well.

After that we check add_mail function work well by adding some mails to the server, and check when stop is called whether the mail state list has a length equal to the number of mails we added.

Then we test enough function. Here since we make the function to return ok even if the MR we want to remove from the server doesn't exist, it would return ok in any case, and only remove MR from the server if MR is in the server (which we think is successful). We check it returns ok and check the length of mail state list when stop is called, the length should be the same as the number of mails we added minus the number of successful enough operations.

The function get_config would return an error message if the MR doesn't exist in our mail state list, so we test a successful get_config operation and a get_config operation that would return a error message, however the server would not crash.

The function add_filter would return an info message if the Label has been registered, or the MR doesn't exist. We test such unsuccessful cases and successful cases with simple filter and chain filter.

We also implemented a mytest_registered_filter to show that if we register filters with the same label, only the first one can be registered successfully. We check this by calling gen_server:call(M, get_all_state) function we implemented to get all the current status of MR, filters and mails, and check if the list of filters has a length equal to the number of successful operations that would cause a filter to be registered.

Function mytest_add_filter_info and mytest_add_filter_result are for checking the robustness of add_filter function. That is, the info we mentioned above, and we check if the mail status after running the filters is correct when stop is called.

2.6 Conclusion

Since I did something more than the basic requirements for implementing mailfilter, and the code passed all the tests I generated, I think I should get a 3/7 grade.

Appendix

ParserImpl.hs

```
-- Put your Parser implmenetation in this file.
module ParserImpl where

import Text.ParserCombinators.ReadP as RP
import Data.Char
import Control.Applicative
import Types
-- probably more imports here

reserveWord :: [String]
reserveWord = ["and", "false", "if", "implies", "in", "is", "not", "or", "true", "unless"]

lexeme :: ReadP a -> ReadP a
lexeme p = do skipSpaces;
              skipMany pComment;
              a <- p;
              skipMany pComment;
              skipSpaces;
              return a

lexeme1 :: ReadP a -> ReadP a
lexeme1 p = do skipSpaces;
               skipMany pComment;
               a <- p;
               skipMany1 pComment;
               skipSpaces;
               return a

pComment :: ReadP String
pComment = (do
        s <- RP.between (string "(*") (string "*)") (RP.many get);
        return s)

pProgram :: ReadP Program
pProgram = (do
        rule <- pRule;
        _ <- lexeme (string ".");
        rules <- pProgram;
        return (rule:rules))
        <++ (do return [])

pRule :: ReadP Rule
pRule = (do
        atom <- pAtom;
        _ <- lexeme (string "if ");
        con <- pCond;
        return (Rule atom con))
        <++ (do
                atom <- pAtom;
                _ <- lexeme1 (string "if");
                con <- pCond;
                return (Rule atom con))
        <++ (do
                atom <- pAtom;
                _ <- lexeme (string "unless ");
```

```
                con <- pCond;
                return (Rule atom (CNot con)))
        <++ (do
                atom <- pAtom;
                _ <- lexeme1 (string "unless");
                con <- pCond;
                return (Rule atom (CNot con)))
        <++ (do
                atom <- pAtom;
                return (Rule atom CTrue))

pCond :: ReadP Cond
pCond = (do
        con <- pCond1;
        pCond0 con)
        <++ pCond1

pCond0 :: Cond -> ReadP Cond
pCond0 con = (do
        _ <- lexeme (string "and ");
        c <- pCond1;
        pCond0 (CAnd con c))
        <++ (do
                _ <- lexeme1 (string "and");
                c <- pCond1;
                pCond0 (CAnd con c)
                )
        <++ (do
                _ <- lexeme (string "or ");
                c <- pCond1;
                pCond0 (COr con c))
        <++ (do
                _ <- lexeme1 (string "or");
                c <- pCond1;
                pCond0 (COr con c))
        <++ (do
                _ <- lexeme (string "implies ");
                c <- pCond;
                pCond0 (COr (CNot con) c))
        <++ (do
                _ <- lexeme1 (string "implies");
                c <- pCond;
                pCond0 (COr (CNot con) c))
        <++ return con


pCond1 :: ReadP Cond
pCond1 = (do
        atom <- pAtom;
        return (CAtom atom))
        <++ (do
                t1 <- pTerm;
                _ <- lexeme (string "is ");
                t2 <- pTerm;
                return (CEq t1 t2))
        <++ (do
                t1 <- pTerm;
                _ <- lexeme1 (string "is");
```

```
                t2 <- pTerm;
                return (CEq t1 t2))
        <++ (do
                t1 <- pTerm;
                _ <- lexeme (string "is not ");
                t2 <- pTerm;
                return (CNot (CEq t1 t2)))
        <++ (do
                t1 <- pTerm;
                _ <- lexeme1 (string "is not");
                t2 <- pTerm;
                return (CNot (CEq t1 t2)))
        <++ (do
                _ <- lexeme (string "true");
                return CTrue)
        <++ (do
                _ <- lexeme (string "false");
                return (CNot CTrue))
        <++ (do
                _ <- lexeme (string "not ");
                con <- pCond;
                return (CNot con))
        <++ (do
                _ <- lexeme1 (string "not");
                con <- pCond;
                return (CNot con))
        <++ (do
                _ <- lexeme (string "(");
                con <- pCond;
                _ <- lexeme (string ")");
                return con)

pAtom :: ReadP Atom
pAtom = do
        name <- pName;
        _ <- lexeme (char '(');
        term <- pTermz;
        _ <- lexeme (char ')');
        return (Atom name term)

pTermz :: ReadP [Term]
pTermz = pTerms
        <++ (do return [])

pTerms :: ReadP [Term]
pTerms = (do
        t <- pTerm;
        _ <- lexeme (char ',');
        ts <- pTerms;
        return (t:ts))
        <++(do
                t <- pTerm;
                return [t])

pTerm :: ReadP Term
pTerm = (do
        vName <- pName;
        return (TVar vName))
```

```
        <++
        (do
                tData <- pConstant;
                return (TData tData))

pName :: ReadP String
pName = lexeme $ do
        x <- satisfy (\c -> isLetter c);
        xs <- munch (\c -> isLetter c || isDigit c || c == '_');
        let vName = x:xs;
        if vName `notElem` reserveWord
                then return vName
        else
                fail "Use reserved word as an VName."

pConstant :: ReadP Data
pConstant = lexeme $ do
        _ <- string "\"";
        content <- RP.many (
                satisfy (\x -> isPrint x && x /= '\"')
                <|> (do _ <- string "\"\""; return ('\"') )
                );
        _ <- string "\"";
        let c = [x | x <- content, x `notElem` "\NUL"];
        return c

parseString :: String -> Either ErrMsg Program
parseString s = if (readP_to_S pProgram s == [])
        then
                Left (EInternal "Parse failed, no results get.")
        else (do
                let tmp = [x | x <- readP_to_S pProgram s, snd x == ""];
                if tmp == []
                        then
                                Left (EUser "Parse failed. Illegal program expression.")
                        else
                                Right (fst (head tmp))
                )

seeParse :: String -> [(Program, String)]
seeParse s = [x | x <- readP_to_S pProgram s, snd x == ""]

seeParses :: String -> [(Program, String)]
seeParses s = [x | x <- readP_to_S pProgram s]
```

PreprocessorImpl.hs

```
-- Put your Preprocessor implementation in this file.
module PreprocessorImpl where

import Types
import Data.Set as S
-- Probably more imports here

type Tmp = ([Atom],[Test])

data ReferP = Neg PSpec
        deriving (Eq, Show)

clausify :: Program -> Either ErrMsg IDB
clausify p =
        let clauses = clausifyP p in
                if checkClauses clauses
                        then Right (IDB (makeDB p) clauses)
                else
                        Left (EUser "Doesn't obey the variable restrictions .")


stratify :: IDB -> [PSpec] -> Either ErrMsg [[PSpec]]
stratify (IDB ps cs) eps = case strata ps cs [eps] of
        (Left a) -> Left a
        (Right l) -> Right (tail l)


strata :: [PSpec] -> [Clause] -> [[PSpec]] -> Either ErrMsg [[PSpec]]
strata [] _ sta = Right sta
strata l c sta =
        let newList = formNewList1 l c sta in
                if newList == []
                        then Left (EUser "Stratify empty.")
                else
                        case findStabilize newList c sta of
                                (Left _) -> Left (EUser "Cannot stabilize.")
                                (Right ll) -> strata (getRemaining (sta ++ [ll]) l) c (sta ++ [ll
                                    ])

getRemaining :: [[PSpec]] -> [PSpec] -> [PSpec]
getRemaining [] l = l
getRemaining (s:ss) l = getRemaining ss (getRemainingHelp s l)


getRemainingHelp :: [PSpec] -> [PSpec] -> [PSpec]
getRemainingHelp [] l = l
getRemainingHelp (x:xs) l = getRemainingHelp xs (removeFromList x l)


formNewList1 :: [PSpec] -> [Clause] -> [[PSpec]] -> [PSpec]
formNewList1 [] _ _ = []
formNewList1 (p:ps) c sta =
        let ref = lookForNeg p c in
                if canStayNeg ref sta
                        then p:(formNewList1 ps c sta)
                else
                        formNewList1 ps c sta
```

```
-- check empty after doing

formNewList2 :: [PSpec] -> [PSpec] -> [Clause] -> [[PSpec]] -> Either String [PSpec]
formNewList2 [] _ _ _ = Left "OK."
formNewList2 (p:ps) l c sta =
        let ref = lookForPos p c in
                if canStayPos ref l sta
                        then formNewList2 ps l c sta
                else
                        Right (removeFromList p l)

findStabilize :: [PSpec] -> [Clause] -> [[PSpec]] -> Either String [PSpec]
findStabilize l c sta =
        case formNewList2 l l c sta of
                (Left _) -> Right l
                (Right ll) ->
                        if ll == []
                                then Left "Cannot stabilize."
                        else findStabilize ll c sta

removeFromList :: PSpec -> [PSpec] -> [PSpec]
removeFromList p [] = []
-- removeFromList p (p:xs) = xs
removeFromList p (x:xs) =
        if p == x
                then xs
        else x:(removeFromList p xs)

canStayPos :: [PSpec] -> [PSpec] -> [[PSpec]] -> Bool
canStayPos [] _ _ = True
canStayPos (r:rs) p sta =
        if isIn r [p] || isIn r sta
                then canStayPos rs p sta
        else
                False

canStayNeg :: [ReferP] -> [[PSpec]] -> Bool
canStayNeg [] _ = True
canStayNeg ((Neg a):rs) p =
        if isIn a p
                then canStayNeg rs p
        else
                False


isIn :: PSpec -> [[PSpec]] -> Bool
isIn _ [] = False
isIn a (p:ps) =
        if a `elem` p
                then True
        else
                isIn a ps

lookForPos :: PSpec -> [Clause] -> [PSpec]
lookForPos _ [] = []
lookForPos (name, i) ((Clause (Atom name1 l) a t):cs) =
        if name == name1
```

```
                then
                        if length l == i
                                then (atoms2pSpecs a) ++ (lookForPos (name, i) cs)
                        else
                                lookForPos (name, i) cs
        else
                lookForPos (name, i) cs
-- lookFor p (_:cs) = lookFor p cs

lookForNeg :: PSpec -> [Clause] -> [ReferP]
lookForNeg _ [] = []
lookForNeg (name, i) ((Clause (Atom name1 l) a t):cs) =
        if name == name1
                then
                        if length l == i
                                then (lookForNegative t) ++ (lookForNeg (name, i) cs)
                        else
                                lookForNeg (name, i) cs
        else
                lookForNeg (name, i) cs

lookForNegative :: [Test] -> [ReferP]
lookForNegative [] = []
lookForNegative ((TNot a):xs) = (Neg (atom2pSpec a)):(lookForNegative xs)
lookForNegative (_:xs) = lookForNegative xs

atom2pSpec :: Atom -> PSpec
atom2pSpec (Atom a l) = (a, (length l))

atoms2pSpecs :: [Atom] -> [PSpec]
atoms2pSpecs [] = []
atoms2pSpecs ((Atom a l):as) = (a, (length l)) :( atoms2pSpecs as)

checkOne :: [PSpec] -> [PSpec] -> Bool
checkOne [] _ = True
checkOne (x:xs) l =
        if x `notElem` l
                then checkOne xs l
        else
                False

checkClauses :: [Clause] -> Bool
checkClauses [] = True
checkClauses (c:cs) =
        if checkVariable c
                then checkClauses cs
        else False

checkVariable :: Clause -> Bool
checkVariable (Clause (Atom name t) pos tes) =
        checkIn (mergeVName (getVFromAtom t) (getThr tes)) (getSec pos)


checkIn :: [VName] -> [VName] -> Bool
checkIn [] sec = True
checkIn (v:vs) sec =
        if v `elem` sec
                then checkIn vs sec
```

```
        else  False


getThr ::  [Test]  −> [VName]
getThr []  =  []
getThr ((TEq (TVar v) (TData _)):ts) = mergeVName [v] (getThr ts)
getThr ((TEq (TData _) (TVar v)):ts) = mergeVName [v] (getThr ts)
getThr ((TEq (TVar u) (TVar v)):ts) = mergeVName [u, v] (getThr ts)
getThr ((TEq (TData _) (TData _)):ts) = getThr ts
getThr ((TNeq (TVar v) (TData _)):ts) = mergeVName [v] (getThr ts)
getThr ((TNeq (TData _) (TVar v)):ts) = mergeVName [v] (getThr ts)
getThr ((TNeq (TVar u) (TVar v)):ts) = mergeVName [u, v] (getThr ts)
getThr ((TNeq (TData _) (TData _)):ts) = getThr ts
getThr ((TNot (Atom name t)):ts) = mergeVName (getVFromAtom t) (getThr ts)

getSec  ::  [Atom] −> [VName]
getSec  []  =  []
getSec  ((Atom name t):as) = mergeVName (getVFromAtom t) (getSec as)

getVFromAtom :: [Term] −> [VName]
getVFromAtom [] = []
getVFromAtom ((TVar vname):ts) = vname:(getVFromAtom ts)
getVFromAtom ((TData _):ts) = getVFromAtom ts

mergeVName :: [VName] −> [VName] −> [VName]
mergeVName l1 l2 = S.elems (S.fromList (l1++l2))


makeDB :: Program −> [PSpec]
makeDB [] = []
makeDB ((Rule (Atom name xs) cond):rs) =
        let  l  = (makeDB rs) in
                let  i  = (name, (length xs)) in
                        if  i  `notElem` l
                                then i:l
                        else  l

clausifyP  ::  Program −> [Clause]
clausifyP  []  =  []
clausifyP  (r:rs)  =  (getClause r) ++ (clausifyP rs)

getClause  ::  Rule −> [Clause]
getClause (Rule atom cond) =
        case  (getTmp cond) of
                (Left  _)  −>  []
                (Right tmps) −> wrapTmp tmps atom


wrapTmp :: [Tmp] −> Atom −> [Clause]
wrapTmp [] _ = []
wrapTmp ((atoms, tests):tmps) atom = (Clause atom atoms tests):(wrapTmp tmps atom)

getTmp :: Cond −> Either String [Tmp]
getTmp (CAtom atom) = Right [([atom],[])]
getTmp (CEq t1 t2) = Right [([],[TEq t1 t2])]
getTmp CTrue = Right [([],[])]
getTmp (CAnd c1 c2) =
        case  (getTmp c1) of
```

```
                (Left _) -> Left "false"
                (Right tmps) -> case (getTmp c2) of
                        (Left _) -> Left "false"
                        (Right tmps1) -> Right (mergeTmp tmps tmps1)

getTmp (COr c1 c2) =
        case (getTmp c1) of
                (Left _) -> case (getTmp c2) of
                        (Left _) -> Left "false"
                        (Right tmps1) -> Right tmps1
                (Right tmps) -> case (getTmp c2) of
                        (Left _) -> Right tmps
                        (Right tmps1) -> Right (tmps ++ tmps1)

getTmp (CNot (CAtom atom)) = Right [([],[TNot atom])]
getTmp (CNot (CEq t1 t2)) = Right [([],[TNeq t1 t2])]
getTmp (CNot (CNot cond)) = getTmp cond
getTmp (CNot (CAnd c1 c2)) = getTmp (COr (CNot c1) (CNot c2))
getTmp (CNot (COr c1 c2)) = getTmp (CAnd (CNot c1) (CNot c2))
getTmp (CNot CTrue) = Left "false"


mergeTmp :: [Tmp] -> [Tmp] -> [Tmp]
mergeTmp [] tmps = []
mergeTmp (x:xs) tmps = (connTmp x tmps) ++ (mergeTmp xs tmps)

connTmp :: Tmp -> [Tmp] -> [Tmp]
connTmp (atoms,tests) [] = []
connTmp (atoms,tests) ((a,t):tmps) = ((atoms++a),(tests++t)):(connTmp (atoms,tests) tmps)
```

EngineImpl.hs

```
-- Put your Preprocessor implementation in this file
module EngineImpl where

import Types
import Data.Set as Set
-- Probably more imports here

execute :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
execute idb [] edb = Right edb
execute (IDB f clauses) (sta:ss) edb = execute1 (IDB f clauses) (sta:ss) (initSta f edb)

-- execute0 :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
-- execute0 idb [] edb = Right edb
-- execute0 idb (sta:ss) edb = execute1 idb (sta:ss) (initSta sta edb)


execute1 :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
execute1 idb [] edb = Right edb
execute1 idb (sta:ss) edb =
        case doSta idb sta edb of
                (Left a) -> Left a
                (Right newEDB) ->
                        if newEDB == edb
                                then execute1 idb ss newEDB
                        else
                                execute1 idb (sta:ss) newEDB


initSta :: [PSpec] -> EDB -> EDB -- use it!
initSta [] edb = edb
initSta (p:ps) edb = (p, Set.empty):(initSta ps edb)

-- edb has been initialized
doSta :: IDB -> [PSpec] -> EDB -> Either ErrMsg EDB
doSta idb [] edb = Right edb
doSta (IDB f clauses) (p:ps) edb =
        case doPspec clauses p edb of
                (Left a) -> Left a
                (Right newEDB) ->
                        doSta (IDB f clauses) ps newEDB


doPspec :: [Clause] -> PSpec -> EDB -> Either ErrMsg EDB
doPspec [] pspec edb = Right edb
doPspec ((Clause (Atom name terms) atoms tests):cs) pspec edb =
        let p = (name, length(terms)) in
                if p == pspec
                        then
                                case doClause (Clause (Atom name terms) atoms tests) edb of
                                        (Left a) -> Left a
                                        (Right newEDB) -> doPspec cs pspec newEDB
                else
                        doPspec cs pspec edb


doClause :: Clause -> EDB -> Either ErrMsg EDB
doClause (Clause (Atom pname terms) atoms tests) edb =
```

```
        case getETable (Clause (Atom pname terms) atoms tests) edb of
                (Left a) -> Left a
                (Right table) ->
                        Right (updateEDB (pname, length(terms)) table edb)

-- after each clause
updateEDB :: PSpec -> ETable -> EDB -> EDB
updateEDB pspec table [] = [(pspec, table)]
updateEDB pspec table ((p, t):es) =
        if p == pspec
                then (p, Set.union table t):es
        else
                (p, t):(updateEDB pspec table es)


getETable :: Clause -> EDB -> Either ErrMsg ETable
getETable (Clause (Atom pname terms) atoms tests) edb =
        if length atoms == 0
                then
                        Right (rows2Set [(directAdd terms)])

        else
                case getRows atoms edb of
                        (Left a) -> Left a
                        (Right (vnames, rows)) ->
                                case refine (vnames, rows) tests edb of
                                        (Left a) -> Left a
                                        (Right rs) ->
                                                case storeRows vnames rs pname terms of
                                                        (Left a) -> Left a
                                                        (Right resRow) ->
                                                                Right (rows2Set resRow)

directAdd :: [Term] -> Row
directAdd [] = []
directAdd ((TData a):ts) = a:(directAdd ts)


-- edb has been initialized
getRows :: [Atom] -> EDB -> Either ErrMsg ([VName], [Row]) -- [Atom] cannot be empty if
    there is variable in 1/3
getRows ((Atom pname terms):[]) edb =
        let pspec = (pname, length(terms)) in
                case getEdb pspec edb of
                        (Left a) -> Left a
                        (Right tab) ->
                                Right (firstVR terms (getTab terms (Set.elems tab)))
getRows ((Atom pname terms):as) edb =
        let pspec = (pname, length(terms)) in
                case getEdb pspec edb of
                        (Left a) -> Left a
                        (Right tab) ->
                                case getRows as edb of
                                        (Left a) -> Left a
                                        (Right res) ->
                                                Right (resAnd res (terms, (getTab terms (Set.
                                                    elems tab))))
```

```
-- after this, do test

rows2Set :: [Row] -> ETable
rows2Set rows = Set.fromList rows

-- row is refined (refine)
storeRows :: [VName] -> [Row] -> PName -> [Term] -> Either ErrMsg [Row]
storeRows vnames [] pname terms = Right []
storeRows vnames (r:rs) pname terms =
        case store vnames r pname terms of
                (Left a) -> Left a
                (Right row) ->
                        case storeRows vnames rs pname terms of
                                (Left a) -> Left a
                                (Right rows) -> Right (row:rows)

store :: [VName] -> Row -> PName -> [Term] -> Either ErrMsg Row
store vnames row pname [] = Right []
store vnames row pname (t:ts) =
        case t of
                (TData d) ->
                        case store vnames row pname ts of
                                (Left a) -> Left a
                                (Right l) -> Right (d:l)
                (TVar v) ->
                        case vInVs v vnames row of
                                (Left a) -> Left (EInternal "Variable name appear in param1
                                    but not param2.")
                                (Right d) ->
                                        case store vnames row pname ts of
                                                (Left a) -> Left a
                                                (Right l) -> Right (d:l)


-- doing test
refine :: ([VName], [Row]) -> [Test] -> EDB -> Either ErrMsg [Row]
refine (vs, []) tests edb = Right []
refine (vs, (r:rs)) tests edb =
        case checkStay vs r tests edb of
                (Left a) -> Left a
                (Right False) -> refine (vs, rs) tests edb
                (Right True) ->
                        case refine (vs, rs) tests edb of
                                (Left a) -> Left a
                                (Right rows) -> Right (r:rows)



checkStay :: [VName] -> Row -> [Test] -> EDB -> Either ErrMsg Bool
checkStay vs row [] edb = Right True
checkStay vs row (t:ts) edb =
        case t of
                (TNot (Atom pname terms)) ->
                        let pspec = (pname, length(terms)) in
                                case getEdb pspec edb of
                                        (Left a) -> Left a
                                        (Right tab) ->
                                                let (vnames, rows) = (firstVR terms (getTab
```

```
                                    terms (Set.elems tab))) in
                                if calTNot vs row vnames rows
                                        then Right False
                                else checkStay vs row ts edb
(TEq t1 t2) ->
        case t1 of
                (TData d1) ->
                        case t2 of
                                (TData d2) ->
                                        if d1 == d2
                                                then checkStay vs row ts edb
                                        else
                                                Right False
                                (TVar v2) ->
                                        case (vInVs v2 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d) ->
                                                        if d == d1
                                                                then checkStay
                                                                    vs row ts
                                                                    edb
                                                        else Right False
                (TVar v1) ->
                        case t2 of
                                (TData d2) ->
                                        case (vInVs v1 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d) ->
                                                        if d == d2
                                                                then checkStay
                                                                    vs row ts
                                                                    edb
                                                        else Right False
                                (TVar v2) ->
                                        case (vInVs v1 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d1) ->
                                                        case (vInVs v2 vs row) of
                                                                (Left _) -> Left
                                                                    (EInternal "
                                                                    Variable
                                                                    name appear
                                                                     in param3
                                                                    but not
                                                                    param2.")
                                                                (Right d2) ->
                                                                        if d1
                                                                            ==
                                                                            d2
                                                                            then

                                                                                checkStay
```

```
                                                                                vs

                                                                                row

                                                                                ts

                                                                                edb

                                                                          else
                                                                              Right

                                                                                False
(TNeq t1 t2) ->
        case t1 of
                (TData d1) ->
                        case t2 of
                                (TData d2) ->
                                        if d1 == d2
                                                then Right False
                                        else
                                                checkStay vs row ts edb
                                (TVar v2) ->
                                        case (vInVs v2 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d) ->
                                                        if d == d1
                                                                then Right False
                                                        else checkStay vs row ts
                                                            edb
                (TVar v1) ->
                        case t2 of
                                (TData d2) ->
                                        case (vInVs v1 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d) ->
                                                        if d == d2
                                                                then Right False
                                                        else checkStay vs row ts
                                                            edb
                                (TVar v2) ->
                                        case (vInVs v1 vs row) of
                                                (Left _) -> Left (EInternal "
                                                    Variable name appear in
                                                    param3 but not param2.")
                                                (Right d1) ->
                                                        case (vInVs v2 vs row) of
                                                                (Left _) -> Left
                                                                    (EInternal "
                                                                    Variable
                                                                    name appear
                                                                     in param3
                                                                    but not
                                                                    param2.")
```

```
                                                    (Right d2) ->
                                                        if d1
                                                            ==
                                                            d2
                                                                then

                                                                    Right

                                                                    False

                                                    else
                                                        checkStay
                                                         vs
                                                        row
                                                        ts
                                                        edb
```

```
calTNot :: [VName] -> [Data] -> [VName] -> [Row] -> Bool -- true if inside, which should
    be removed
calTNot vnames1 row vnames2 [] = False
calTNot vnames1 row vnames2 (r:rs) =
        if helpTNot vnames1 row vnames2 r
                then True
        else calTNot vnames1 row vnames2 rs

helpTNot :: [VName] -> [Data] -> [VName] -> [Data] -> Bool -- True if the same, should
    remove
helpTNot [] [] vnames2 row = True
helpTNot (v:vs) (d:ds) vnames2 row =
        case (vInVs v vnames2 row) of
                (Left _) -> helpTNot vs ds vnames2 row
                (Right d1) ->
                        if d1 == d
                                then helpTNot vs ds vnames2 row
                        else False


vInVs :: VName -> [VName] -> [Data] -> Either String Data
vInVs v [] [] = Left "Not in."
vInVs v (v1:vs) (d1:ds) =
        if v == v1
                then Right d1
        else vInVs v vs ds


getEdb :: PSpec -> EDB -> Either ErrMsg ETable
getEdb pspec [] = Left (EUser "Table not found.")
getEdb pspec ((p, t):ts) =
        if pspec == p
                then Right t
        else
                getEdb pspec ts

-- where's the first?
firstVR :: [Term] -> [Row] -> ([VName], [Row])
firstVR terms rows = ((combineV [] terms), (andHelp [] [] terms rows))
```

```
resAnd :: ([VName], [Row]) -> ([Term], [Row]) -> ([VName], [Row])
resAnd (names, rows1) (terms, rows2) = (combineV names terms, calAnd names rows1 terms
    rows2)

calAnd :: [VName] -> [Row] -> [Term] -> [Row] -> [Row]
calAnd names [] terms rows = []
calAnd names (r:rs) terms rows = (andHelp names r terms rows)++(calAnd names rs terms rows)

andHelp :: [VName] -> [Data] -> [Term] -> [Row] -> [Row]
andHelp names row1 terms [] = []
andHelp names row1 terms (row2:rs) =
        if canCom names row1 terms row2
                then
                        (combineD names row1 terms row2):(andHelp names row1 terms rs)
        else andHelp names row1 terms rs

addvr2Row :: ([VName], [Data]) -> ([VName], [Row]) -> ([VName], [Row])
addvr2Row (vs1, r) (vs2, rs) = (vs1, (r:rs))

canCom :: [VName] -> [Data] -> [Term] -> [Data] -> Bool
canCom [] [] _ _ = True
canCom (v:vs) (d1:ds) terms row =
        case vIsInTerm v terms row of
                Left _ -> canCom vs ds terms row
                Right d ->
                        if d == d1
                                then canCom vs ds terms row
                        else False


vIsInTerm :: VName -> [Term] -> [Data] -> Either String Data
vIsInTerm v [] [] = Left "Not in."
vIsInTerm v ((TVar vv):ts) (d:ds) =
        if v == vv
                then Right d
        else
                vIsInTerm v ts ds
vIsInTerm v ((TData d1):ts) (d:ds) = vIsInTerm v ts ds

combineD :: [VName] -> [Data] -> [Term] -> [Data] -> [Data]
combineD vlist dlist [] [] = dlist
combineD vlist dlist (t:ts) (d:ds) = case t of
                (TData _) -> combineD vlist dlist ts ds
                (TVar v) ->
                        if v `elem` vlist
                                then combineD vlist dlist ts ds
                        else
                                d:(combineD vlist dlist ts ds)

combineV :: [VName] -> [Term] -> [VName]
combineV vlist [] = vlist
combineV vlist (t:ts) = case t of
                (TData _) -> combineV vlist ts
                (TVar v) ->
                        if v `elem` vlist
                                then combineV vlist ts
```

```
                    else
                            v:(combineV vlist ts)

addTo :: (VName, Data) -> ([VName], [Data]) -> ([VName], [Data])
addTo (v, d) (vl, dl) = (v:vl, d:dl)


-- Set.elems (some Set of row) -> List of row
getTab :: [Term] -> [Row] -> [Row] -- the term of an atom (PSpec), and corresponding
    ETable
getTab terms [] = []
getTab terms (r:rs) =
        if moveIn terms r []
                then r:(getTab terms rs)
        else
                getTab terms rs


moveIn :: [Term] -> [Data] -> [(VName, Data)] -> Bool -- [term] and [data] should have the
    same length
moveIn [] [] _ = True
moveIn ((TData d):ts) (d1:rs) tmp =
        if d == d1
                then moveIn ts rs tmp
        else False
moveIn ((TVar v):ts) (d1:rs) tmp =
        case inTemp v tmp of
                Left _ -> moveIn ts rs ((v, d1):tmp)
                Right d ->
                        if d == d1
                                then moveIn ts rs tmp
                        else False

inTemp :: VName -> [(VName, Data)] -> Either String Data
inTemp v [] = Left "."
inTemp v ((name, d):tmps) =
        if v == name
                then Right d
        else inTemp v tmps
```

BlackBox.hs

```
-- This is a suggested skeleton for your main black-box tests. You are not
-- required to use Tasty, but be sure that your test suite can be build
-- and run against any implementation of the APQL APIs.

import Types
import Parser
import Preprocessor
import Engine
-- Do not import from the XXXImpl modules here!

import Test.Tasty
import Test.Tasty.HUnit
import qualified Data.Set as S
import qualified Data.Map as M

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests

tests :: TestTree
tests = myTest -- replace this

testCaseBad s t =
  testCase ("*" ++ s) $
    case t of
      Right a -> assertFailure $ "Unexpected success: " ++ show a
      Left (EUser _) -> return () -- any message is fine
      Left em -> assertFailure $ "Error: " ++ show em


myTest :: TestTree
myTest =
  testGroup "my tests"
  [
    testCase "parse Rule=Atom" $
      parseString "p(x)." @?= Right [Rule (Atom "p" [TVar "x"]) CTrue],
    testCase "parse Rule=Atom if Cond, Cond=Atom" $
      parseString "p(x) if q(x)." @?= Right [Rule (Atom "p" [TVar "x"]) (CAtom (Atom "q" [
          TVar "x"]))],
    testCase "parse Rule=Atom if Cond, Cond=is and isnot (), varname isnot, TData & TVar" $
      parseString "p(x) if v is \"u\" and (k is not isnot)." @?= Right [Rule (Atom "p" [TVar "x
          "]) (CAnd (CEq (TVar "v") (TData "u")) (CNot (CEq (TVar "k") (TVar "isnot"))))],
    testCase "parse Rule=Atom unless, Cond=or implies not false true" $
      parseString "p(x) unless not false implies q(x) or true ." @?= Right [Rule (Atom "p" [
          TVar "x"]) (CNot (CNot (COr (CNot (CNot CTrue)) (COr (CAtom (Atom "q" [TVar "x
          "])) CTrue))))],
    testCase "parse fail1" $
      case parseString "p(x)" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "parse fail2" $
      case parseString "p(x) if p" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "parse comment" $
      parseString "(**)p(x) if(*aaa*)true." @?= Right [Rule (Atom "p" [TVar "x"]) CTrue],
    testCase "clausify if false" $
      clausify [Rule (Atom "p" [TVar "x"]) (CNot CTrue)] @?= Right (IDB [("p",1)] []),
```

```
  testCase "clausify not(C and C)" $
    clausify [Rule (Atom "p" [TVar "x"]) (CAnd (CAtom (Atom "q" [TVar "x"])) (CNot (CAnd
        CTrue (CEq (TVar "x") (TData "true"))))))]
      @?= Right (IDB [("p",1)] [Clause (Atom "p" [TVar "x"]) [Atom "q" [TVar "x"]] [TNeq (
          TVar "x") (TData "true")]]),
  testCase "clausify not(C or C)" $
    clausify [Rule (Atom "p" [TVar "x"]) (CAnd (CAtom (Atom "q" [TVar "x"])) (CNot (COr
        CTrue (CEq (TVar "x") (TData "true"))))))]
      @?= Right (IDB [("p",1)] []),
  testCase "clausify not (not C)" $
    clausify [Rule (Atom "p" []) (CNot (CNot (CAtom (Atom "q" []))))] @?= Right (IDB [("p
        ",0)] [Clause (Atom "p" []) [Atom "q" []] []]),
  testCase "clausify C and false" $
    clausify [Rule (Atom "p" []) (CAnd (CAtom (Atom "q" [])) (CNot CTrue))] @?= Right (
        IDB [("p",0)] []),
  testCase "clausify C and (C or C)" $
    clausify [Rule (Atom "p" []) (CAnd (CAtom (Atom "q" [])) (COr (CAtom (Atom "r" []))
        CTrue))] @?= Right (IDB [("p",0)] [Clause (Atom "p" []) [Atom "q" [],Atom "r" []] [],
        Clause (Atom "p" []) [Atom "q" []] []]) ,
  testCase "clausify if false" $
    clausify [Rule (Atom "p" []) (CNot CTrue)] @?= Right (IDB [("p",0)] []),
  testCase "clausify or" $
    clausify [Rule (Atom "p" []) (COr (CAtom (Atom "q" [])) (CAtom (Atom "r" [])))] @?=
        Right (IDB [("p",0)] [Clause (Atom "p" []) [Atom "q" []] [],Clause (Atom "p" []) [Atom "r
        " []] []]) ,
  testCase "clausify fail : variable constrain" $
    case clausify [Rule (Atom "p" [TVar "x"]) CTrue] of
      Left e -> return ()
      Right p -> assertFailure $ "Unexpected clause: " ++ show p,
  testCase "stratify" $
    case stratify myIDB [("r",1)] of
      Right res ->
        if checkPSpec res myStrat
          then return ()
        else
          assertFailure $ "Stratify fail"
      Left e -> assertFailure $ "Stratify fail",
  testCase "stratify fail" $
    case stratify myIDB1 [("r",1)] of
      Left e -> return ()
      Right a -> assertFailure $ "Stratify should fail but did not",
  testCase "execute" $
    fmap M.fromList (execute myIDB myStrat [(("r",1), (S.fromList [["a"], ["b"], ["c"]]) )])
      @?= Right (M.fromList myEDB)
]
where
  myIDB = IDB [("p",2),("q",1),("s",1)]
          [Clause (Atom "p" [TVar "x",TVar "y"]) [Atom "q" [TVar "x"],Atom "r" [TVar "y"]] [],
           Clause (Atom "q" [TData "a"]) [] [],
           Clause (Atom "s" [TVar "x"]) [Atom "r" [TVar "x"]] [TNot (Atom "q" [TVar "x"])]]
  myStrat = [[("p",2) ,("q",1) ],[(" s",1) ]]
  myIDB1 = IDB [("p",1)]
               [Clause (Atom "p" [TVar "x"]) [Atom "q" [TVar "x"]] []]
  myEDB = [(("p",2),S.fromList [["a","a "],[" a","b "],[" a","c "]]) ,
           (("q",1) ,S.fromList [[" a "]]) ,
           (("s",1) ,S.fromList [[" b "],[" c "]]) ,
           (("r",1) ,S.fromList [[" a "],[" b "],[" c "]]) ]
```

```
checkPSpec :: [[PSpec]] -> [[PSpec]] -> Bool
checkPSpec l1 l2 =
  if length l1 == length l2
    then checkSta l1 l2
  else False

checkSta :: [[PSpec]] -> [[PSpec]] -> Bool
checkSta [] [] = True
checkSta (s1:ss1) (s2:ss2) =
  if S.fromList s1 == S.fromList s2
    then checkSta ss1 ss2
  else
    False


rudimentary :: TestTree
rudimentary =
 testGroup "Rudimentary tests"
   [testCase "parse1" $
      parseString pgmStr @?= Right pgmAST,
    testCaseBad "parse2" $
      parseString "p(x) if .",
    testCase "clausify1" $
      clausify pgmAST @?= Right pgmIDB,
    testCaseBad "clausify2" $
      clausify [Rule (Atom "p" [TVar "x"]) CTrue],
    testCase "stratify1" $ -- too strict! other correct answers also possible
      stratify pgmIDB [("r",1)] @?= Right pgmStratX,
    testCaseBad "stratify2" $
      stratify (IDB [("p",0)]
                    [Clause (Atom "p" []) [] [TNot (Atom "p" [])]]) [],
    testCase "execute" $
      fmap M.fromList (execute pgmIDB pgmStratX [(("r",1), pgmExtR)])
        @?= Right (M.fromList pgmEDB) ]
 where
   pgmStr = "p(x,y) if q(x) and r(y). q(\"a\").s(x) if r(x) and not q(x)."
   pgmAST = [Rule (Atom "p" [TVar "x", TVar "y"])
                  (CAnd (CAtom (Atom "q" [TVar "x"]))
                        (CAtom (Atom "r" [TVar "y"]))),
             Rule (Atom "q" [TData "a"])
                  CTrue]
   pgmIDB = IDB [("p", 2), ("q",1)]
                [Clause (Atom "p" [TVar "x", TVar "y"])
                        [Atom "q" [TVar "x"], Atom "r" [TVar "y"]]
                        [],
                 Clause (Atom "q" [TData "a"]) [] []]
   pgmStratX = [[("p",2), ("q",1) ]]
   pgmExtR = S.fromList [["b"], ["c "]]
   pgmExtQ = S.fromList [["a"]]
   pgmExtP = S.fromList [["a", "b"], ["a", "c "]]
   pgmEDB = [(("p",2),pgmExtP), (("q",1), pgmExtQ), (("r",1), pgmExtR)]
```

mailfilter.erl

```
-module(mailfilter).
-behaviour(gen_server).

% You are allowed to split your Erlang code in as many files as you
% find appropriate.
% However, you MUST have a module (this file) called mailfilter.

% Export at least the API:

-export(
  [ start/1
  , stop/1
  , default/4
  , add_mail/2
  , get_config/1
  , enough/1
  , add_filter/4
  ]).

% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

% You may have other exports as well
-export([test1/2, test2/2, test3/2]).

-type mail() :: any().
-type data() :: any().
-type label() :: any().
-type result() :: {done, data()} | inprogress .
-type labelled_result() :: {label(), result()}.
-type filter_result() :: {just, data()}
                       | {transformed, mail()}
                       | unchanged
                       | {both, mail(), data()}.
-type filter_fun() :: fun( (mail(), data()) -> filter_result() ).

-type filter() :: {simple, filter_fun()}
                | {chain, list(filter())}
                | {group, list(filter()), merge_fun()}
                | {timelimit, timeout(), filter()}.
-type merge_fun() :: fun( (list(filter_result() | inprogress)) -> filter_result() | continue ).

% API:

start(_Cap) ->
  gen_server:start({local, mailfilter}, ?MODULE, [], []).

stop(MS) ->
  Ret = gen_server:call(MS, get_state, infinity),
  gen_server:cast(MS, stop_all_filters),
  gen_server:stop(MS),
  Ret.

% stop_filter(PID) ->
%   exit(PID, kill).

add_mail(MS, Mail) ->
```

```
  Res = gen_server:call(MS, {add_mail, Mail}, infinity),
  {_, MR} = Res,
  {_, {_, L}} = get_config(MR),
   cast_filter_by_list (L, MR),
  Res.

get_config(MR) ->
  gen_server: call (?MODULE, {get_config, MR}, infinity).

default(MS, Label, Filt, Data) ->
  Flag = gen_server:call(?MODULE, {already_in, Label}, infinity),
  if Flag ->
    ok;
  true ->
    gen_server:cast(MS, {default, Label, Filt, Data})
  end.

enough(MR) ->
  gen_server:cast(?MODULE, {enough, MR}).

add_filter (MR, Label, Filt, Data) ->
  Flag = gen_server:call(?MODULE, {already_in, Label}, infinity),
  MSt = gen_server:call(?MODULE, {mail_in, MR}, infinity),
  if Flag ->
    {info, "Label has been registered."};
  true ->
    if MSt ->
      gen_server:cast(?MODULE, {add_filter, MR, Label, Filt, Data}),
      gen_server:cast(?MODULE, {cast_filters, Label, MR});
    true ->
      {info, "no such MR."}
    end
  end.

 cast_filter_by_list (_MR) -> ok;

 cast_filter_by_list ([{Label, _}|T], MR) ->
  gen_server:cast(?MODULE, {cast_filters, Label, MR}),
   cast_filter_by_list (T, MR).

 get_filters ([]) -> [];

 get_filters ([Key|T]) ->
    [{Key, inprogress}] ++ get_filters(T).

get_mailstate ([], _) -> [];

get_mailstate([Key|T], M) ->
    Item = maps:get(Key, M),
    [Item] ++ get_mailstate(T, M).

get_keys ([], _Filters) -> [];

get_keys([Key|T], Filters) ->
    {_, _, Default} = maps:get(Key, Filters),
    case Default of
      1 ->
        Ret = [Key];
```

```
      0 ->
        Ret = []
    end,
    Ret ++ get_keys(T, Filters).


%
    --------------------------------------------------------------------

% Function: init/1
% Description: Initiates the server
% Returns: {ok, State}          |
%          {ok, State, Timeout} |
%          ignore               |
%          {stop, Reason}
%
    --------------------------------------------------------------------

init ([]) -> {ok, {0, maps:new(), maps:new()}}.
% mail count(as MR), filters by label, mail state by MR

%
    --------------------------------------------------------------------

% Function: handle_call/3
% Description: Handling call messages
% Returns: {reply, Reply, State}          |
%          {reply, Reply, State, Timeout} |
%          {noreply, State}               |
%          {noreply, State, Timeout}      |
%          {stop, Reason, Reply, State}   | (terminate/2 is called)
%          {stop, Reason, State}            (terminate/2 is called)
%
    --------------------------------------------------------------------

handle_call(get_state, _From, State) ->
    {_, _, Mails} = State,
    Keys = maps:keys(Mails),
    MailState = get_mailstate(Keys, Mails),
    Reply = {ok, MailState},
    {reply, Reply, State};

handle_call( get_all_state , _From, State) ->
    Reply = {ok, State},
    {reply, Reply, State};

handle_call({already_in, Label}, _From, State) ->
    {_, Filters, _} = State,
    Tmp = maps:is_key(Label, Filters),
    Reply = Tmp,
    {reply, Reply, State};

handle_call({mail_in, MR}, _From, State) ->
    {_, _, Mails} = State,
    Tmp = maps:is_key(MR, Mails),
    if Tmp ->
      {reply, true, State};
    true -> {reply, false, State}
```

```
    end;

handle_call({add_mail, Mail}, _From, State) ->
    {Count, Filters, Mails} = State,
    MR = Count,
    All_Keys = maps:keys(Filters),
    Keys = get_keys(All_Keys, Filters),
    Labelled_Res = get_filters(Keys),
    State1 = {Count + 1, Filters, Mails#{MR => {Mail, Labelled_Res}}},
    Reply = {ok, MR},
    {reply, Reply, State1};

handle_call({get_config, MR}, _From, State) ->
    {_, _, Mails} = State,
    Tmp = maps:is_key(MR, Mails),
    if Tmp ->
      Reply = {ok, maps:get(MR, Mails)};
      true -> Reply = {error, "MR does not represent any mail in our system."}
    end,
    {reply, Reply, State}.

% handle_call({run_it, Filt, Mail, Data}, _From, State0) ->
%     case Filt of
%       {simple, Fun} ->
%         Res = Fun(Mail, Data);
%       {chain, Filt_List} ->
%         Res = run_chain(Filt_List, Mail, Data, unchanged);
%       {timeout, Time, Filt1} ->
%         {ok, PID} = gen_server:start({global, {Filt, Mail, Data}}, ?MODULE, [], []),
%         Res = gen_server:call(PID, {run_it, Filt, Mail, Data}, Time),
%          stop_filter(PID),
%         global:unregister_name({Filt, Mail, Data})
%     end,
%     {reply, Res, State0}.


%
    --------------------------------------------------------------------

% Function: handle_cast/2
% Description: Handling cast messages
% Returns: {noreply, State}          |
%          {noreply, State, Timeout} |
%          {stop, Reason, State}            (terminate/2 is called)
%
    --------------------------------------------------------------------

handle_cast({default, Label, Filt, Data}, State) ->
    {Count, Filters, Mails} = State,
    Tmp = maps:is_key(Label, Filters),
    if Tmp -> Filters1 = Filters;
    true -> Filters1 = Filters#{Label => {Filt, Data, 1}}
    end,
    State1 = {Count, Filters1, Mails},
    {noreply, State1};

handle_cast({enough, MR}, State) -> %stop the filters here
    {Count, Filters, Mails} = State,
```

```
    Tmp = maps:is_key(MR, Mails),
    if Tmp ->
      {_, List} = maps:get(MR, Mails),
       stop_filters_by_list (List, MR),
      Tmp = maps:is_key(MR, Mails),
      if Tmp ->
        Mails1 = maps:remove(MR, Mails);
      true ->
        Mails1 = Mails
      end,
      State1 = {Count, Filters, Mails1},
      {noreply, State1};
    true -> {noreply, State}
  end;

handle_cast({ add_filter , MR, Label, Filt, Data}, State) ->
    {Count, Filters , Mails} = State,
    Tmp = maps:is_key(Label, Filters),
    if Tmp ->
      State1 = State;
    true ->
      Filters1 = Filters#{Label => {Filt, Data, 0}},
      Tmp1 = maps:is_key(MR, Mails),
      if Tmp1 ->
        {Mail, Labelled_Res} = maps:get(MR, Mails),
        Config = {Mail, Labelled_Res++[{Label, inprogress}]},
        Mails1 = Mails#{MR := Config},
        State1 = {Count, Filters1, Mails1};
      true ->
        State1 = State
      end
    end,
    {noreply, State1};

handle_cast( stop_all_filters , State) ->
    {_, _, Mails} = State,
    MRs = maps:keys(Mails),
     stop_all_filter (MRs, Mails),
    {noreply, State};

handle_cast({ cast_filters , Label, MR}, State) -> % change PIDs
  {Count, Filters , Mails} = State,
  case global:whereis_name({MR, Label}) of
    undefined -> ok;
    Pid ->
      exit(Pid, kill )
      % gen_server:stop(Pid)
  end,
  {ok, PID} = gen_server:start({global, {MR, Label}}, ?MODULE, [], []),
  gen_server:cast(PID, {run_all, MR, Label, State}),

  State1 = {Count, Filters, Mails},
  {noreply, State1};

handle_cast({update, MR, Label, Mail, Res}, State) ->
    {Count, Filters , Mails} = State,
    {_, Labelled_Res} = maps:get(MR, Mails),
    case Res of
```

```
        {just, New_Data} ->
          New_Labbeled_Res = change_result(Label, New_Data, Labelled_Res),
          New_Config = {Mail, New_Labbeled_Res};
        {transformed, New_Mail} ->
          Labbeled_Res1 = change_result(Label, nothing, Labelled_Res),
          New_Labbeled_Res = set_undo(Label, Labbeled_Res1),
          New_Config = {New_Mail, New_Labbeled_Res},
          % restart all other filters
          L_R = lists:delete({Label, {done, nothing}}, New_Labbeled_Res),
           cast_filter_by_list (L_R, MR);
        unchanged ->
          New_Labbeled_Res = change_result(Label, nothing, Labelled_Res),
          New_Config = {Mail, New_Labbeled_Res};
        {both, New_Mail, New_Data} ->
          Labbeled_Res1 = change_result(Label, New_Data, Labelled_Res),
          New_Labbeled_Res = set_undo(Label, Labbeled_Res1),
          New_Config = {New_Mail, New_Labbeled_Res},
          % restart all other filters
          L_R = lists:delete({Label, {done, New_Data}}, New_Labbeled_Res),
           cast_filter_by_list (L_R, MR)
    end,
    New_Mails = Mails#{MR := New_Config},
    New_State = {Count, Filters, New_Mails},
    {noreply, New_State};

handle_cast({run_all, MR, Label, State}, State0) ->
    {_, Filters, Mails} = State,
    {Filt, Data, _} = maps:get(Label, Filters),
    {Mail, _} = maps:get(MR, Mails),
    % {ok, PID} = gen_server:start({global, {MR, Label, 0}}, ?MODULE, [], []),
    % Res = gen_server:call(PID, {run_it, Filt, Mail, Data}, infinity),
    % stop_filter (PID),
    % global:unregister_name({MR, Label, 0}),
    case Filt of
      {simple, Fun} ->
        Res = Fun(Mail, Data);
      {chain, Filt_List} ->
        Res = run_chain(Filt_List, Mail, Data, unchanged)
    end,
    gen_server:cast(?MODULE, {update, MR, Label, Mail, Res}),

    {stop, normal, State0}.

%
    --------------------------------------------------------------------

% Function: handle_info/2
% Description: Handling all non call/cast messages
% Returns: {noreply, State}          |
%          {noreply, State, Timeout} |
%          {stop, Reason, State}            (terminate/2 is called)
%
    --------------------------------------------------------------------

handle_info(_Info, State) ->
    {noreply, State}.

%
```

```
%--------------------------------------------------------------------

% Function: terminate/2
% Description: Shutdown the server
% Returns: any (ignored by gen_server)
%
%--------------------------------------------------------------------

terminate(_Reason, _State) ->
    ok.

%
%--------------------------------------------------------------------

% Func: code_change/3
% Purpose: Convert process state when code is changed
% Returns: {ok, NewState}
%
%--------------------------------------------------------------------

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

change_result(Label, Data, [{Label, _}|T]) ->
    [{Label, {done, Data}}] ++ T;

change_result(Label, Data, [H|T]) ->
    [H] ++ change_result(Label, Data, T).

set_undo(_Label, []) -> [];

set_undo(Label, [{Label1, Res}|T]) ->
    if
      Label =:= Label1 ->
        Res1 = [{Label1, Res}] ++ set_undo(Label, T);
      true ->
        Res1 = [{Label1, inprogress}] ++ set_undo(Label, T)
    end,
    Res1.


% all can be changed to call

 stop_filters_by_list (_, _MR) -> ok;

 stop_filters_by_list ([{Label, _}|T], MR) ->
    case global:whereis_name({MR, Label}) of
      undefined -> ok;
      Pid ->
        exit(Pid, kill),
        global:unregister_name({MR, Label})
        % gen_server:stop(Pid)
    end,
     stop_filters_by_list (T, MR).

 stop_all_filter (_, _Mails) ->
    ok;
```

```
 stop_all_filter ([MR|T], Mails) ->
    {_, List} = maps:get(MR, Mails),
     stop_filters_by_list (List, MR),
     stop_all_filter (T, Mails).

run_chain ([], _Mail, _Data, R0) -> R0;

run_chain([Filter|T], Mail, Data, R0) ->
    case Filter of
      {simple, Fun} ->
        R = Fun(Mail, Data);
      {chain, Filt_List} ->
        R = run_chain(Filt_List, Mail, Data, R0)
    end,
    case R of
      {just, New_Data} ->
        M1 = Mail,
        D1 = New_Data;
      {transformed, New_Mail} ->
        M1 = New_Mail,
        D1 = Data;
      unchanged ->
        M1 = Mail,
        D1 = Data;
      {both, New_Mail, New_Data} ->
        M1 = New_Mail,
        D1 = New_Data
    end,
    run_chain(T, M1, D1, R).

test1(_M, _D) -> {transformed, "bb"}.
test2(_M, _D) -> {just, 0}.
test3(M, D) -> test3(M, D).
```

test_mailfilter.erl

```
-module(test_mailfilter).

-include_lib("eqc/include/eqc.hrl").

-export([test_all/0, test_everything/0]).
-export([wellbehaved_filter_fun/0, filter/1, prop_mail_is_sacred/0, prop_consistency/0]). %
    Remember to export the other function from Q2.2


% You are allowed to split your test code in as many files as you
% think is appropriate, just remember that they should all start with
% 'test_'.
% But you MUST have a module (this file) called test_mailfilter.


test_all() ->
        eqc:module(test_mailfilter).

test_everything() ->
    test_all().

wellbehaved_filter_fun() ->
        I1 = int(),
        I2 = int(),
        I3 = int(),
        I4 = int(),
    oneof([fun(_Mail, _Data) -> {just, I1} end,
        fun(_Mail, _Data) -> {transformed, I2} end,
        fun(_Mail, _Data) -> unchanged end,
        fun(_Mail, _Data) -> {both, I3, I4} end]).

filter(FunGen) ->
        oneof([{chain, ?SUCHTHAT(L, list(FunGen()), length(L)>0)}, {simple, FunGen()}]).

prop_mail_is_sacred() ->
        {ok, M} = mailfilter:start(infinite),
        {ok, MR1} = mailfilter:add_mail(M, "a"),
        {ok, MR2} = mailfilter:add_mail(M, "b"),
        mailfilter:enough(MR1),
        {ok, MR3} = mailfilter:add_mail(M, "c"),
        mailfilter:enough(MR3),
        mailfilter:enough(MR3),
        {ok, Res} = mailfilter:stop(M),
        length(Res) == 1.

prop_consistency() ->
        {ok, M} = mailfilter:start(infinite),
        mailfilter:default(M, t0, {simple, fun(_A, _B)->{transformed, 0} end}, []),
        {ok, MR} = mailfilter:add_mail(M, "a"),
        mailfilter:add_filter(MR, t1, {simple, fun(_A, _B)->{transformed, 1} end}, []),
        timer:sleep(10),
        {ok, Res} = mailfilter:stop(M),
        if Res == [{0,[{t0,{done,nothing}},{t1,inprogress}]}] -> true;
                true ->
                        if Res == [{1,[{t0,inprogress},{t1,{done,nothing}}]}] -> true;
                                true -> false
                        end
```

```
    end.


prop_filter () ->
    ?FORALL(Filter, filter(fun wellbehaved_filter_fun/0),
        begin
            {ok, M} = mailfilter:start( infinite ),
            mailfilter :default(M, t0, Filter , []) ,
            {ok, MR} = mailfilter:add_mail(M, "aa"),
            mailfilter : add_filter (MR, t1, Filter , []) ,
            timer:sleep(10),
            {ok, Res} = mailfilter :stop(M),
            collect (Res, true)
        end
    ).
```

test_unit.erl

```
-module(test_unit).
-export([test_all/0, test_everything/0]).
-include_lib("eunit/include/eunit.hrl").

test_all() -> eunit:test(testsuite(), [verbose]).

test_everything() ->
    test_all().

testsuite() ->
        [{"unit test", spawn, [test_start(), test_default1(), test_default2(),
                test_add_mail(), test_enough(), test_get_config(), test_add_filter(),
                 mytest_registered_filter(), mytest_add_filter_info(), mytest_add_filter_result()
                    ]}].

test_start() ->
        {
                "Start and Stop.",
                fun() ->
                        {A, B} = mailfilter:start(infinite),
                        ?assertMatch({ok, _}, {A, B}),
                        ?assertEqual({ok, []}, mailfilter:stop(B))
                end
        }.

test_default1() ->
        {
                "Default. Simple filters .",
                fun() ->
                        {ok, M} = mailfilter:start(infinite),
                        ?assertEqual(ok, mailfilter:default(M, t0, {simple, fun(_A, _B)->{just,
                            1} end}, [])),
                        ?assertEqual(ok, mailfilter:default(M, t1, {simple, fun(_A, _B)->{
                            transformed, 2} end}, [])),
                        ?assertEqual(ok, mailfilter:default(M, t2, {simple, fun(_A, _B)->
                            unchanged end}, [])),
                        ?assertEqual(ok, mailfilter:default(M, t3, {simple, fun(_A, _B)->{both,
                             3, 4} end}, [])),
                         mailfilter:stop(M)
                end
        }.

test_default2() ->
        {
                "Default. Chain filters .",
                fun() ->
                        {ok, M} = mailfilter:start(infinite),
                        ?assertEqual(ok, mailfilter:default(M, t0, {chain, [{simple, fun(_A, _B)
                            ->{just, 1} end}, {simple, fun(_A, _B)->{transformed, 2} end}]}, [])
                            ),
                        ?assertEqual(ok, mailfilter:default(M, t1, {chain, [{simple, fun(_A, _B)
                            ->{both, 3, 4} end}]}, [])),
                         mailfilter:stop(M)
                end
        }.

test_add_mail()->
```

```
        {
                "Add mail. With or without default. When stop, check the length of mail state
                    list .",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                        ?assertMatch({ok, _}, mailfilter :add_mail(M, "a")),
                        mailfilter :default(M, t0, {simple, fun(_A, _B)->{just, 1} end}, []),
                        ?assertMatch({ok, _}, mailfilter :add_mail(M, "b")),
                        {ok, S} = mailfilter :stop(M),
                        ?assertEqual(2, length(S))
                end
        }.

test_enough() ->
        {
                "Enough. Enough with MR in or not in. When stop, check the length of mail state
                    list .",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                        ?assertEqual(ok, mailfilter :enough(1)),
                        mailfilter :add_mail(M, "a"),
                        mailfilter :add_mail(M, "a"),
                        ?assertEqual(ok, mailfilter :enough(0)),
                        {ok, S} = mailfilter :stop(M),
                        ?assertEqual(1, length(S))
                end
        }.

test_get_config () ->
        {
                "Get config. When MR in or not in.",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                        ?assertMatch({error, _}, mailfilter :get_config(1)),
                        mailfilter :add_mail(M, "a"),
                        ?assertMatch({ok, _}, mailfilter :get_config(0)),
                        mailfilter :stop(M)
                end
        }.

test_add_filter () ->
        {
                "Add Filter. Simple and Chain. MR in and not in.",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                        mailfilter :add_filter (0, t0, {simple, fun(_A, _B)->{just, 1} end}, []),
                        mailfilter :add_mail(M, "a"),
                        mailfilter :add_filter (0, t1, {simple, fun(_A, _B)->{just, 1} end}, []),
                        mailfilter :add_filter (0, t2, {chain, [{simple, fun(_A, _B)->{both, 3,
                            4} end}]}, []),
                        mailfilter :stop(M)
                end
        }.

mytest_registered_filter () ->
        {
                "Filters with one label would only register once, the latter ones would be
                    discard.",
```

```
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                         mailfilter :default(M, t0, {simple, fun(_A, _B)->{just, 1} end}, []),
                         mailfilter :default(M, t0, {simple, fun(_A, _B)->{transformed, 2} end},
                             []),
                         mailfilter :add_mail(M, "a"),
                         mailfilter : add_filter (0, t0, {simple, fun(_A, _B)->{just, 1} end}, []),
                        {ok, {_C, F, _D}} = gen_server:call(M, get_all_state ),
                        ?assertEqual(1, maps:size(F)),
                         mailfilter :stop(M)
                end
        }.

mytest_add_filter_info () ->
        {
                "If MR not in or Label has been registered, return info .",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                        ?assertMatch({info, _}, mailfilter : add_filter (0, t0, {simple, fun(_A, _B)
                            ->{just, 1} end}, [])),
                         mailfilter :add_mail(M, "a"),
                        ?assertEqual(ok, mailfilter : add_filter (0, t1, {simple, fun(_A, _B)->{
                            just, 1} end}, [])),
                        ?assertMatch({info, _}, mailfilter : add_filter (0, t1, {simple, fun(_A, _B)
                            ->{just, 1} end}, [])),
                         mailfilter :stop(M)
                end
        }.

mytest_add_filter_result () ->
        {
                "Check the result of the default and add filter functions .",
                fun () ->
                        {ok, M} = mailfilter:start( infinite ),
                         mailfilter : add_filter (0, t2, {simple, fun(_A, _B)->{just, 1} end}, []),
                         mailfilter :default(M, t1, {simple, fun(_A, _B)->{just, 1} end}, []),
                         mailfilter :add_mail(M, "a"),
                         mailfilter : add_filter (0, t0, {simple, fun(_A, _B)->{both, 3, 4} end}, [])
                             ,
                        {ok, {_C, F, _D}} = gen_server:call(M, get_all_state ),
                        ?assertMatch([t0, t1], maps:keys(F)),
                        timer:sleep(10),
                        {ok, S} = mailfilter :stop(M),
                        ?assertEqual([{3,[{ t1,{done,1}},{t0,{done,4}}]}], S)
                end
        }.
```