AP exam 2020

Exam number 4

2020-11-06

0 APQL: Parser

In this section, I present and explain key design choices, and features of my implementation of the APQL parser; choice of parser library, disambiguation of grammar, handling of whitespace, and limitations.

The handed-in code in `code/apql/src` (or appendix A) is supplied with code comments where appropriate, and is largely self-documenting.

0.1 Parser: Choice of parser library; consequences thereof

I have chosen `ReadP` for my implementation, and did so for the simple reason that this is what I familiarized myself with in the second assignment of the course.

As of yet I have not looked much into the `Parsec` library, and I am confident that this has its disadvantages; in particular, the `ReadP` library does not support very helpful error handling. In the future, it might be advantageous for me to port my parser to `Parsec`.

However, the `ReadP` is far simpler to work with, and is more than sufficient for parsing context-free grammars (the task at hand), so I am satisfied with my choice.

0.2 Parser: Disambiguating the grammar

The Boa grammar, as presented in the assignment text, is somewhat ambiguous with respect to parsing of the `Cond` non-terminal. The assignment text defines associativity and precedence of both unary and binary `Cond` operators; before implementation into `ReadP`, I need to eliminate left-recursion and ambiguity in the grammar.

To do this, I left-factor and re-structure the grammar into a parsing tree with recursive descent. Disambiguation is only necessary for the *Cond* non-terminal, so I only apply modifications here. I won't clutter the report with a table of my revised grammar, but rather refer to the next section, where I illustrate how the parser tree is implemented.

0.2.1 Implementation of parse tree for disambiguated grammar

Below snippet shows how parsing of the disambiguated *Cond* is implemented using `ReadP`:

```
1  parseCond :: ReadP Cond
2  parseCond = parseCondBinOps
3
4  parseCondBinOps :: ReadP Cond
5  parseCondBinOps = infix1
6    where infix1  = chainr1 infix2   \$ keyword "implies" \$> CNot .: COr
7          infix2  = chainl1 infix3   \$ keyword "or"      \$> COr
8          infix3  = chainl1 parseNot \$ keyword "and"     \$> CAnd
9
10 parseNot :: ReadP Cond
11 parseNot = (keyword "not" >> CNot <\$> parseNot) <|> parseBottom
12
13 -- Not actually the bottom of the parse tree, but rather the
14 -- bottommost internal node in the conceptual parse tree.
15 parseBottom :: ReadP Cond
16 parseBottom = CAtom <\$> parseAtom
17           <|> parseBoolConsts
18           <|> parseTermBinOps
19           <|> between (char' '(') (char' ')') parseCond
```

`parseCond` represents the top of the `Cond` parser (sub-)tree. Associative binary operators are easily handled with the `ReadP` built-ins `chainr1` and `chainl1` for right- and left associative binops, respectively.

Precedence of binops is handled by placing tighter binding operators lower in the parse tree, such that the parser is forced to attempt parsing of these first (lines 5-8).

Logical negation binds tighter than the binary ops, and is thus placed below these in the parse tree. This parser is recursive since logical negation can nest (right-associatively) (line 11).

Now we get to the ”bottom” of the parsing tree in line 14; from here, bool constants, atoms, ”is/is not” expressons, and parenthesized conditions are parsed.

0.3 APQL string constants

The APQL grammar specifies a string constant to be zero or more printable ASCII characters surrounded by double quotes; an APQL string may *contain* literal double quotes, if those quotes are denoted in the source program with two consecutive double quotes.

Below is a snippet of my string constant parser:

```
parseData :: ReadP Data
parseData = lexeme \$ between (char '"') (char '"') strConst
  where strConst = many (string "\"\"" \$> '"' <|> -- replace consecutive quotes with one;
                   (satisfy isStrContent))       -- parse any printable char but double quotes.
        isStrContent c = isPrint c && isAscii c && c /= '"'
```

To handle the specification, `parseData` parses zero or more occurences of two consecutive double quotes (line 3) *or* any printable ASCII byte that is not a double quote (line 4), since a single double quote would terminate the string.

0.4 Handling whitespace in parsing

0.4.1 Whitespace: code comments and tokenization

Below is my implementation of an auxiliary parser which skips an arbitrary number of code comments with an arbitrary amount of leading and trailing whitespace, and the helper function I use for tokenizing parsers.

```
1  -- Comment parser.
2  -- TODO: probably parses too much whitespace, but this is
3  --       only a matter of efficiency and thus not a priority.
4  skipComments :: ReadP ()
5  skipComments = many (skipSpaces >> string "(*" >>
6                  manyTill (satisfy (const True)) (string "*)")) >> skipSpaces
7
8  ...
9
10 -- used to tokenize parsers. skips leading comments and/or whitespace.
11 lexeme :: ReadP a -> ReadP a
12 lexeme = (skipComment >>)
```

As is apparent from the snippet, I parse whitespace *before* a given parser; this is against Filinski's advice of skipping whitespace *after* parsers; it is, however, in line with the original advice of my TA [1], and has thus what has been my preferred method since the second weekly assignment.

In addition `skipComment` could probably do with some optimization; in particular, it is probably not very efficient (or well-thought-out) usage of `skipSpaces` to simply skip spaces before and after comments (lines 5-6) but it is effective nonetheless.

0.4.2 Whitespace: keywords and names

The APQL grammar requires some whitespace between keywords/names and adjacent letters/digits, since this character could possibly be part of a different keyword/name.

My handling of this specification is to simply eat up as much as possible that fits the description of a keyword, perform a single character look-ahead and assert that this is a legal follow-up to a keyword/identifier. I do so with the following helper functions:

[1] They have since redacted that advice and should not be held accountable ;)

```
keyword :: String -> ReadP String
keyword = (<* (look >>= guard . canFollowKeyword)) . string'
```

where `canFollowKeyword` is false for strings beginning with digits or letters, and true for everything else (including the empty string), and `string'` is `lexeme . string`. This is also used in my name parser.

0.5 APQL parser: Known limitations

At this point, ìe. *before* validation testing, my implementation has no known limitations wrt. the specifications. Emphasis on *known*; I do not claim that my program does not have any hidden limitations or bugs, but we shall remain agnostic about these at least until validation testing.

1 APQL: Preprocessor

In this section, I present my implementation of the APQL preprocessor. I discuss first the implementation of code transformations specified in the assignment, then I present my implementations of `clausify` and `stratify`.

The handed-in code in `code/apql/src/PreprocessorImpl.hs` (or appendix A) is supplied with code comments where appropriate, and is largely self-documenting.

1.1 Preprocessor: Code transformations

The assignment text describes three sets of APQL code transformation equivalences, which I will not discuss here. For brevity, I shall refer to the three transformations as ”NOT flipping”, ”AND distribution”, and ”OR splitting”.

The implementations of these code transformation is largely trivial, as it is simply a number of function clauses making AST substitutions, but here is the function tying it all together:

```
1 transform :: Program -> Program
2 transform prog = if prog' == prog then prog' else transform prog'
3   where prog' = splitORRules $ distribAnds $ flipNots prog
```

The assignment text specifies to perform the code transformations in the order given in line 3 of the above snippet; however, since applying one code transformation very often uncovers new opportunities for (re-)applying another, I recursively transform the program until the transformations converge.

Convergence is always guaranteed since no right-hand side of a transformation matches the pattern of any left-hand side of any other transformation (or of itself).

1.2 Preprocessor: Clausification

After code transformation, the first step of clausification is an entirely straight-forward mapping of transformed rules to clauses. What is more interesting is the verification of clauses that follows; for a clause to verify, it must satisfy the condition that for any variable occuring in the clause head atom or in any of the `is/is not` tests in the body, that variable must also be occur in a non-negated atom reference in the clause body.

Below snippet shows how I verify clauses:

```
1  verifyClause :: Clause -> Either ErrMsg Clause
2  verifyClause clause@(Clause (Atom _ head_args) atoms ts) =
3    if need `isSubsetOf` have
4    then return clause
5    else Left $ EUser $ "Cannot verify clause. Missing variables: "
6                         ++ show (need \\ have)
7    where vars terms = [var | var@TVar{} <- concat terms]
8          have = vars $ map args atoms
9          need = vars $ head_args : [[a, b] | TNeq a b <- ts]
10                                ++ [[a, b] | TEq  a b <- ts]
```

The function computes "have" and "need" sets, comprising variables occuring in the head atom and tests in the clause body, and variables occuring in non-negated atoms in the clause body, respectively, and simply computes whether "need" is a subset of "have" (line 3). If it is not, an appropriate `EUser` error is returned to signal a faulty input program (lines 5-6).

In line 7, a list comprehension is used to easily extract `TVars` from a list of `Terms`; similarly, in lines 9-10, comprehensions are used to extract `is/is not` tests from the lists of tests (since this also includes atom negations.

1.2.1 Clausification: Choice of monads

For my implementation of `clausify` and all its helper functions, I use only the `Either ErrMsg` monad to signify errors in clause verification. I did not readily see any opportunity to make the program more efficient by using any of the reader/writer/state monads we have worked with throughout the course, so I chose not to insert them unnecessarily.

I might even say that using `Either ErrMsg` is a little overkill, since there is only one possible error in clausification: failure to verify clause.

This is because any syntactically correct program can always be transformed (by the transformation equivalences mentioned earlier) to a set of rules each exhibiting a form that can always be clausified (I do not prove this but refer to the assignment text where it is stated). By ensuring a fully transformed program (by iteration until convergence), any other errors barring clause verification will have been caught earlier.

1.3 Preprocessor: Stratification

After clausification, the stratification pass is called. The stratifier takes an IDB and a set of extensional predicates as input; the assignment text states that it is valid to make the assumption that the IDB given has been previously computed by a call to `clausify` and that it represents a properly clausified program. I make this assumption.

I do not, however, make any assumptions on the contents of the input IDB and extensional predicates; the specifications state that there must be no overlap in in- and extensional predicates. This is easily verified by asserting an empty intersection between the two (line 3 of the snippet in the next subsubsection).

1.3.1 Stratification: `SimpleClause`

When dealing with clauses during stratification, we are only concerned with the predicate in the clause head, as well as the positive and negative references to other predicates (other atoms, more precisely) made in the clause body.

Thus, in order to greatly simplify coding `stratify`, I introduce the type synonym `SimpleClause`, as seen below:

```
type SimpleClause = (PSpec, ([PSpec], [PSpec]))
```

The new type is a tuple whose first element is the `PSpec` for the predicate head of the clause, and whose second element is a tuple of lists of `PSpecs` for the positive and negative references to other atoms made by this clause, respectively.

Each `Clause` in the IDB passed to `stratify` is first mapped to a `SimpleClause` before invoking the main functionality of the stratifier. Lines 8-10 of the below snippet show how this is done:

```
1  stratify :: IDB -> [PSpec] -> Either ErrMsg [[PSpec]]
2  stratify (IDB ips clauses) eps =
3    if ips `disjoint` eps then
4       makeStrata (map simpleClause clauses) ips []
5    else Left $ EUser $ "Cannot stratify: Overlap in ex- and intensionals."
6
7    where
8      simpleClause (Clause atom posRefs tests) =
9        (pSpec atom, (map pSpec posRefs, negRefs tests))
10     negRefs tests = [pSpec a | TNot a <- tests]
```

1.3.2 Stratification: Building strata

The reason for clausifying before stratification is the ability to build strata bottom-up in iterative fashion. I implement this iterative strata building in a function `makeStrata`, which, starting with all intensional predicates unplaced, builds strata one by one until there are no remaining unplaced predicates. Below snippet shows my implementation of a different function, `makeStratum`, which is used by `makeStrata` to build a single stratum:

```
1  makeStratum :: [SimpleClause] -> [PSpec] -> [PSpec]
2             -> Either ErrMsg [PSpec]
3  makeStratum clauses stratum removed =
4    if stratum' == [] then Left $ EUser "Cannot stratify: empty stratum."
5    else if stratum' == stratum then return stratum'
6    else makeStratum clauses' stratum' removed'
7
8    where
9      stratum' = filter keep stratum
10     clauses' = filter ((`elem` stratum') . fst) clauses
11     removed' = removed ++ (stratum \\ stratum')
12
13     -- keep p if it has no neg references to atoms in its stratum,
14     -- and no pos references to atoms removed in this iteration.
15     keep p = neg `disjoint` stratum && pos `disjoint` removed
16       where (pos, neg) = foldl combine ([], [])
17                            [snd c | c <- clauses, fst c == p]
18 $
```

Given a list of `SimpleClause`, a stratum (initially all predicates still unplaced at this point) and a list of predicates removed from this stratum (initially empty) predicates, `makeStratum` computes a new stratum `stratum'` (line 9).

`makeStratum` computes this new `stratum'` by removing all predicates from the current stratum which either:

- *directly refers negatively* to an atom in this current stratum (including itself), or
- *indirectly refers negatively* to an atom in this current stratum (by transitivity of the ”refers” relation)

To check the former means simply asserting an empty intersection of this predicate's negative references and the predicates of the stratum. The latter is a little more subtle; to check indirect references, I need to keep track of predicates previously removed from this stratum, but then this also becomes a simple check for a null intersection between positive references and removed predicates.

All this is implemented in the `keep` predicate in line 15 of the above snippet (line 16-17 performs the actual extraction of positive and negative references).

Alternatively, and perhaps more ideal, one could compute the transitive closure of the ”refers negatively” relation immediately and remove all predicates in one go without calling recursively.

If, the newly computed `stratum'` is empty as a result of removing all predicates, this means that the input program must have contained mutually recursive atom negations, and the program is then not stratifiable. This is signaled with an `EUser` error, since this is a fault in the input program.

1.3.3 Stratification: Choice of monads

Again, I simply use the `Either ErrMsg` monad in implementing my stratification. Here, there are two possible errors: overlapping in- and extensional predicates, and truly unstratifiable programs as consequence of mutually recursive atom negations (the former requires filtering the input EDB, whilst the latter requires rewriting the source program), so here, I would argue that it makes somewhat more sense to use `Either ErrMsg`.

Looking back, I would have liked to have used the `State` monad to keep track of removed predicates. I initially decided not to, since my first thought was to remove all relevant predicates within one call to `makeStratum`, as explained above.

A `Writer` monad could also have been used to store finished strata, and I regret not pursuing this from the start.

1.4 Preprocessor: Known limitations and immediate assessment

Again, I do not know of (or suspect) any limitations in my preprocessor, but I shall not make any claims until validation testing.

This is not a technical limitation, but my implementation `stratify` is not very elegant in the code; it is cluttered by the need to pass around lists of currently computed strata and currently removed predicates around. In the future, I would like to rewrite my program to implement the changes discussed in the subsubsection 1.3.3.

---

2 APQL: Engine

I fail to finish a working implementation of the execution engine. However, I feel that I have made a respectable go at a solution, and so in the following section, I will *briefly* present my thoughts.

The handed-in code in `code/apql/src/EngineImpl.hs` (or appendix A) is supplied with code comments where appropriate, and is largely self-documenting.

2.1 Engine: Devising the algorithm

The assignment text for the execution engine is extremely ambiguous and required many reads (for me, anyway). After many iterations, I devise the following algorithm from the sparse specifications given in the report:

```
evalStrata strata:
  for each stratum in strata
  | do until stable:
  | | for each pred in stratum:
  | | |
  | | | contribs = empty EDB
  | | | map clauses for this pred:
  | | | |
  | | | | map possible instantiations of atom head:
  | | | | | apply atom instantiation to clause (including head).
  | | | | |
  | | | | | verify instantiated clause. asterisk denotes possible failure, in
  | | | | |   which case the next possible instantiation should be tried:
  | | | | | for each atom2 in atom's posrefs:
  | | | | | | * apply atom instantiation to atom2. is atom2' known?
  | | | | | for each test in this atom's tests:
  | | | | | | * instantiate negative references; check non-membership
  | | | | | |   in edb and current contributions.
  | | | | | | * instantiate EQ/NEQ tests and test these.
  | | | | | |
  | | | | |
  | | | | | made it this far? good! this instantiation satisfies clause.
  | | | | | contrib = Just head. return contrib
  | | | |
  | | | | now have a [Maybe contrib] of each clause's contrib (if any).
  | | | | unpack, removing Nothings (or just use mapMaybe),
  | | | | and turn into an EDB which can be merged outside.
  | | | |
  | | | now have list of all contribs for this atom and its clauses.
  | | | return contribs `edbMerge` (clause contributions).
  | |
  | | stable? good! merge contribs with current EDB.
  |
  | return finished EDB.
```

2.2 Engine: Implementation

Since I did not achieve a working implementation, this subsection is perhaps not very interesting. Nevertheless, I would like to share some of my ideas.

2.2.1 Implementation: data types used

I introduce a number of type synonyms to ease implementation:

- `type Stratum = [(PSpec, [Clause])]`: a stratum is a list of pairs of predicates and their associated clauses.
- `type Strata = [Stratum]`: a strata is, well, a list of strata.
- `type EDBM = M.Map PSPec ETable`: I convert the external `EDB` to a map of `PSpecs` to ETables, rather than a list of pairs of `PSpecs` and `ETables`. This will make lookup, insertion, and union in/of extensional databases a breeze.
- `type Instance = [(Term, Term)]`: an Instance is a list of pairs of `TVars` to `TDatas`, and represent the environment of an atom instantiation. These are created in atom instantiations, and are used to more easily access the environment during clause application.

2.2.2 Implementation: creating atom instantiations

Below snippet shows how I create instances based on extensional data and a given atom:

```haskell
-- An atom is attempted instantiated with a row of data by
-- a pairwise match-up of atom arguments to row entries.
makeInstance :: Atom -> Row -> Maybe Instance
makeInstance (Atom name args) row = zipWithM matchup args row
  where matchup (TVar _)   d  = Just (TVar name, TData d)
        matchup (TData d1) d2 =
          if d1 == d2 then Just (TVar name, TData d2)
                      else Nothing
```

The `matchup` function is the meat of `makeInstance`. Given a `Term` and a `Data`, `matchup` creates a `Maybe (Term, Term)`, mapping a variable to the instantiated value, or `Nothing`, if this is not possible.

2.2.3 Implementation: applying clauses

During execution of strata, the execution engine needs to *apply clauses*, which comprises trying every different instantiation of the clause head atom and verifying that references and tests made in the clause body are satisfied under this particular instantiation. If these are satisfied, then the instantiated clause head should be added to the contributions of this clause.

Below snippet shows how I attempt to implement this:

```haskell
contribution :: PSpec -> [Clause] -> EDBM -> Maybe EDBM
contribution _ [] edb = Just edb
contribution p (c@(Clause head _ _):cs) edb = do
  contribution p cs $ edbUnions $ mapMaybe (applyClause edb c) insts

  where insts  = mapMaybe (makeInstance head) extens
        extens = edb `extensFor` p

applyClause :: EDBM -> Clause -> Instance -> Maybe EDBM
applyClause edb (Clause head posrefs tests) inst =
  mapM (matchAtom edb inst) posrefs >> -- match and verify pos references.
    mapM (checkTest edb inst) tests $> -- match and verify tests.
      edbSingletonFromAtom head        -- add contribution to an EDBM.
```

I am almost certain that this is the source of bugs in my implementation, or at least the biggest source thereof.

My idea was to use the `Maybe` monad to signify failure to verify clauses under certain clause head instantiations, but from manual testing I can discern that when a clause fails under *one* instantiation, then subsequent instantiations are not even tried and results from prior verifications are discarded altogether.

3 APQL Testing

In this section, I discuss my APQL validation test plan and report the results of testing. I only plan to test those of the implemented modules which I have claimed correct, and will thus not be testing my failed implementation of the execution engine.

3.1 APQL Testing: Reproduction of tests

I use `Tasty` for testing. All of my tests can be viewed in `code/apql/tests/suite1` or appendix A.5.1 to this report.

To reproduce tests, navigate to `code/apql` and run `stack test`.

3.2 APQL Testing: Testing goals

The goal of testing is to devise a test plan to attain full edge case coverage of the parser and preprocessor with unit testing of each implemented API function (and to some degree, helper functions), which can uncover as many bugs as possible.

In doing so, I also want to cover each possible type of failure in either of the modules and to assert correct handling of these.

Wrt. negative testing of the parser: recall that the `ReadP` library does not support overly telling error messages; this restricts negative testing to a simple "success/fail" distinction.

3.3 APQL Testing: Test plan

3.3.1 Test plan: strategy

At the time of the third weekly assignment, implementing the BoaParser, Filinski at one point suggested on the absalon discussion forum that the BoaParser could be satisfiably tested using the top-level `parseString` interface only, since all expressions were programs in their own right, and as such it should be possible to essentially unit test individual functionalities using a black-box testing strategy.

I do not see why this should not also be true for the APQL modules, and so this is the strategy I choose.

In addition, after first asserting correctness of my parser, I will use it to generate test input for the clausifier of the preprocessor, and, in turn, after asserting correct clausification, I will use this to generate test input for the stratifier.

Positive testing

For the parser module, for each constructor, I want to assert correct parsing of the language feature, including correct association and precedence. In addition, I want to test a number of parsing identities, eg. that `p() unless q()` parses the same as `p() if not q()`.

For the preprocessor module, I want to test correct transformation of a syntacticaly correct program; correct clausification of a properly transformed program; and stratification of a properly clausified program. Here, there are not very many different equivalence classes to test, but ideally, I want to hit all of them.

Negative testing

For the parser module, for each constructor, I want to test various types of parsing failures (eg. association or whitespacing errors).

For the preprocessor module, I of course want to test correct detection and reporting of each type of faulty input program. Given a syntactically correct program, code transformation cannot fail, so this I will not negative test. Given a properly transformed program, the clausifier can only fail in one way, and given a properly clausified program, the stratifier can only fail in two ways, so this should only amount to three equivalence classes of test cases.

3.3.2 Test plan: Test suite

For lack of time, I won't go into detail with my test suite, but each test has a fitting name explaining just what that test asserts, so I will advice the reader to view the test cases in my test plan by running `stack test`, or, alternatively, in the source code in `code/apql/tests/suite1`.

3.4 APQL Testing: validation testing results

All of my validation tests pass successfully.

3.5 APQL Testing: Evaluation

My final test suite includes 75 tests, and I manage to cover mostly every unit test case and equivalence class test I had planned for.

Based on validation test results alone, I am almost convinced that my implementation is sound; I would have liked to have rigorously sought out possibly uncovered edge cases, but once again, I succumb to the deadline.

More importantly, I have not performed any *actual* integration tests, but only simple tests linking the modules with simple inputs, so I cannot say for certain that my implementation would not break on general inputs (eg. large, contrived programs).

Aside from validation test results, I am generally satisfied with my implementation and what it has taught me of Datalog (on which APQL is based).

---

4 Mailfilter

In this section, I present my implementation of the `mailfilter` server. I first discuss my implementation of the API, then the mailfilter/coordinator/worker relationship (more on this later).

The handed-in code in `code/mailfilter/src/` (or appendix A) is supplied with code comments where appropriate, and is *somewhat* self-documenting.

4.1 Mailfilter: Design

Clients should be able to add mails and filters to the `mailfilter` server, and for a given registered mail, multiple filters should be able to run concurrently.

I design my program such that for each `Mail` registered at the `mailfilter` server, there exists a `coordinator` process handling all filters for `Mail`.

Since the `mailfilter` server only needs to communicate with the client and send asynchronous requests to `coordinators`, I choose to implement it as a `gen_server`.

4.1.1 Design: `coordinator`

To obtain concurrency, the `coordinator` is responsible for *coordinating* the computation of filters for its `Mail`. It does this by spawning a new `worker` process for each new filter attached, and then to fetch results and validate/invalidate filter results, as well as to prompt `workers` to re-computate using an updated `Mail2`, if necessary.

Once all attached filters have produced a valid result, the `coordinator` sends the updated config for `Mail` back to the `mailfilter` in an asynchronous cast.

Since the `coordinator` only needs to send and receive asynchronous requests to and from the `mailfilter` server and its own `workers`, I choose to also implement this as a `gen_server`.

I should be able to implement the `coordinator` as an entirely asynchronous server using only casts to communicate with the `mailfilter` and its own `workers`. Then it is up to the underlying protocols to ensure that messages arrive, and for the `mailfilter` or `worker` to correctly handle these casts.

4.1.2 Design: `worker`

The `worker` should simply be a continuous server with the ability to evaluate a filter, and to remain idle until it is later prompted to shutdown or re-evaluate its filter after becoming invalidated by the `coordinator`.

I should also be able to implement the `worker` as an asynchronous server using only casts. It might make sense to implement the `worker` as a `gen_statem`, since you can argue that it has two states, namely computing and being idle, but since the `worker` neither can nor should accept calls/casts during computation, there is really only one state (or no state), and so I can also simply use a `gen_server` for the `worker`.

4.2 Mailfilter: API

The API specifies both synchronous and asynchronous requests to the mailserver. The return types for synchronous API functions are `{ok, Value} | {error, Reason}`, whilst the asynchronous requests are undefined.

Below is a snippet illustrating my handling of synchronous API requests (and possible errors thereof), and my handling of asynchronous API requests, respectively:

```
1  -spec get_config(ext_mail_ref()) -> {ok, config()} | my_error().
2  get_config({MailServer, MailTag}) ->
3    case gen_server:call(MailServer, {get_config, MailTag}) of
4      {ok, Config}    -> {ok, Config};
5      {error, Reason} -> {error, Reason};
6      _               -> {error, internal_err}
7    end.
8
9  ...
10
11 -spec default(pid(), label(), filter(), data()) -> any().
12 default(MailServer, Label, Filter, InitData) ->
13   gen_server:cast(MailServer, {add_default, Label, Filter, InitData}).
```

4.2.1 API: Mail references

Whereas synchronous API functions take the PID of the mail server as argument, asynchronous API functions take an unspecified *mail reference*, and it is then up to me to design a mail reference that facilitates the API.

Since each `Mail` is handled by a separate `coordinator`, it might've been suitable to let this mail reference simply be the PID of a `coordinator`. But the only responsibility of the `coordinator` should be maintaining the `worker` processes, and not

For this reason, I design the mail reference type to be a tuple `{MailServer, MailTag}`, where `MailServer` is the PID of the `mailfilter` server, and `MailTag` is the (unique) tag associated with the client's original call to `add_mail`.

This way, the `mailfilter` server can lookup the `MailTag` and address the appropriate `coordinator` (if any at that point in time).

Lines 2 and 3 of the above snippet shows how the mail reference is used in API calls involving mail references.

4.3 Mailfilter: Implementation

In this subsection, I will give description of key parts of my imlementation.

4.3.1 Implementation: `mailfilter` state

Below is a snippet of the type of the state of the `mailfilter` `gen_server`:

```
-type state() ::
  #{current  := integer(),           % current number of filters.
    capacity := integer() | infinite, % max filter capacity.

    coords   := coords(),  % mail tag -> coordinator   for this mail.
    configs  := configs(), % mail tag -> latest config for this mail.

    defaults := defaults() % default filters.
   }.

-type coords()   :: #{mail_tag() := pid()}.
-type configs()  :: #{mail_tag() := config()}.
-type defaults() :: #{label() := {filter(), data(), integer()}}.

-type config()   :: #{label() := result()}.
-type mail_tag() :: reference().
```

Since a `mailfilter` needs to maintain a certain capacity, it maintains the `current` number of filters attached, aswell as the `capacity` originally specified at server startup.

For each mail, the `mailfilter` maintains a PID of the coordinator for that mail, as well as the last known config for that mail. These are kept in the fields `coords` and `configs`, respectively.

Lastly, since the `mailfilter` needs to maintain a set of default filters for new incoming mails, the state has a field `defaults`, which is a map from mail tag to a tuple containing the filter and initial data associated with that label, as well as the size of this filter (more on this later).

4.3.2 Implementation: Capacity handling

When a filter is added to the `mailfilter` server by a client, the number of filters inherent in the possibly recursively defined filter can be determined by recursively iterating the `filter()` type.

But then, an important decision must be made. How to count the number of filters?

One might argue that the capacity required to evaluate a filter is equal to the number of processes required in order to evaluate that filter concurrently; then, the problem of determining the filter count becomes a little more subtle, since most filters can be evaluated sequentially.

In fact, as long as a filter contains no `group` type filters, the capacity needed to evaluate that filter remains constant however large the number of inherent filter functions grows.

Thus, to simplify matters, I make the following assumption to the design: instead of thinking about capacity in terms of the number of filters that can concurrently, I think of capacity as the CPU time (not wall-clock time) needed to evaluate filters.

I also make the simplification that I am only concerned with being *within a constant factor* of server processing capacity, meaning I can assume all filters to take `1` unit of CPU time to evaluate. Then, the counting of filters becomes simply:

```
filter_count({simple, _}) ->
  1;

filter_count({chain, Filters}) ->
  filter_list_count(Filters);

filter_count({group, Filters, _}) ->
  filter_list_count(Filters);

filter_count({timelimit, _, Filter}) ->
  filter_count(Filter).

filter_list_count(Filters) ->
  lists:sum(lists:map(fun (Filter) -> filter_count(Filter)
                      end, Filters)).
```

In the `mailfilter` server, the handling of capacity when adding filters then becomes (with lines of code pertaining to capacity highlighted):

```
-spec handle_cast(term() -> state()) -> {noreply, state()}.
handle_cast({add_filter, MailTag, Label, Filter, InitData},
            #{coords   := Coords,
              capacity := Cap,
              current  := Current} = State) ->

  NumFilters = filter_count(Filter),
  RoomForFilter = (Current + NumFilters) =< Cap,

  FilterExists = stateFilterExists(State, MailTag, Label),
  DoAddFilter = RoomForFilter and (not FilterExists),

  if DoAddFilter ->
         {ok, Coord} ->

           gen_server:cast(Coord, {add_filter, Label, Filter, InitData}),

           State2 = stateUpdateConfig(State, MailTag, Label, inprogress),

           Current2 = Current + NumFilters,

           {noreply, State2{current => Current2}};

         _MailUnknown  -> {noreply, State}
       end;
     not DoAddFilter -> {noreply, State}
  end;
```

Recall that the `mailfilter` maintains a map of default filters, whose values contain the number of filters in that particular default filter. These filters are added to the count when adding mails with `add_mail`.

Then, when mails are later unregistered with `enough`, the total number of filters for that mail is freed after stopping the `coordinator`.

4.3.3 Implementation: coordinating filter reults

As explained, each `coordinator` needs to maintain a number of `worker` processes for the filters attached to the given mail.

When a `worker` has finished evaluating its filter, it sends the result back to its `coordinator` in an asynchronous message tagged with `update_config`.

Depending on the type of filter result, the `coordinator` now has to figure out what to do with this result. Below is a snippet of the `coordinator`'s handling of config updates:

```erlang
handle_cast({update_config, Label, FilterResult},
            #{parent := Parent,
              tag    := MailTag} = State) ->

  State3 = #{config := Config2} =
    case FilterResult of

      {transformed, Mail} ->
        invalidate_all_but(Label, Mail, State);

      {both, Mail, Data} ->
        State2 = state_update_result(State, Label, {done, Data}),
        invalidate_all_but(Label, Mail, State2);

      {_JustOrUnchanged, Data} ->
        State2 = state_update_result(State, Label, {done, Data}),
        state_update_flag(State2, Label, valid);

      _Unexpected -> State
    end,

  case all_valid(State3) of
    true -> % notify parent of the new Config for this mail.
      gen_server:cast(Parent, {update_config, MailTag, Config2});

    _ -> nop % some workers need to recompute their filters first.
  end,

  {noreply, State3};
```

The highlighted lines handle each of the four types of filter results.

Notice that `{just, Data}` and `{unchanged, Data}` results share a case; I let `worker` processes return `{unchanged, Data}` rather than simply unchanged such that `coordinators` do not have to maintain initial filter data, but only finished results.

To understand how this function works:

`invalidate_all_but(Label, Mail, State)` is an auxiliary function which, given a filter label and the current `coordinator` state, invalidates all other filters but the one with label `Label` and prompts a re-evaluation with the new mail `Mail`, then returns the new state of the `coordinator`.

`state_update_result(State, Label, Data)` updates the config for `Label` with new data `Data`.

`all_valid(State)` returns true if all filters in the state are valid at the time of calling the function; else false.

4.3.4 Implementation: filter evaluation

Below is a snippet from `worker.erl` of my `worker` process' handling of `simple` and `chain` filter evaluation:

```erlang
run_filter({simple, FilterFun}, Mail, Data) ->
    FilterFun(Mail, Data);

run_filter({chain, Filters}, Mail0, Data0) ->
  InitAcc = {unchanged, Mail0, Data0},

  {FilterResult, _, DataResult} =
    lists:foldr(
      fun (Filter, {FilterResultAcc, MailAcc, DataAcc}) ->
        FilterResultAcc2 = combine_filter_results(FilterResultAcc,
                                                  run_filter(Filter, MailAcc, DataAcc)),

        {MailAcc2, DataAcc2} = update_accs(FilterResultAcc2, MailAcc, DataAcc),
        {FilterResultAcc2, MailAcc2, DataAcc2}
      end, InitAcc, Filters),

  case FilterResult of
    unchanged -> {unchanged, DataResult};
    _         -> FilterResult
  end;
```

The highlighted lines implement a left fold over the filters in the `chain`.

The initial value of the fold is `unchanged, Mail0, Data0`, where `Mail0` and `Data0` is the current state of the mail (which may differ from the original mail), and the initial data given by the client.

`combine_filter_results()` is a helper function which combines two filter results, since eg. a `{just, Data}` and a `{transformed, Mail}` must be combined into a `{both, Data, Mail}`.

4.4 Mailfilter: Assessment

4.4.1 Assessment: Simplifications

I have been forced to make a number of simplifications to the `mailfilter` specifications. In some particular order, they are:

- the simplification to server capacity as described in section 4.3.2.
- `group` filters are run sequentially, and thus differ from `chain` only in that the client supplies their own merge function. This decision was mostly for the purposes of testing capacity handling.
- I do not attempt to handle `timeout` filters.

4.4.2 Assessment: Limitations

Ignoring possible functionality breaking bugs in my implementation, and under the assumptions of the simplifications to the specifications mentioned above, then my program has no limitations that I am aware of.

4.4.3 Assessment: code quality

Even before validation testing, I am very confident that my implementation is *full of* bugs, and very possibly also additional limitations aside from those mentioned in the previous subsubsection.

I still very often experience *quirky* behaviour, eg. from the `coordinator` processes after invalidating many `workers` if those `workers` are evaluating non-terminating filters.

These types are in some part due to the hassles of working with the dynamic type system of Erlang; it can often be very hard to determine the source of bugs, especially synchronization bugs due to mishandling of protocols.

It is definitely, however, in even greater part due to sloppy coding on my own part. I have eg. neglected to really get into the habit of using `dialyzer`, since the few times I have used it, it has not caught many of my errors since most everything is typed `term()` or `any()` and there's not much to do about that.

In addition, whereas I with the APQL part of the exam followed a strict testing-first method, here, I was very quick to start coding out of fear of missing functionality by the time of deadline. This, evidently, was a bad decision.

Appendix

A Code: question 1, APQL

A.1 code/apql/src/ParserImpl.hs

```
module ParserImpl where

import Text.ParserCombinators.ReadP
import Control.Monad (guard)
import Control.Applicative ((<|>),  (<**>), liftA2)
import Data.Functor (($>))
import Data.Char (isAlpha, isDigit, isPrint)

import Utils ((.:))
import Types

-------------
---  API  ---
-------------
parseString :: String -> Either ErrMsg Program
parseString = parseMain parseProgram
run = parseString

parseMain :: ReadP a -> String -> Either ErrMsg a
parseMain parser str =
  case readP_to_S (parser <* eof) str of
    [(result, _)] -> Right result
    []            -> Left $ EUser     $ "Parsing error! Invalid program."
    _             -> Left $ EInternal $ "Internal error! Ambiguous parse."



----------------------------------
---  PROGRAM AND RULE PARSING  ---
----------------------------------
-- A program is zero or more rules each terminated with a period.
-- A program can, of course, befollowed by arbitrary whitespace
-- and arbitrarily many comments.
parseProgram :: ReadP Program
parseProgram = endBy parseRule (char' '.') <* skipComments

parseRule :: ReadP Rule
parseRule = flip Rule CTrue <$> parseAtom
        <|> rule' (parseAtom <* keyword "if") parseCond
        <|> rule' (parseAtom <* keyword "unless") (CNot <$> parseCond)
  where rule' = liftA2 Rule



----------------------
---  COND PARSING  ---
----------------------
parseCond :: ReadP Cond
parseCond = parseCondBinOps
```

```haskell

parseCondBinOps :: ReadP Cond
parseCondBinOps = chainr1 infix2   $ keyword "implies" $> CNot .: COr
  where infix2  = chainl1 infix3   $ keyword "or"      $> COr
        infix3  = chainl1 parseNot $ keyword "and"     $> CAnd

parseNot :: ReadP Cond
parseNot = (keyword "not" >> CNot <$> parseNot) <|> parseBottom

-- Not really the bottom of the parse tree, but rather the
-- bottommost internal node in the conceptual parse tree.
parseBottom :: ReadP Cond
parseBottom = CAtom <$> parseAtom
          <|> between (char' '(') (char' ')') parseCond
          <|> parseBoolConsts
          <|> parseTermBinOps

parseBoolConsts :: ReadP Cond
parseBoolConsts = keyword "true"  $> CTrue
              <|> keyword "false" $> CNot CTrue

parseTermBinOps :: ReadP Cond
parseTermBinOps = parseTerm <**> termBinOp <*> parseTerm
  where termBinOp = keyword "is"                   $> CEq
               <|> (keyword "is" >> keyword "not") $> CNot .: CEq



-------------------------------
---  TERM AND ATOM PARSING  ---
-------------------------------
parseTerm :: ReadP Term
parseTerm = (TVar <$> parseName) <|> (TData <$> parseData)

parseTermz :: ReadP [Term]
parseTermz = sepBy parseTerm (char' ',')

parseAtom :: ReadP Atom
parseAtom = Atom <$> parseName <*> parenthesized parseTermz


----------------------
---  MISC PARSING  ---
----------------------
parseName :: ReadP VName
parseName = lexeme $ name >>= \i -> look >>= guard .
              (i `notElem` reserved &&) . canFollowKeyword >> return i
  where name     = liftA2 (:) letter nameTail
        nameTail = many (letter <|> number <|> char '_')
        reserved = ["and", "or", "true", "false", "if",
                    "unless", "implies", "is", "not"]

parseData :: ReadP Data
parseData = lexeme $ between (char '"') (char '"') strConst
  where strConst = many ((string "\"\"" $> '"')            -- replace double quotes with single quotes;
                      <|> satisfy (\c -> isPrint c && c /= '"')) -- else parse anything printable but single quotes.


-- Comment parser.
-- FIXME: probably parses too much whitespace, but this
-- only a matter of efficiency and thus not a priority.
skipComments :: ReadP ()
skipComments = many (skipSpaces >> string "(*" >>
                manyTill (satisfy (const True)) (string "*)")) >> skipSpaces

```

```haskell
-- Keyword parser. Parse only successful if keyword is
-- followed by something that can legally follow a keyword.
keyword :: String -> ReadP String
keyword = (<* (look >>= guard . canFollowKeyword)) . string'

-- Also used by parseName.
canFollowKeyword :: String -> Bool
canFollowKeyword (c:_) = not (isDigit c || isAlpha c)
canFollowKeyword _     = True



---------------
--  HELPERS  --
---------------
-- I choose to skip whitespace *before* parsers, since this is what I have been
-- used to working with since before becoming familiar with Andrzej's advice of
-- parsing white-space *after* parsers.
lexeme :: ReadP a -> ReadP a
lexeme = (skipComments >>)

char' :: Char -> ReadP Char
char' = lexeme . char

string' :: String -> ReadP String
string' = lexeme . string

oneOf :: [Char] -> ReadP Char
oneOf = satisfy . flip elem

letter, number, anyChar :: ReadP Char
letter  = satisfy isAlpha
number  = satisfy isDigit
anyChar = satisfy (const True)

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

A.2 code/apql/src/PreprocessorImpl.hs

```haskell
module PreprocessorImpl where

import Data.List

import Types
import Utils



-- The simplified Clause type I use in stratification,
-- since here, a lot of information can be safely ignored.
-- A SimpleClause is a clause head and the positive/negative
-- references to other atoms made by this particular atom.
type SimpleClause = (PSpec, ([PSpec], [PSpec]))



----------------------
--- STRATIFICATION ---
----------------------
stratify :: IDB -> [PSpec] -> Either ErrMsg [[PSpec]]
stratify (IDB ips clauses) eps =
  if ips `disjoint` eps then
     makeStrata (map simplify clauses) ips []
  else Left $ EUser $ "Cannot stratify: Overlap in ex- and intensionals."

  where simplify (Clause atom posRefs tests) =
          (pSpec atom, (map pSpec posRefs, getNegRefs tests))
        getNegRefs tests = [pSpec a | (TNot a) <- tests]


-- Since we can ignore a lot of information during stratification,
-- makeStrata and makeStratum use the SimpleClause type I have defined in Util.hs.
makeStrata :: [SimpleClause] -> [PSpec] -> [[PSpec]]
           -> Either ErrMsg [[PSpec]]
makeStrata _ [] placed = return placed
makeStrata clauses unplaced placed = do
  i <- makeStratum clauses unplaced []
  makeStrata clauses (unplaced \\ i) (placed ++ [i])

makeStratum :: [SimpleClause] -> [PSpec] -> [PSpec]
            -> Either ErrMsg [PSpec]
makeStratum _ [] _ = Left $ EUser "Cannot stratify: empty stratum."
makeStratum clauses stratum removed =
  if stratum' == stratum then return stratum'
  else makeStratum clauses' stratum' removed'

  where stratum' = filter keep stratum
        clauses' = filter ((`elem` stratum') . fst) clauses
        removed' = removed ++ (stratum \\ stratum')

        -- keep p if it has no neg references to atoms in its stratum,
        -- and no pos references to atoms removed in this iteration.
        keep p = neg `disjoint` stratum && pos `disjoint` removed
          where (pos, neg) = foldl combine ([], []) $ map snd $
                             filter ((== p) . fst) clauses



----------------------
```

```haskell
--- CLAUSIFICATION ---
----------------------
clausify :: Program -> Either ErrMsg IDB
clausify program = makeIDB <$> makeClauses (transformProgram program)
  where makeIDB clauses = IDB (makePSpecs clauses) clauses
        makePSpecs = nub . map makePSpec
        makePSpec (Clause (Atom pname args) _ _) = (pname, length args)

makeClauses :: Program -> Either ErrMsg [Clause]
makeClauses = mapM (\(Rule atom cond) ->
  verifyClause $ uncurry (Clause atom) $ makeClause cond)

makeClause :: Cond -> ([Atom], [Test])
makeClause (CAnd c1 c2)       = makeClause c1 `combine` makeClause c2
makeClause (CAtom a)          = ([a], [])
makeClause (CNot (CAtom a))   = ([], [TNot a])
makeClause (CEq t1 t2)        = ([], [TEq  t1 t2])
makeClause (CNot (CEq t1 t2)) = ([], [TNeq t1 t2])
makeClause _                  = ([], [])

verifyClause :: Clause -> Either ErrMsg Clause
verifyClause clause@(Clause (Atom _ head_args) atoms ts) =
  if need `subsetOf` have
  then return clause
  else Left $ EUser $ "Cannot verify clause: " ++ show clause ++ ". Missing variables: "
                      ++ show (need \\ have)
  where vars terms = [var | var@TVar{} <- concat terms]
        have = vars $ map args atoms
        need = vars $ head_args : [[a, b] | TNeq a b <- ts] ++
                                  [[a, b] | TEq  a b <- ts]



------------------------------
--- PROGRAM TRANSFORMATION ---
------------------------------
-- AND distribution can uncover new opportunities for "not flipping", and
-- possibly vice versa, so to be safe, I repeat the transformation until
-- no more transformations apply. Perhaps rule splitting should not be
transformProgram :: Program -> Program
transformProgram prog = if prog' == prog then prog'
                                          else transformProgram prog'
  where prog'     = transform prog
        transform = splitRules . distribAnds . flipNots

-- "Flip" logical negations in conditions for all rules in program.
flipNots :: Program -> Program
flipNots = map (\(Rule a cond) -> Rule a $ flipNot cond)

-- Distribute AND conditions for all rules in program.
distribAnds :: Program -> Program
distribAnds = map (\(Rule a cond) -> Rule a $ distribAnd cond)

-- Split rules of the form "atom if c1 or c2", where c1 and c2 and conditions,
-- into multiple rules - do so recursively for all rules in program. Also,
-- remove any rules of the form "atom if false".
splitRules :: Program -> Program
splitRules = concatMap splitRule

flipNot :: Cond -> Cond
flipNot (CNot (CAnd c1 c2)) = COr  (CNot (flipNot c1)) (CNot (flipNot c2))
flipNot (CNot (COr  c1 c2)) = CAnd (CNot (flipNot c1)) (CNot (flipNot c2))
flipNot (CNot (CNot c)) = flipNot c
flipNot (CAnd c1 c2) = CAnd (flipNot c1) (flipNot c2)
flipNot (COr  c1 c2) = COr  (flipNot c1) (flipNot c2)
```

```
flipNot c = c

distribAnd :: Cond -> Cond
distribAnd (CAnd _ (CNot CTrue)) = CNot CTrue
distribAnd (CAnd (CNot CTrue) _) = CNot CTrue
distribAnd (CAnd c1 CTrue) = distribAnd c1
distribAnd (CAnd CTrue c2) = distribAnd c2
distribAnd (CAnd c1 (COr c2 c3)) = COr (CAnd c1 c2) (CAnd c1 c3)
distribAnd (CAnd (COr c1 c2) c3) = COr (CAnd c3 c1) (CAnd c3 c2)
distribAnd (CAnd c1 c2) = CAnd (distribAnd c1) (distribAnd c2)
distribAnd (COr  c1 c2) = COr  (distribAnd c1) (distribAnd c2)
distribAnd (CNot c)     = CNot (distribAnd c)
distribAnd c = c

splitRule :: Rule -> Program
splitRule (Rule atom (COr c1 c2)) =
  splitRule (Rule atom c1) ++ splitRule (Rule atom c2)
splitRule (Rule _ (CNot CTrue)) = []
splitRule rule = [rule]
```

A.3 code/apql/src/EngineImpl.hs

```haskell
module EngineImpl where

import Control.Monad (guard, zipWithM, foldM)
import Data.Functor (($>))
import Data.Maybe (mapMaybe, fromMaybe)
import qualified Data.Set as S
import qualified Data.Map as M

import Types
import Utils (pSpec, args, (.:))

-- The internals of my execution engine use a different representation of
-- strata. A stratum is a list of tuples of atoms and their clauses.
type Strata  = [Stratum]
type Stratum = [(PSpec, [Clause])]


-- The type I use to represent EDB's internally.
-- Using a Map makes lookup and merging of EDB's easier.
type EDBM = M.Map PSpec ETable

-- an instance is a mapping of variables to data (these both have type Term,
-- but is essentially a map of TVar to TData)
type Instance = [(Term, Term)]
        -- options = edb `edbGetExtens` atom :: [Row] -- ETable
        -- instances atom = mapMaybe (makeInstance atom) options



----------------------------
--- EXECUTION ENGINE API ---
----------------------------
execute :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
execute _ _ _ = Left $ EUnimplemented "Failed attempt at an implementation :("



----------------------------------------------------
--- FAILED EXECUTION ENGINE IMPLEMENTATION BELOW ---
----------------------------------------------------
execute_ :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
execute_ (IDB preds clauses) strata edb =

  if preds /= concat strata then Left $ EUser "Mismatch between input\
                                              \ predicates and strata!"
  -- internalize strata and EDB, then run evalStrata.
  else
    M.toList <$> foldM (flip evalStratum) edb' strata' -- evalStrata strata' edb'

  where strata' = map toStratum strata
        edb'    = M.fromListWith S.union edb

        toStratum    = map (\p -> (p, clausesFor p))
        clausesFor p = filter (`isClauseFor` p) clauses
        isClauseFor (Clause a _ _) p = pSpec a == p


evalStratum :: Stratum -> EDBM -> Either ErrMsg EDBM
evalStratum stratum edb =
  if edb' == edb then return edb' -- nothing's changed this iteration? return! :)
```

```haskell
  else evalStratum stratum edb'

  where
    contributions = foldM (\edb' (p, cs) -> contribution p cs edb') edb stratum

    contribution :: PSpec -> [Clause] -> EDBM -> Maybe EDBM
    contribution _ [] edb = Just edb
    contribution p (c@(Clause head _ _):cs) edb = do
      contribution p cs $ edbUnions $ mapMaybe (applyClause edb c) insts

      where insts  = mapMaybe (makeInstance head) extens
            extens = edb `extensFor` p

    applyClause :: EDBM -> Clause -> Instance -> Maybe EDBM
    applyClause edb (Clause head posrefs tests) inst =
      matchAtom edb inst head >>= \head' ->  -- match clause head.
        mapM (matchAtom edb inst) posrefs >> -- match and verify pos references.
          mapM (checkTest edb inst) tests $> -- match and verify tests.
            edbSingletonFromAtom head'       -- add contribution to an EDBM.



    checkTest :: EDBM -> Instance -> Test -> Maybe ()
    checkTest edb inst (TNot atom) = matchAtom edb inst atom $> ()
    checkTest _ inst (TEq  t1 t2) = (guard .: (==)) (matchTerm inst t1)
                                                    (matchTerm inst t2)
    checkTest _ inst (TNeq t1 t2) = (guard .: (/=)) (matchTerm inst t1)
                                                    (matchTerm inst t2)

    matchTerm :: Instance -> Term -> Maybe Term
    matchTerm _    dat@(TData _) = Just dat
    matchTerm inst var@(TVar  _) = lookup var inst

    matchAtom :: EDBM -> Instance -> Atom -> Maybe Atom
    matchAtom edb inst (Atom name args) =
      mapM (matchTerm inst) args >>= \args' ->
        if edbInstExists edb (Atom name args')
        then Just (Atom name args') else Nothing

    edb' = edb `edbUnion` fromMaybe edb contributions



-- An atom is attempted instantiated with a row of data by
-- a pairwise match-up of atom arguments to row entries.
makeInstance :: Atom -> Row -> Maybe Instance
makeInstance (Atom name args) row = zipWithM matchup args row
  where matchup (TVar _)   d  = Just (TVar name, TData d)
        matchup (TData d1) d2 = if d1 == d2 then Just (TVar name, TData d2)
                                            else Nothing


-------------------------------
--- EDBM SPECIFIC UTILITIES ---
-------------------------------
edbInstExists :: EDBM -> Atom -> Bool
edbInstExists edb atom = args' `S.member` etable
  where etable = M.findWithDefault S.empty (pSpec atom) edb

        args'  = [dat | (TData dat) <- args atom]

edbSingletonFromAtom :: Atom -> EDBM
edbSingletonFromAtom atom@(Atom _ args) =
  M.singleton (pSpec atom) $ S.singleton [dat | (TData dat) <- args]
```

```
edbUnion :: EDBM -> EDBM -> EDBM
edbUnion = M.unionWith S.union

edbUnions :: [EDBM] -> EDBM
edbUnions = M.unionsWith S.union

extensFor :: EDBM -> PSpec -> [Row]
extensFor = S.toList .:  flip (M.findWithDefault S.empty)-- extensFor'
```

A.4 code/apql/src/Utils.hs

```haskell
module Utils where

import Types
import Data.List (intersect)



--------------------
--- ATOM HELPERS ---
--------------------
pSpec :: Atom -> PSpec
pSpec (Atom p args) = (p, length args)

args :: Atom -> [Term]
args (Atom _ args') = args'



--------------------
--- MISC UTILITY ---
--------------------
-- composition of a binary and a unary operator.
(.:) :: (c -> d) -> (a -> b -> c) -> a -> b -> d
(.:) = (.) . (.)

-- are these two lists disjoint??
disjoint :: Eq a => [a] -> [a] -> Bool
disjoint = null .: intersect

-- is xs a subset of ys?
subsetOf :: Eq a => [a] -> [a] -> Bool
subsetOf xs ys = all (`elem` ys) xs

-- given two tuples of lists, pairwisely appends these lists.
combine :: ([a], [b]) -> ([a], [b]) -> ([a], [b])
combine (as, bs) (xs, ys) = (as ++ xs, bs ++ ys)
```

A.5 APQL: Test suite

A.5.1 code/apql/tests/suite1/WhiteBox.hs

```haskell
import Test.Tasty
import Test.Tasty.HUnit

import ParserTests
import PreprocessorTests

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

allTests :: TestTree
allTests = testGroup "My APQL unit tests"
  [
   parserTests
  ,preprocessorTests
  ]
```

A.5.2 code/apql/tests/suite1/ParserTests.hs

```haskell
module ParserTests where

import Test.Tasty
import Test.Tasty.HUnit

import Types
import MyTestUtils



-------------------
--- PARSER TEST ---
-------------------
parserTests :: TestTree
parserTests = testGroup ">>>> Parser tests"
  [mainParserTests
  ,miscTests
  ]

mainParserTests :: TestTree
mainParserTests = testGroup ">> Cond and rule tests"
  [

   posP "the empty program" "" []

  ,posP "right-association of implies" "a() if b() implies c() implies d()."
     [Rule (Atom "a" []) (CNot (COr (CAtom (Atom "b" [])) (CNot (COr (CAtom (Atom "c" [])) (CAtom (Atom "d" []))))))]

  ,posP "left-association of or" "a() if b() or c() or d()."
     [Rule (Atom "a" []) (COr (COr (CAtom (Atom "b" [])) (CAtom (Atom "c" []))) (CAtom (Atom "d" [])))]

  ,posP "left-association of and" "a() if b() and c() and d()."
```

```
     [Rule (Atom "a" []) (CAnd (CAnd (CAtom (Atom "b" [])) (CAtom (Atom "c" []))) (CAtom (Atom "d" [])))]

  ,posP "precedence of and/or/implies" "a() if b() and c() implies d() or e()."
     [Rule (Atom "a" []) (CNot (COr (CAnd (CAtom (Atom "b" [])) (CAtom (Atom "c" [])))
                                    (COr (CAtom (Atom "d" []))
                                         (CAtom (Atom "e" []))))))]

  ,posP "associativity of logical negation" "p() if not x() and not not not x()."
     [Rule (Atom "p" []) (CAnd (CNot (CAtom (Atom "x" []))) (CNot (CNot (CNot (CAtom (Atom "x" []))))))]
  ,posP "precedence of logical negation" "p() if not x() and not not not x()."
     [Rule (Atom "p" []) (CAnd (CNot (CAtom (Atom "x" []))) (CNot (CNot (CNot (CAtom (Atom "x" []))))))]
  ,posP "precedence of parenthesized expressions" "p() if not (x() or y())."
     [Rule (Atom "p" []) (CNot (COr (CAtom (Atom "x" [])) (CAtom (Atom "y" []))))]

  ,posP "precedence of \"is\"/\"is not\" 1:" "p() if not x is y."
     [Rule (Atom "p" []) (CNot (CEq (TVar "x") (TVar "y")))]

  ,posP "precedence of \"is\"/\"is not\" 2" "p() if not x is \"y\"."
     [Rule (Atom "p" []) (CNot (CEq (TVar "x") (TData "y")))]


  ,negP "non-association of \"is\""     "p() if x is y is z."
  ,negP "non-association of \"is not\"" "p() if x is not y is not z."

  ,negP "missing closing parenthesis" "foo(a, b) if (not (bar(y) and (x is y or baz()))."
  ,negP "unexpected closing parenthesis" "foo(a, b) if not bar(y) and x is y or baz())."
  ,negP "parenthesis around terms" "foo(a, b) if (\"x\") is y."
  ,negP "missing rule terminating period" "foo(a, b)"

  ,testGroup "> A couple of Cond and Rule parsing properties" $
    [
     "p() if x is not y."      `equalP` "p() if not x is y."
    ,"p() unless q()."         `equalP` "p() if not q()."
    ,"p() if a() implies b()." `equalP` "p() if not (a() or b())."
    ,"p()."                    `equalP` "p() if true."

    ,"p(a, b, c) if (not((not((bar(x))))and foo is baz))." `equalP`
     "p(a, b, c) if not (not bar(x) and foo is baz)."
    ]
  ]



miscTests :: TestTree
miscTests = testGroup ">> Misc parser tests"
  [
   stringConstTests
  ,keywordHandlingTests
  ,whitespaceHandlingTests
  ]

stringConstTests :: TestTree
stringConstTests = testGroup "> String constant tests"
  [
   posP "simple string const" "foo(\"barrrr\")."
     [Rule (Atom "foo" [TData "barrrr"]) CTrue]


  ,posP "string const with printable whitespace" "foo(\"   hej der \")."
     [Rule (Atom "foo" [TData "   hej der "]) CTrue]

  ,posP "double quotes in string const" "foo(\"\"\"fooo\"\"\")."
     [Rule (Atom "foo" [TData "\"fooo\""]) CTrue]

  ,posP "nested double quotes in string const" "foo(\"\"\"foo\"\"fooo\"\"\"\"bar\")."
```

```
     [Rule (Atom "foo" [TData "\"foo\"fooo\"\"bar"]) CTrue]

  ,negP "printable but non-ascii characters" "foo(\"€€€apeæøå\")."
  ,negP "non-printable characters" "foo(\"'\255', '\19'\")."
  ,negP "non-printable whitespace" "foo(\"    \t\")."
  ,negP "bad double quotes in string const" "foo(\"\"hej der\"\"\")."
  ]

keywordHandlingTests :: TestTree
keywordHandlingTests = testGroup "> Keyword handling tests"
  [
   posP "respected keywords" "foo() if a is b."
     [Rule (Atom "foo" []) (CEq (TVar "a") (TVar "b"))]
  ,posP "notx correctly parsed as a variable" "foo(x, y) if y is notx."
     [Rule (Atom "foo" [TVar "x",TVar "y"]) (CEq (TVar "y") (TVar "notx"))]
  ,posP "case sensitive reserved keywords" "foo(x, y) if uNless is And."
     [Rule (Atom "foo" [TVar "x",TVar "y"]) (CEq (TVar "uNless") (TVar "And"))]

  ,negP "reserved keyword as term"       "p() if (and is yes)."
  ,negP "reserved keyword as data const" "p() if and"
  ,negP "reserved keyword as pred name"  "p() if and"
  ]


whitespaceHandlingTests :: TestTree
whitespaceHandlingTests = testGroup "> Whitespace handling tests" $
  (map ($ pos_tests_expected)
 [
  posP "sane amount of whitespace" $ "foo(a,b) unless\n\t\t (car(a) implies man(b)) and baz(b)\n\t or\n\t\t" ++
                                    "true and \"b\" is not \"a\".\n\nbaz(\"b\")."

 ,posP "minimal whitespace (parentheses instead)" $ "foo(a,b)unless(car(a)implies(man(b)))and(baz(b))or(true)" ++
                                                   "and(\"b\"is not\"a\").baz(\"b\")."

 ,posP "spaces all over" $ " foo ( a , b ) unless ( car ( a ) implies ( man ( b ) ) ) and ( baz ( b ) ) or" ++
                              " ( true ) and ( \"b\" is not \"a\" ) . baz ( \"b\" ) . "

 ,posP "comments as whitespace" $ "foo(a,b)(**)unless(**)(car(a)(**)implies(**)man(b))(**)and(**)baz(b)(**)" ++
                                 "or(**)true(**)and(**)\"b\"(**)is(**)not(**)\"a\".(**)baz(\"b\")."

 ,posP "tab and newlines" $ "foo(a,b)\n unless\n (car(a)\t\t implies man(\nb)) and baz(b\n) or true\n" ++
                           " and \"b\" is not \"a\".\nbaz(\"b\")."

 ,posP "all types of whitespace" $ "foo(a,b)\n\r unless\v\v\n (car(a)\t\t\n implies\f\f\f man(\nb)) and\r" ++
                                  " baz(b\n) or\v true\n and \"b\" is not \"a\".\nbaz(\"b\")."
 ])
 ++
 [
  posP "just a comment"  "(*just a comment*)" []
 ,posP "trailing comment"  "foo().(**)" [Rule (Atom "foo" []) CTrue]
 ,posP "leading comment"  "(*hej*)foo()."  [Rule (Atom "foo" []) CTrue]
 ,negP "missing whitespace 1" "p() ifq()."
 ,negP "missing whitespace 2" "p() if yes isno."
 ,negP "missing whitespace 3" "p() if yesis no."
 ,negP "missing whitespace 4" "p() if bar() andbaz()."
 ]
 where pos_tests_expected =
         [Rule (Atom "foo" [TVar "a",TVar "b"])
                 (CNot (COr
                         (CAnd (CNot (COr (CAtom (Atom "car" [TVar "a"]))
                                          (CAtom (Atom "man" [TVar "b"]))))
                               (CAtom (Atom "baz" [TVar "b"])))
                         (CAnd
                           CTrue
                           (CNot (CEq (TData "b") (TData "a")))))),
```

```
            Rule (Atom "baz" [TData "b"]) CTrue]
```

A.5.3 code/apql/tests/suite1/PreprocessorTests.hs

```
module PreprocessorTests where

import Test.Tasty
import Test.Tasty.HUnit
import Data.List (sort)

import Types
import MyTestUtils



preprocessorTests :: TestTree
preprocessorTests = testGroup ">>>> Preprocessor tests"
  [
   transformTests
  ,clausifyTests
  ,stratifyTests
  ]



transformTests :: TestTree
transformTests = testGroup (">> Tests of each of the transformation "
                            ++ "equivalences in the assignment.")
  [
   posT "negation equivalence 1" "p() if not (a() and b())."
       [Rule (Atom "p" []) (CNot (CAtom (Atom "a" []))),
        Rule (Atom "p" []) (CNot (CAtom (Atom "b" [])))]
  ,posT "negation equivalence 2" "p() if not (a() or b())."
       [Rule (Atom "p" []) (CAnd (CNot (CAtom (Atom "a" [])))
                                 (CNot (CAtom (Atom "b" []))))]
  ,posT "negation equivalence 3" "p() if not (not a())."
       [Rule (Atom "p" []) (CAtom (Atom "a" []))]


  ,posT "associativity of or splitting" "foo(a, b) if x is y or a is not b or foo(b, b)."
                  [Rule (Atom "foo" [TVar "a",TVar "b"])
                        (CEq (TVar "x") (TVar "y")),
                   Rule (Atom "foo" [TVar "a",TVar "b"])
                        (CNot (CEq (TVar "a") (TVar "b"))),
                   Rule (Atom "foo" [TVar "a",TVar "b"])
                        (CAtom (Atom "foo" [TVar "b",TVar "b"]))]


  ,posT "and distribution equivalence" "p() if x is y and (a() or b())."
                  [Rule (Atom "p" []) (CAnd (CEq (TVar "x") (TVar "y"))
                                            (CAtom (Atom "a" []))),
                   Rule (Atom "p" []) (CAnd (CEq (TVar "x") (TVar "y"))
                                            (CAtom (Atom "b" [])))]
  ,posT "discard \"atom and false\" rules" "q() if p(). p() if false."
                  [Rule (Atom "q" []) (CAtom (Atom "p" []))]

  ,posT "multiple rounds of transformations needed"
        "p(x) if q(x) and not (r(x) and x is not a)."
                 [Rule (Atom "p" [TVar "x"])
                       (CAnd (CAtom (Atom "q" [TVar "x"]))
                             (CNot (CAtom (Atom "r" [TVar "x"])))),
                  Rule (Atom "p" [TVar "x"])
                       (CAnd (CAtom (Atom "q" [TVar "x"]))
                             (CEq (TVar "x") (TVar "a")))]
```

```haskell

  , testGroup "> A couple of transformation properties" $
    [
     posT "transforming empty program" "" []
    ,equalT "or splitting" "foo(a, b) if bar(a, b) or baz(c, e)."
                           "foo(a, b) if bar(a, b). foo(a, b) if baz(c, e)."
    ,equalT "associativity of or splitting" "p() if (a() or (b() or (c() or d()) ) or e())."
                                            "p() if ((((a() or b()) or c()) or d()) or e())."
    ,equalT "commutativivity of or splitting" "p() if (a() or (b() or (c() or d()) ) or e())."
                                              "p() if ((((b() or d()) or a()) or e()) or c())."

    ]
  ]

clausifyTests :: TestTree
clausifyTests = testGroup ">> clausify tests"
  [
   posC "trivial satisfaction of variable occurence condition 1" "p() if a(b, c, \"d\", e)."
     (IDB [("p", 0)] [Clause (Atom "p" []) [Atom "a" [TVar "b", TVar "c", TData "d", TVar "e"]] []])

  ,posC "trivial satisfaction of variable occurence condition 2" "p(\"a\") if foo()."
     (IDB [("p", 1)] [Clause (Atom "p" [TData "a"]) [Atom "foo" []] []])



  ,posC "recursive satisfaction of occurence condition" "p(a, b, c) if p(c, b, a)."
     (IDB [("p", 3)] [Clause (Atom "p" [TVar "a", TVar "b", TVar "c"]) [Atom "p" [TVar "a", TVar "b", TVar "c"]] []])

  ,posC "many clauses, all satisfy occurence condition" "p(a) if bar(a). p(\"b\", \"c\") if x is not y and bar(x, y)."
     (IDB [("p", 1),  ("p", 2)]
          [Clause (Atom "p" [TVar "a"])             [Atom "bar" [TVar "a"]]           [],
           Clause (Atom "p" [TData "b", TData "c"]) [Atom "bar" [TVar "x", TVar "y"]] [TNeq (TVar "x") (TVar "y")]
          ])

  ,posCNoSort "simple test of correct preserving of order of rules in result IDB"
              "p(\"a\"). bar(a, b) if x is b and car(a, b, x). bar(a, b) if btl(b, a) and not not true. p(\"b\")."
     (IDB [("p",1), ("bar",2)]
          [Clause (Atom "p"   [TData "a"])         []                                        [],
           Clause (Atom "bar" [TVar "a",TVar "b"]) [Atom "car" [TVar "a",TVar "b",TVar "x"]] [TEq (TVar "x") (TVar "b")],
           Clause (Atom "bar" [TVar "a",TVar "b"]) [Atom "btl" [TVar "b",TVar "a"]]          [],
           Clause (Atom "p"   [TData "b"])         []                                        []])

  ,negC "test var missing from non-negated atom vars"   "p1() if x is not \"a\"."

  ,negC "head var missing from non-negated atom vars"   "p(a) if not bar(a)."

  ,negC "head var missing from non-negated atom vars 2" "p(a) if bar()."

  ,negC "head and test var missing from non-negated atom vars" "p(x, z) if x is y and q(x)."

  ,negC "many clauses, only one breaks occurence condition"
          "p(a) if and c(a). p(e) if not b(a) and c(e). p(a) if not(a)."

  ]


stratifyTests :: TestTree
stratifyTests = testGroup ">> stratify tests"
  [
  posS "Assert empty program produces empty strata"
          ("", []) []

 ,posS "Example stratification from the assignment text"
          ("p() if q() and r(). p() if p() and not r(). q() if q()" ++
           " and not s(). s() if r().", [("r", 0)])
```

```
          [[("s", 0)], [("p", 0), ("q", 0)]]

 ,posS "Multiple strata" ("different3(a, b, a) if not same(a, a) and is_happy(a, b)." ++
                          "same(a, b) if not different2(a, b) and error(b, a)." ++
                          "error(a, b) if is_dead_var(a) and is_dead_var(b)." ++
                          "tautology(yes, no) if ((yes(yes) and no(no)) implies error(a, b)) " ++
                          "and same(yes, no)."
                          , [("different2", 2), ("is_dead_var", 1)])
                       [[("same", 2), ("error", 2)], [("different3", 3), ("tautology", 2)]]

 ,posS "`is_taller` example"
   ("is_taller(a, b) if taller(a, b). is_taller(a, b) if " ++
              "taller(a, c) and is_taller(c, b).", [("taller", 2)]) [[("is_taller", 2)]]

 ,negS "Overlap in ex- and intensional predicates"
        ("p() if q() and r(). p() if p() and not r(). q() if q() and not s(). s() if r().",
              [("foo", 17), ("bar", 8), ("p", 0), ("a", 3)])

 ,negS "Mutually recursive negative references within same stratum"
        ("foo(a, b) if not bar(b, a) and baz(a, b). bar(a, b) if not foo(b, b) and baz(b, a).",
                       [])
  ]
```

A.5.4 code/apql/tests/suite1/MyTestUtils.hs

```
{-# LANGUAGE StandaloneDeriving #-}
module MyTestUtils where

import Data.List (sort)

import Test.Tasty
import Test.Tasty.HUnit
import Types

import PreprocessorImpl
import ParserImpl



-----------------------------------------
--- GENERIC TESTING UTILITY FUNCTIONS ---
-----------------------------------------
posTestGeneric :: (Show b, Eq b) =>
                  (a -> Either ErrMsg b) -> String -> a -> b -> TestTree
posTestGeneric test_me test_name input expected =
  testCase test_name $ (test_me input) @?= (Right expected)


negTestGeneric :: (Eq a, Show b) =>
                  (a -> Either ErrMsg b) -> String -> a -> TestTree
negTestGeneric test_me test_name input = testCase ("*NEG* " ++ test_name) $
  case (test_me input) of
    Left (EUser _)            -> return ()
    Left (EUnimplemented err) -> assertFailure err
    Left (EInternal err)      -> assertFailure $ ">> Unexpected internal error: " ++ err
    Right p                   -> assertFailure $ ">> Unexpected success! Got: "   ++ show p


equalGeneric :: (Show b, Eq b) =>
                (a -> Either ErrMsg b) -> String -> a -> a -> TestTree
equalGeneric test_me test_name input1 input2 =
  testCase test_name $ (test_me input1) @?= (test_me input2)










----------------------------------------
--- PARSER TESTING UTILITY FUNCTIONS ---
----------------------------------------
posP = posTestGeneric parseString
negP = negTestGeneric parseString
equalP a b = equalGeneric parseString (a ++ " <=> " ++ b) a b



----------------------------------
--- CLAUSIFY TESTING UTILITIES ---
----------------------------------
deriving instance Ord Rule
```

```haskell
deriving instance Ord Atom
deriving instance Ord Term
deriving instance Ord Cond
deriving instance Ord IDB
deriving instance Ord Test
deriving instance Ord Clause

sortIDB :: IDB -> IDB
sortIDB (IDB ps cs) = IDB (sort ps) (map sortClause cs)
  where sortClause (Clause atom atoms tests) = Clause (sortAtom atom) (map sortAtom atoms) (sort tests)
        sortAtom (Atom name args) = Atom name (sort args)

clausify' input = sortIDB <$> (parseString input >>= clausify)


posC test_name input expected = posTestGeneric clausify' test_name input (sortIDB expected)
negC   = negTestGeneric clausify'
equalC = equalGeneric   clausify'

-- used to test preserving of order of rules in the output IDB.
clausifyNoSort input = parseString input >>= clausify
posCNoSort = posTestGeneric clausifyNoSort


-- transform testing
transform' str = sort . transform <$> parseString str
posT test_name input expected = posTestGeneric transform' test_name input (sort expected)
negT    = negTestGeneric transform'
equalT  = equalGeneric   transform'



---------------------------------
--- STRATIFY TESTING UTILITIES ---
---------------------------------
stratify' (input, eps) = (parseString input >>= clausify >>= flip stratify eps)

posS   = posTestGeneric stratify'
negS   = negTestGeneric stratify'
equalS = equalGeneric   stratify'
```

B Code: question 2, mailfilter

B.1 code/mailfilter/src/mailfilter.erl

```erlang
-module(mailfilter).

-behaviour(gen_server).

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

% gen_server mandated exports.
-export([ init/1
        , terminate/2
        , handle_cast/2
        , handle_call/3
        ]).

-export([safeMapsRemove/2
        ]).



%%%%%%%%%%%
%%% API %%%
%%%%%%%%%%%
-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()}
                         | {both, mail(), data()}
                         | unchanged.

-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).

-type my_error() :: {error, any()} | {error, internal_err}.


-spec start(integer()) -> {ok, pid()} | my_error().
start(Cap) ->
```

```erlang
  case gen_server:start(?MODULE, Cap, []) of
    {ok, MailServer} -> {ok, MailServer};
    {error, Reason}  -> {error, Reason};
    _                -> {error, internal_err}
  end.


-spec stop(any()) -> {ok, ext_state()} | my_error().
stop(MailServer) ->
  case gen_server:call(MailServer, stop) of
    {ok, State}     -> State;
    {error, Reason} -> {error, Reason};
    _               -> {error, internal_err}
  end.


-spec add_mail(pid(), mail()) -> {ok, ext_mail_ref()} | my_error().
add_mail(MS, Mail) ->
  case gen_server:call(MS, {add_mail, Mail}) of
    {ok, MailRef}   -> {ok, MailRef};
    {error, Reason} -> {error, Reason};
    _               -> {error, internal_err}
  end.


-spec get_config(ext_mail_ref()) -> {ok, config()} | my_error().
get_config({MailServer, MailTag}) ->
  case gen_server:call(MailServer, {get_config, MailTag}) of
    {ok, Config}    -> {ok, Config};
    {error, Reason} -> {error, Reason};
    _               -> {error, internal_err}
  end.


-spec default(pid(), label(), filter(), data()) -> any().
default(MailServer, Label, Filter, InitData) ->
  gen_server:cast(MailServer, {add_default, Label, Filter, InitData}).


-spec enough(ext_mail_ref()) -> any().
enough({MS, MailTag}) ->
  gen_server:cast(MS, {enough, MailTag}).


-spec add_filter(ext_mail_ref(), label(), filter(), data()) -> any().
add_filter({MailServer, MailTag}, Label, Filter, InitData) ->
  gen_server:cast(MailServer, {add_filter, MailTag, Label, Filter, InitData}).






%%%%%%%%%%%%%%%%
%%% INTERNALS %%%
%%%%%%%%%%%%%%%%
-type state() ::
  #{current  := integer(),            % current number of filters.
    capacity := integer() | infinite, % max filter capacity.

    coords   := coords(),  % mail ref -> coordinator   for this mail.
    configs  := configs(), % mail ref -> latest config for this mail.

    defaults := defaults() % default filters.
   }.
```

```erlang

-type coords()   :: #{mail_tag() := pid()}.
-type configs()  :: #{mail_tag() := config()}.
-type defaults() :: #{label() := {filter(), data(), integer()}}.

-type mail_tag() :: reference().
-type ext_mail_ref() :: {pid(), mail_tag()}.


-type config() :: #{label() := result()}.
-type ext_state() :: list({mail(), list(labelled_result())}).



-spec handle_call(term(), {pid(), term()}, state()) ->
  {reply, {ok, term()}, state()} | {reply, {error, term()}, state()}.
handle_call({add_mail, Mail}, {_, Tag},
            #{configs := Configs, defaults := Defaults,
              current := Current} = State) ->


  Me = self(),
  MailTag = Tag, % TODO: different format?. Could also be make_ref().
                 % For now, just use Tag associated with sender's call.

  % spawn a new coordinator for this mail.
  case coordinator:start(Mail, MailTag, Defaults, Me) of
    {ok, Coord} ->

      State2 = add_coord(MailTag, Coord, State),
      DefaultsLabels = maps:keys(Defaults),

      % for each default filter, init emptry entry in the config for Mail.
      DefaultResults = maps:from_list(
                         lists:map(fun (Label) -> {Label, inprogress}
                                   end, DefaultsLabels)),
      % update configs and state.
      Configs2 = Configs#{MailTag => DefaultResults},
      NumNewFilters = default_filters_count(Defaults),
      State3 = State2#{configs => Configs2,
                       current => Current + NumNewFilters},

      {reply, {ok, {Me, MailTag}}, State3};

    {error, Reason} -> {reply, {error, Reason}, State};
    _               -> {reply, {error, coord_spawn_error}, State}
  end;


% Returns the latest known config for Mail. Does not prompt the given
% coordinator to send back updated config.
% TODO: is this assumption wrong/dangerous?
handle_call({get_config, MailTag}, _From,
            #{configs := Configs} = State) ->

  Reply =
    case safeMapsFind(MailTag, Configs) of
      {ok, Config} ->
        case Config of
          #{} = Config2 -> {ok, maps:to_list(Config2)};
          inprogress    -> {ok, inprogress};
          _Unexpected   -> {error, internal_err}
        end;

      lookup_fail  -> {error, label_unknown};
```

```
      InternalErr  -> InternalErr
    end,
  {reply, Reply, State};

handle_call(stop, From, #{configs := Configs} = State) ->
  gen_server:reply(From, {ok, lists:map(fun ({Tag, Config}) ->
                                            {Tag, maps:to_list(Config)}
                                       end, maps:to_list(Configs))}),
  {stop, {shutdown, normal}, State};

handle_call(_Msg, _From, State) ->
  {reply, {error, unrecognized_call}, State}.


-spec handle_cast(term(), state()) -> {noreply, state()}.
handle_cast({add_default, Label, Filter, InitData},
            #{defaults := Defaults} = State) ->

  case safeMapsFind(Label, Defaults) of

    lookup_fail -> % label does not exist? good, let's add it!
      Defaults2 = Defaults#{Label => {Filter,
                                      InitData,
                                      filter_count(Filter)}},
      {noreply, State#{defaults => Defaults2}};

    _ -> {noreply, State}
  end;


handle_cast({enough, MailTag}, #{coords   := Coords,
                                 current  := Current,
                                 defaults := Defaults} = State) ->

  case safeMapsFind(MailTag, Coords) of
    {ok, Coord} ->
      gen_server:stop(Coord),
      case safeMapsRemove(MailTag, Coords) of
        {ok, Coords2} ->

          NumFiltsRem = default_filters_count(Defaults), % remember to free up capacity !
          {noreply, State#{coords => Coords2,
                           current => Current - NumFiltsRem - 1}};

        _ -> {noreply, State} % should never match since we just looked up MailTag.

      end;
    _MailUnknown -> {noreply, State} % no coordinator for mail? consider it stopped already.
  end;

handle_cast({add_filter, MailTag, Label, Filter, InitData},
            #{coords   := Coords,
              capacity := Cap,
              current  := Current} = State) ->

  NumFilters = filter_count(Filter),
  RoomForFilter = (Current + NumFilters) =< Cap,

  FilterExists = stateFilterExists(State, MailTag, Label),
  DoAddFilter = RoomForFilter and (not FilterExists),

  if DoAddFilter ->
       case safeMapsFind(MailTag, Coords) of
         {ok, Coord} ->

```

```
          gen_server:cast(Coord, {add_filter, Label, Filter, InitData}),

          State2 = stateUpdateConfig(State, MailTag, Label, inprogress),

          Current2 = Current + NumFilters,

          {noreply, State2#{current => Current2}};

        _MailUnknown  -> {noreply, State}
      end;
    not DoAddFilter -> {noreply, State}
  end.


handle_cast({update_config, MailTag, New}, State) ->
  {noreply, stateSetConfig(State, MailTag, New)};

handle_cast({worker_shutdown, MailTag, Label},
            #{configs := Configs} = State) ->

  case safeMapsFind(MailTag, Configs) of
    {ok, Config} ->
      case safeMapsRemove(Label, Config) of
        {ok, Config2} ->
          Configs2 = Configs#{MailTag => Config2},
          {noreply, State#{configs => Configs2}};
        _ -> {noreply, State}
      end;
    _ -> {noreply, State}
  end;

handle_cast(_, State) -> {noreply, State}.


-spec init(integer()) -> {ok, state()}.
init(Capacity) ->
  InitState = #{current  => 0,
                capacity => Capacity,
                coords   => #{},
                configs  => #{},
                defaults => #{}
               },
  {ok, InitState}.

% TODO: cleanup necessary at this point..?
terminate(_Reason, _State) -> ok.

%%%%%%%%%%%%%%%
%%% HELPERS %%%
%%%%%%%%%%%%%%%
add_coord(MailTag, Coord, #{coords := Coords} = State) ->
  State#{coords => Coords#{MailTag => Coord}}.


stateSetConfig(#{configs := Configs} = State,
               MailTag, New) ->
  State#{configs => Configs#{MailTag => New}}.

stateUpdateConfig(#{configs := Configs} = State,
                  MailTag, Label, New) ->
  case safeMapsFind(MailTag, Configs) of
    {ok, Config} ->

      Config2 = Config#{Label => New},
      State#{configs => Configs#{MailTag => Config2}};
```

```
    _ -> State
  end.


stateFilterExists(#{configs := Configs}, MailTag, Label) ->
  try safeMapsFind(MailTag, Configs) of
    {ok, Config} -> maps:is_key(Label, Config);
    _ -> false
  catch
    error:_ -> false % TODO: should probably return this explicitly.
  end.


filter_count({simple, _}) ->
  1;
filter_count({chain, Filters}) ->
  filter_list_count(Filters);

filter_count({group, Filters, _}) ->
  filter_list_count(Filters);

filter_count({timelimit, _, Filter}) ->
  filter_count(Filter).

filter_list_count(Filters) ->
  lists:sum(lists:map(fun (Filter) -> filter_count(Filter)
                      end, Filters)).

default_filters_count(Defaults) ->
  lists:sum(lists:map(fun ({_Filter, _Data, Count}) -> Count
                      end, maps:values(Defaults))).

%%%%%%%%%%%%%%%%%%%%
%%% MISC HELPERS %%%
%%%%%%%%%%%%%%%%%%%%
safeMapsFind(Key, Map) ->
  try maps:find(Key, Map) of
    {ok, Val} -> {ok, Val};
    _         -> lookup_fail
  catch
    error:Exception -> io:format(">> safeMapsFind() - caught: ~p~n~n",
                                 [Exception]),
                       {error, {internal_error, Exception}}
  end.

safeMapsRemove(Key, Map) ->
  try maps:remove(Key, Map) of
    Map2 -> {ok, Map2}
  catch
    _:_ -> {error, internal_error}
  end.
```

B.2 code/mailfilter/src/coordinator.erl

```
-module(coordinator).

-behaviour(gen_server).

% gen_server mandated exports.
-export([ start/4
        , init/1
        , terminate/2
        , handle_cast/2
        , handle_call/3
        ]).


%%%%%%%%%%%%%%%%%%%%%%%
%%% COORDINATOR API %%%
%%%%%%%%%%%%%%%%%%%%%%%
start(Mail, MailTag, Defaults, MailServer) ->
  gen_server:start_link(?MODULE, {Mail, MailTag, Defaults, MailServer}, []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% COORDINATOR INTERNALS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-type mail_tag() :: mailfilter:mail_tag().
-type mail()     :: mailfilter:mail().
-type label()    :: mailfilter:label().
-type data()     :: mailfilter:data().

-type filter()           :: mailfilter:filter().

-type flag()    :: valid | invalid.
-type flags()   :: #{label() := flag()}.
-type workers() :: #{label() := pid()}.
-type config()  :: mailfilter:config().


-type coord_state() ::
  #{orig    := mail(), % original mail.
    mail    := mail(), % current state of mail.

    workers := workers(), % workers, flags, and config map filter
    flags   := flags(),   % labels to worker servers, valid flags, and
    config  := config(),  % last config for these filters, respectively.

    parent  := pid(),     % mailfilter server whom this coord reports back to.
    tag     := mail_tag() % identifies this coord to the mailfilter server.
   }.


state_update_result(#{config := Config} = State, Label, New) ->
  State#{config => Config#{Label => New}}.


state_update_flag(#{flags := Flags} = State, Label, New) ->
  State#{flags => Flags#{Label => New}}.


invalidate_all_but(Label, #{workers := Workers,
                            flags   := Flags,
                            mail    := NewMail} = State) ->
```

```
  case safeMapsRemove(Label, Workers) of
    {ok, Workers2} ->
      maps:map(fun (_Label, Pid) ->
                   gen_server:cast(Pid, {invalidate, NewMail})
               end, Workers2),

      State#{flags => (maps:map(fun (_Label, _Flag) ->
                    invalid
                end, Flags))#{Label => valid}};
    _ -> State
  end.


handle_cast({worker_shutdown, Label},
            #{parent := Parent,
              tag    := MailTag} = State) ->
  gen_server:cast(Parent, {worker_shutdown, MailTag, Label}),
  {noreply, State};


handle_cast({update_config, Label, FilterResult},
            #{parent := Parent,
              tag    := MailTag} = State) ->
  State3 = #{config := Config2} =
    case FilterResult of

      {transformed, Mail} ->
        invalidate_all_but(Label, State#{mail => Mail});

      {both, Mail, Data} ->
        State2 = state_update_result(State, Label, {done, Data}),
        invalidate_all_but(Label, State2#{mail => Mail});

      {_JustOrUnchanged, Data} ->
        State2 = state_update_result(State, Label, {done, Data}),
        state_update_flag(State2, Label, valid);

      _Unexpected -> State
    end,

  case all_valid(State3) of
    true -> % notify parent of the new Config for this mail.
      gen_server:cast(Parent, {update_config, MailTag, Config2});

    _ -> nop % some workers need to recompute their filters first.
  end,

  {noreply, State3};


handle_cast({add_filter, Label, Filter, InitData},
            #{mail    := Mail,
              workers := Workers} = State) ->

  case spawn_worker(Mail, Label, Filter, InitData) of
    {ok, Worker} ->
      Workers2 = Workers#{Label => Worker},
      State2 = #{flags := Flags} = invalidate_all_but(Label, State#{mail => Mail}),
      Flags2 = Flags#{Label => invalid},
      {noreply, State2#{workers => Workers2, flags => Flags2}};

    _SpawnError -> {noreply, State}
  end.

handle_call(_, _, State) ->
```

```
  {reply, {error, call_unexpected}, State}.


-spec init({mail(), mail_tag(), #{label() := {filter(), data()}}, pid()})
  -> {ok, coord_state()}.
init({Mail, MailTag, Defaults, MailServer}) ->

  Workers =  spawn_default_filters(Mail, Defaults),
  Flags   =  init_flags(Defaults),
  Config  =  init_config(Defaults),

  InitState = #{orig => Mail,
                mail => Mail,

                workers => Workers,
                flags   => Flags,
                config  => Config,

                parent  => MailServer,
                tag     => MailTag
               },

  {ok, InitState}.

init_flags(Defaults) ->
  maps:map(fun(_, _) -> invalid end, Defaults).

init_config(Defaults) ->
  maps:map(fun(_Key, {_Filter, InitData, _}) -> InitData end, Defaults).


% TODO: cleanup necessary at this point?
terminate(_Reason, _State) -> ok.


%%%%%%%%%%%%%%%
%%% HELPERS %%%
%%%%%%%%%%%%%%%
spawn_worker(Mail, Label, Filter, InitData) ->
  Me = self(),
  case worker:start(Mail, Label, Filter, InitData, Me)  of
    {ok, Worker}    -> {ok, Worker};
    {error, Reason} -> {error, Reason};
    _               -> {error, worker_spawn_error}
  end.


-spec spawn_default_filters(mail(), #{label() := {filter(), data(), integer()}})
        -> workers().
spawn_default_filters(Mail, Defaults) ->

  % PidsOrErrors is a map of type #{label() := (pid() | Error)}
  PidsOrErrors =
    maps:map(fun (Label, {Filter, InitData, _}) ->
                 case spawn_worker(Mail, Label, Filter, InitData) of
                 {ok, Coord} -> Coord;
                 _           -> error
                 end
             end, Defaults),

  % filter out any errors.
  maps:from_list(
    lists:filter(fun ({_Label, PidOrError}) ->
                       PidOrError =/= error
                 end, maps:to_list(PidsOrErrors))).
```

```

all_valid(#{flags := Flags}) ->
  lists:all(fun (Flag) -> Flag == valid end, maps:values(Flags)).

%%%%%%%%%%%%%%%%%%%%
%%% MISC HELPERS %%%
%%%%%%%%%%%%%%%%%%%%
safeMapsRemove(Key, Map) -> mailfilter:safeMapsRemove(Key, Map).
```

B.3 code/mailfilter/src/worker.erl

```erlang
-module(worker).

-behaviour(gen_server).

% gen_server mandated exports.
-export([ start/5
        , init/1
        , terminate/2
        , handle_cast/2
        , handle_call/3
        , handle_continue/2
        ]).


%%%%%%%%%%%%%%%%%%%%%%
%%% MAILWORKER API %%%
%%%%%%%%%%%%%%%%%%%%%%
% start(Mail, Tag, Coordinator) ->
start(Mail, Label, Filter, InitData, Coordinator) ->
  gen_server:start_link(
    ?MODULE, {Mail, Label, Filter, InitData, Coordinator}, []).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% MAILWORKER INTERNALS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-type mail()   :: mailfilter:mail().
-type label()  :: mailfilter:label().
-type data()   :: mailfilter:data().
-type filter() :: mailfilter:filter().
-type filter_result() :: mailfilter:filter_result().

-type worker_state() ::
  #{mail   := mail(),   % current state of mail.
    label  := label(),  % filter label.
    filter := filter(), % filter.
    data   := data(),   % current filter data.
    parent := pid()     % parent coord whom this worker reports back to.
   }.



-spec init({mail(), label(), filter(), data(), pid()})
  -> {ok, worker_state()}.
init({Mail, Label, Filter, InitData, Coordinator}) ->

  InitState = #{mail   => Mail,
                label  => Label,
                filter => Filter,
                data   => InitData,
                parent => Coordinator
               },
  {ok, InitState, {continue, do_filter}}.

handle_cast({invalidate, NewMail}, State) ->
  {noreply, State#{mail => NewMail}, {continue, do_filter}};

handle_cast(_, State) ->
  {noreply, State}.
```

```erlang

handle_continue(do_filter, #{label  := Label,
                             mail   := Mail,
                             filter := Filter,
                             data   := Data,
                             parent := Parent} = State) ->
  try
    run_filter(Filter, Mail, Data)
  of
    FilterResult ->
      gen_server:cast(Parent, {update_config, Label, FilterResult}),
      {noreply, State}
  catch
    _:_ -> timer:sleep(1), % TODO: figure out why this sleep is necessary
                           %       when it obviously shouldn't be.
           {stop, {shutdown, normal}, State}
  end.

% ignore calls.
handle_call(_, _, State) ->
  {reply, {error, call_unexpected}, State}.

terminate(_Reason, #{parent := Parent, label := Label}) ->
  gen_server:cast(Parent, {shutdown, Label}).


%%%%%%%%%%%%%%%%%%%%%%%%%
%%% FILTER EVALUATION %%%
%%%%%%%%%%%%%%%%%%%%%%%%%
-spec run_filter(filter(), mail(), data()) -> filter_result().
run_filter({simple, FilterFun}, Mail, Data) ->
    FilterFun(Mail, Data);

run_filter({chain, Filters}, Mail0, Data0) ->
  InitAcc = {unchanged, Mail0, Data0},

  {FilterResult, _, DataResult} =
    lists:foldr(
      fun (Filter, {FilterResultAcc, MailAcc, DataAcc}) ->
        FilterResultAcc2 = combine_filter_results(FilterResultAcc,
                             run_filter(Filter, MailAcc, DataAcc)),

        {MailAcc2, DataAcc2} = update_accs(FilterResultAcc2, MailAcc, DataAcc),
        {FilterResultAcc2, MailAcc2, DataAcc2}
      end, InitAcc, Filters),

  case FilterResult of
    unchanged -> {unchanged, DataResult};
    _         -> FilterResult
  end;


% TODO: for the moment, this is not very parallel (or:
%       parallel with a very small degree of parallelism).
run_filter({group, Filters, MergeFun}, Mail0, Data0) ->
  FilterResults = lists:map(fun (Filter) ->
                              run_filter(Filter, Mail0, Data0)
                            end, Filters),
  MergeFun(FilterResults);


% TODO: not implemented! but I let it return unchanged for testing purposes.
run_filter({timelimit, _TimeOut, _Filter}, _Mail, _Data) ->
  unchanged;

```

```
run_filter(_, _, _) -> unchanged.


%%%%%%%%%%%%%%%
%%% HELPERS %%%
%%%%%%%%%%%%%%%
update_accs(FiltRes, Mail0, Data0) ->
  case FiltRes of
    {just, Data}        -> {Mail0, Data };
    {transformed, Mail} -> {Mail,  Data0};
    {both, Mail, Data}  -> {Mail,  Data };
    _Unchanged          -> {Mail0, Data0}
  end.

combine_filter_results(Acc1, Acc2) ->
  case {Acc1, Acc2} of
    {unchanged, _} -> Acc2;
    {_, unchanged} -> Acc1;

    {_,                {both, _, _}}     -> Acc2;
    {{just, _},        {just, _}}        -> Acc2;
    {{transformed, _}, {transformed, _}} -> Acc2;

    {{just, Data}, {transformed, Mail}}    -> {both, Mail, Data};
    {{transformed, Mail}, {just, Data}}    -> {both, Mail, Data};
    {{both, Mail, _}, {just, Data}}        -> {both, Mail, Data};
    {{both, _, Data}, {transformed, Mail}} -> {both, Mail, Data};

    _ -> throw(missing_combine_case)  % this should never match, but even if it did, it
                                      % would be preferable to discover it immediately,
                                      % ie. from a nasty exception.
  end.
```

B.4 code/mailfilter/src/test$_{m}ailfilter.erl$

```erlang
-module(test_mailfilter).

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

-export([test_all/0, test_everything/0]).
-export([test_mailfilter/0]). % Remember to export the other function from Q2.2


test_everything() ->
  test_all().

test_all() ->
  test_mailfilter().

test_mailfilter() ->
  Capacity = 17,

  {ok, MS} = mailfilter:start(Capacity),

  Mail1 = 5,
  Mail2 = 19,

  % adding two default filters, which together take up 14 capacity.
  mailfilter:default(MS, foo, nested_chain_filter(), 7),
  % mailfilter:default(MS, bar, nested_chain_filter(), 5),

  {ok, MR1} = mailfilter:add_mail(MS, Mail1),
  {ok, MR2} = mailfilter:add_mail(MS, Mail2),

  % this filter will ask for 7 capacity, and will thus be rejected by the mail server.
  mailfilter:add_filter(MR2, bar, nested_chain_filter(), 4),

  timer:sleep(50),

  {ok, Conf1} = mailfilter:get_config(MR1),
  {ok, Conf2} = mailfilter:get_config(MR2),
  ?assertEqual(lists:flatlength(Conf1), 1),
  ?assertEqual(lists:flatlength(Conf2), 1),

  mailfilter:enough(MR1), % kill filter for MR1, freeing up 7 capacity.

  timer:sleep(50),

  % there should now be room for the new chain filter of size 7.
  mailfilter:add_filter(MR2, baz, nested_chain_filter(), 2),

  timer:sleep(50),

  {ok, Conf3} = mailfilter:get_config(MR2),

  % if the config for MR2 is now 2 elements long,
  ?assertEqual(lists:flatlength(Conf3), 2),

  mailfilter:stop(MS).


decrement(Mail, MyCount) ->
  if Mail =< 0 -> {just, MyCount};
     Mail >  0 -> {both, Mail - 1, MyCount + 1}
  end.
```

```erlang

simple_filter() -> {simple, fun decrement/2}.

chain_filter() ->
  {chain, [simple_filter(), simple_filter(), simple_filter()]}.

nested_chain_filter() ->
  {chain, [simple_filter(), chain_filter(), chain_filter()]}.
```