Advanced Programming - Exam

Exam Number 37

November 12, 2021

Contents

Question 1: An automatic programming system

1.1 - Parser

Completeness

For this module I implemented the full API with both parseStringType and parseStringTDeclz using the Parsec library. Known bugs are mentioned in the Correctness section.

My transformed grammar from entry point `Type` can be seen in Figure 1 on the next page (screenshotted from the comments in my implementation file). Note that for the `Type` nonterminal I renamed it to `pType` in my implementation due to `type` being a haskell keyword.

I first removed the left recursion in the `Type '->' Type` sequence with a new nonterminal `Assign`, followed by left-factorising the two sequences starting with `'('` with a new nonterminal `TParen`. At this point, the only disambiguation issue was that infix arrows had higher precedence over type constructors. To fix this I right-factorised out a new nonterminal `Type'` (since every sequence in `Type` ended in `Assign`). This allowed me to use `Type'` instead of `Type` in the `Typez` nonterminal, removing the possibility of an infix arrow before the end of the constructor.

The transformed grammar for entrypoint `TDeclz` can be seen in Figure 2 on the next page. To remove the left recursion in `TDeclz` I just swapped the nonterminals in the sequences. I also left factorised `FDecls` and `Fields`. Otherwise the grammar remained the same.

Correctness

Through my testing I found two deficiencies. The first is that comments that do not close are false-positively parsed, with everything after the opening of the comment ignored.

The second is that illegal characters in names causes the parser to stop consuming anything that comes after, for example `"a@b -> a"` evaluates to `PTVar "a"`. I use `many isName` to check names have the correct characters, but once an illegal character is found it returns the name up to that point instead of failing. I could not find a combinator to fix this, so I instead tried `manyTill isName endOfName`, where `endOfName` checks the next character(s) pertain to a well-formed input string. I could not get it to work, but I left my attempted implementation commented out in my code.

I did my best to test all aspects of my implementation. I split my tests into test groups: Names, PType, TDeclz, Whitespace and Disambiguation. To more thoroughly test (and debug) I could have used WhiteBox to test specific parsers, for example for my name and whitespace parsers. With the tests I identified the two above issues. There is one newtype parsing test that fails, but that is due to the naming issue.

```
Type   ::= Type' Assign

Type'  ::= TVar
         | '(' TParen ')'
         | TCon Typez

Assign ::= epsilon
         | "->" Type

TParen ::= Type TParen'
TParen'::= epsilon
         | ',' Type

Typez  ::= epsilon
         | Type' Typez
```

Figure 1: Transformed grammar for Type entrypoint

```
TDecl  ::= 'type' TDHead '=' Type
         | 'newtype' TDHead '=' RCon '{' Field "::" Type '}'
         | 'data' TDHead '=' RCon '{' FDeclz '}'

TDHead ::= TCon TVarz

TVarz  ::= epsilon
         | TVar TVarz

FDeclz ::= epsilon
         | FDecls

FDecls ::= FDecl FDecls'
FDecls'::= epsilon
         | ',' FDecls

FDecl  ::= Fields "::" Type

Fields ::= Field Fields'
Fields'::= epsilon
         | ',' Fields
```

Figure 2: Transformed grammar for TDeclz entrypoint

Robustness

There is not much room for breaking the parser through the API. The input has to be a string, and if a string is poorly formed it simply returns the error message generated from Parsec.

Efficiency

With Parsec I hopefully have a more efficient implementation than if I used ReadP, due to Parsec's committing to a branch after the first character is consumed. I only use `try` once in the `vName` parser to check for keywords.

Maintainability

With the grammar in comments at the top of the file I argue that it should be simple to follow the implementation. I also split my code into labelled sections.

1.2 - Resolver

Completeness

For this module I fully implemented `resolve`, and made a (far from complete) start at `declare`. For `resolve`, each `PTVar` has the type-variable environment function applied to it, and each `PTApp` has its constructor checked in the type-constructor environment with its list of parser types resolved recursively. This function ensures type-constructors exist in the environment, ensures that they are semantically well formed (even though the parser will do this already for `(,)` and `(->)`, and will also fail if the type-variable environment produces an error message.

Correctness

I did not find a way to resolve a `PType` with a custom type-constructor, as `SType` only seems to handle record constructors (as well as `STProd` and `STArrow`) so any `PType` with such will fail to resolve no matter the type-constructor environment. An example of such a `PType` is `F a`. Otherwise, there are no known deficiencies.

I took a test-driven approach, so both functions are fully tested despite `declare` not working. For `resolve` I made sure to test how poorly-formed environments are handled, and for `declare` I tested all the semantics of forming declarations. I used a helper function `testDeclare` to test declare in the way the example test case did. I also added a test case that resolves a type using the output of the declaration function, namely the example from page 8 of the specification

Efficiency

Unless there is a more efficient implementation for `lookup` or `mapM`, I believe I implemented `resolve` in the most efficient way possible.

Robustness

Resolve can only take a correctly-formed `PType`. Any used poorly-formed function in a type-constructor or type-variable environment will get caught by the function and produce the appropriate error message. A poorly-formed type-constructor environment can get through, for example one with multiple bindings for the same constructor, but in this case the prelude `lookup` function just finds the first in the list. Therefore, there is little room for breaking the resolver with `resolve`.

Maintainability

My attempt at `declare` is a mess as it is unfinished, but resolve is quite succinct and maintainable.

1.3 - Coder

Completeness

For this module I implemented the `pick` and `solution` functions. Due to time constraints I did not have time to tackle `produce`, nor did I have time to write tests for it for test-driven development.

In `solution`, for the base cases of a success node or failure node I return those as a list appropriately. When there is a choice, `Choice (a:as)`, I implement breadth-first by recursively finding the solution to the tail of the list, `solutions (Choice as)`, and then appending the solution of the head, `solutions a`, to the end of the list.

In the specifications it says `n` is non-negative; I interpreted this to mean that it cannot be 0. In the implementation, whenever there is a success node `Found x`, I check `n == 1`, and if false I do the next recursive call with `n-1`. When generating the solutions of the tail of the list, `solutions (Choice as)`, I check `length >= n`, and if false I do the next recursive call with `n-length`.

Correctness

From the testing I found no deficiencies in what I implemented. For pick I tested lists of different sizes, as well as created more tests based off the example one in the suite to ensure I correctly implemented bind in my monad instance.

With `solution` I made sure to have the breadth-first search working correctly as that is a key characteristic of the function. I did this by testing a number of medium-sized trees that would have a different output if doing depth-first searching. I also made sure my implementation of my parameter `n` is correct.

Ideally I would have written tests for produce as well despite not implementing it, as well as written tests for using the three functions together.

Efficiency

In my implementation I have not consciously done anything to improve efficiency.

Robustness

As mentioned in the specification, I assume `n` in the `solution` to be non-negative. For my implementation it also has to be at least `1`, otherwise `n` will keep decreasing and the equality `n == 1` will never be met. This means the whole solution list gets generated, and in the case of a recursive tree it will never end. Otherwise, there is not much room for breaking my coder with the two implemented functions.

Maintainability

There perhaps there is a better way of implementing solution, but I think it is still quite clear and succinct.

Autoprog Testing

I only used blackbox testing; the whitebox is left as it was initially. With more time I would have added tests that combine modules. I did split the tests into clear test groups, but the variables defined in the `where` clause are messy which hinders maintainability a bit. The results of running `stack test` is saved as a `.txt` file in the appendix. Note that tests were written for some functions that were not implemented (namely `declare`), and some tests false-positively succeed because they are intended to fail, but fail for the wrong reason. In total 17 out of 101 tests failed.

Question 2: A Cache of Fun

2.1 - Frappe

Completeness

Here I implemented every API function except `stable/2`. The `frappe.erl` module contains only the entry-point to these functions, with each making a call to a `server.erl` module that uses `gen_server`. The state of this frappe server is `[Capacity, [Key, Value, Cost] = Queue]`, where `Capacity` is the capacity of the cache given in `fresh/1`, and `Queue` is a list of items. I originally tried using the standard queue module, but was met with some unexpected behaviour when turning it into a list to use `keymember/3` and `keyfind/3` to get items.

To pop LRU items from the queue I use `{_, PoppedQueue} = lists:split(1, Queue)`. I handle popping only in my `insert_item/2` and `update_item/2` helper functions, which are called in `set/4`, `insert/4`, `update/4` and `upsert/3`. Because of this, `upsert/3` only counts as usage if a key is successfully updated. To reorder an item in the queue I use `lists:keydelete(Key, 1, Queue) ++ [Item]`, where `{Key, Val, C} = Item`. This is done in the aforementioned `update_item/2`, in `read/2`, and of course would also be in `stable/3` if I were to implement it.

Correctness

The reason for not implementing `stable/3` was because my write operations were implemented synchronously, and I did not have time to fix both. Therefore key coherency is met, but not key concurrency. To implement this concurrency I would use a `gen_statem` to track the state of write operations.

Besides these unimplemented details I am not aware of any deficiencies in my implementation. Along with `test_frappe.erl` I also used a `test_eunit.erl` module for test-driven development with EUnit, where I extensively tested every detail I could find in the specification, with all 44 unit tests succeeding (see appendix for results to `test_all/0`). I did not have time to unit test key concurrency or `stable/3`, but if I were to attempt those functions I would continue with test-driven development and write the tests first.

Efficiency

As I am writing this I realise I did not write tests on larger caches, so I am not entirely sure how efficiently one might perform.

Robustness

Since type-checking is not possible in Erlang, passing a parameter that raises an unintended error is entirely possible. With the assumption that the types are correct, I believe my implementation is quite robust. The cache's capacity

is checked to be a positive integer before starting, keys and values can be any term, and transformation functions that fail are handled in a try/catch block.

Maintainability

I argue my code is quite maintainable. I name functions/variables in a readable manner, add a few comments for any non-obvious code, separate my concerns, label my sections in code, and place reused code into helper functions. I would not feel daunted having to go back and change my implementation to be key concurrent, for example.

2.2 - Frappe QuickCheck

Completeness

This module is quite messy. I unfortunately could not get my cache generators to work, so although I "implemented" some properties I could not get far with them since I was roadblocked. I did implement `mktrans/2` and `terminating_transformation/1`, but since I could not use them in the properties I am unsure if they work.

For inspiration I looked at the `dict` example module, as well as followed the *Slow-paced walk-through of using QuickCheck for testing Emoji* videos on absalon. The `dict` example is where I got inspiration for `prop_duplicate_keys`, which I ended up implementing in the same way (I would have fixed the `lists:usort/1` issue once I came to it, but I was stuck with generator). I also wanted to implement `prop_stopped_server`, which tests that any stopped server can no longer receive calls.

I first tried to implement the generator similar to in the `dict` example, using `?LAZY` and symbolic calls, but unfortunately a cache cannot be recursively created in the same way. I instead needed to start the cache and then pass the `FS` to make a number of symbolic calls. This is doable using apqc, since frappe is a stateful interface, but unfortunately attempting a generator this way was giving me cryptic error messages, namely `{'EXIT',{nodedown,none}}`. I left all attempted implementations in the submission.

For generating transformation functions with `mktrans`, the `Opr` argument is given as `oneof([new, add_val, set_cap, val_is_cap, throw])`. These functions have a mix of outputs, with `new` inserting a new key and the rest updating existing keys. As the `Args` I just passed a generated `value()`, given as `frequency([{4, int()}, {1, char()}])`. I want transformation functions to occasionally have errors, but not too often, so I used frequency to minimise the chances. Some example errors would be adding a character to an integer, or setting the capacity as a character. Since the type of a value does not matter to the implementation I thought just having integers and characters would be plenty.

Correctness

The properties do not work. `prop_cache_under_capacity` I left undefined, as well as my `prop_stopped_server` property. `prop_capacity_invariant` and `prop_unique_keys` use my `?LAZY` implementation of the cache generator, and they simply output "`OK, passed 100 tests`" when tested.

As previously mentioned, I suspect `mktrans/2` and `terminating_transformation/1` may work, but since I could not use them in properties I am unsure.

Maintainability

The code is in an unfinished state, with many functions not working, but I put it in sections so that it is clear what everything is intended to do.

Frappe Testing

I opted to use both QuickCheck and EUnit testing for this question. The `test_all/0` function in `test_frappe.erl` includes the output of my properties, as well as my EUnit tests from `test_eunit.erl`, the output of which is included in a `.txt` file in the appendices.

Appendix

Parser

```
module ParserImpl where

import Defs

import Text.ParserCombinators.Parsec
import Data.Char

-- GRAMMAR (After transformation)

-- Type    ::= Type' Assign

-- Type'   ::= TVar
--           | '(' TParen ')'
--           | TCon Typez

-- Assign  ::= epsilon
--           | "->" Type

-- TParen  ::= Type TParen'
-- TParen' ::= epsilon
--           | ',' Type

-- Typez   ::= epsilon
--           | Type' Typez


-- TDeclz  ::= epsilon
--           | TDelc TDeclz
--           | ';' TDeclz

-- TDecl   ::= 'type' TDHead '=' Type
--           | 'newtype' TDHead '=' RCon '{' Field "::" Type '}'
--           | 'data' TDHead '=' RCon '{' FDeclz '}'

-- TDHead  ::= TCon TVarz

-- TVarz   ::= epsilon
--           | TVar TVarz

-- FDeclz  ::= epsilon
--           | FDecls

-- FDecls  ::= FDecl FDecls'
-- FDecls' ::= epsilon
--           | ',' FDecls

-- FDecl   ::= Fields "::" Type

-- Fields  ::= Field Fields'
-- Fields' ::= epsilon
--           | ',' Fields

-- TCon    ::= cName
```

```haskell
-- TVar   ::= vName
-- RCon   ::= cName
-- Field  ::= vName

parseStringType :: String -> Either ErrMsg PType
parseStringType input =
  case parse (do whitespace; pt <- pType; return pt) "parse error" input of
    Left err -> Left $ show err
    Right ptype -> Right ptype

parseStringTDeclz :: String -> EM [TDecl]
parseStringTDeclz input =
  case parse (do whitespace; pt <- tDeclz; return pt) "parse error" input of
    Left err -> Left $ show err
    Right tDeclz -> Right tDeclz

-- PType parsers
pType :: Parser PType
pType = do t <- pType'; t' <- assign t; return t'

pType' :: Parser PType
pType' = do v <- tVar; return $ PTVar v
    <|> do t <- between (symbol "(") (symbol ")") pTypeParen;
    return t
    <|> do c <- tCon; t <- pTypez; return $ PTApp c t;

assign :: PType -> Parser PType
assign t1 = do symbol "->"; t2 <- pType; return $ PTApp "(->)" [t1, t2]
    <|> return t1

pTypeParen :: Parser PType
pTypeParen = do t <- pType; t' <- pTypeParen' t; return t'

pTypeParen' :: PType -> Parser PType
pTypeParen' t1 = do symbol ","; t2 <- pType; return $ PTApp "(,)" [t1, t2]
    <|> return t1

pTypez :: Parser [PType]
pTypez = do t <- pType'; ts <- pTypez; return (t:ts)
    <|> return []

-- TDeclz parsers
tDeclz :: Parser [TDecl]
tDeclz = do eof; return []
    <|> do td <- tDecl; tds <- tDeclz; return (td:tds)
    <|> do symbol ";"; tds <- tDeclz; return tds;

tDecl :: Parser TDecl
tDecl = do symbol "type"; th <- tDHead; symbol "="; pt <- pType;
    return $ TDSyn th pt
    <|> do symbol "newtype"; th <- tDHead; symbol "="; rc <- rCon;
    symbol "{"; f <- field; symbol "::"; pt <- pType; symbol "}";
    return $ TDRcd th rc [(f, pt)]
```

```
    <|> do symbol "data"; th <- tDHead; symbol "="; rc <- rCon;
    symbol "{"; fdz <- fDeclz; symbol "}"; return $ TDRcd th rc fdz

tDHead :: Parser TDHead
tDHead = do c <- tCon; vz <- tVarz; return (c, vz)

tVarz :: Parser [TVName]
tVarz = do v <- tVar; vz <- tVarz; return (v:vz)
    <|> return []

fDeclz :: Parser [(FName, PType)]
fDeclz = do fds <- fDecls; return fds
    <|> return []

fDecls :: Parser [(FName, PType)]
fDecls = do fd <- fDecl; fds <- fDecls' fd; return fds

fDecls' :: [(FName, PType)] -> Parser [(FName, PType)]
fDecls' fd = do symbol ","; fds <- fDecls; return $ fd ++ fds
    <|> return fd

fDecl :: Parser [(FName, PType)]
fDecl = do fs <- fields; symbol "::"; pt <- pType; return [(f, pt)
    | f <- fs]

fields :: Parser [FName]
fields = do f <- field; fs' <- fields' f; return fs'

fields' :: FName -> Parser [FName]
fields' f = do symbol ","; fs <- fields; return (f:fs)
    <|> return [f]

-- Terminals
tVar :: Parser TVName
tVar = vName

tCon :: Parser TCName
tCon = cName

rCon :: Parser RCName
rCon = cName

field :: Parser FName
field = vName

-- TODO: names with illegal characters not working
vName :: Parser TVName
vName = lexeme $ do try (do reserved; notFollowedBy isName); fail "
    reserved keyword"
    <|> do h <- satisfy isLower; t <- many isName; return $ [h] ++
    t

-- cName starts with uppercase, hence no checking for keywords
cName :: Parser TCName
cName = lexeme $ do h <- satisfy isUpper; t <- many isName; return
    $ [h] ++ t

```

```haskell
-- Lexeme
whitespace :: Parser ()
whitespace = do string "{-"; manyTill anyToken (string "-}");
    return ()
    <|> do spaces; return ()

lexeme :: Parser a -> Parser a
lexeme p = do a <- p; whitespace; return a

symbol :: String -> Parser ()
symbol s = lexeme $ do string s; return ()

-- Helper parsers
reserved :: Parser String
reserved = do string "type"
    <|> do string "newtype"
    <|> do string "data"

isName :: Parser Char
isName = satisfy (\c -> isAlphaNum c || c == '_' || c == '\'')

-- ATTEMPTED FIX TO ILLEGAL CHARACTERS
-- usage in vName/cName: manyTill isName endOfName

-- endOfName :: Parser ()
-- endOfName = try (do string "/"; return ())
-- endOfName = try (do string "->"; anyToken; return ())
--     <|> try (do char ')'; return ())
--     <|> try (do char ','; anyToken; return ())
--     <|> try (do spaces; return ())
--     <|> try (do eof; return ())
```

Listing 1: ParserImpl.hs

Resolver

```haskell
module ResolverImpl where

import Defs

resolve :: TCEnv -> (TVName -> EM SType) -> PType -> EM SType
resolve tce tve pt =
  case pt of
    PTVar x -> tve x
    PTApp c x ->
      case lookup c tce of
        Nothing -> Left $ show c ++ " is not defined in type-
    constructor environment"
        Just f -> do
          st <- mapM (resolve tce tve) x
          f st

declare :: [TDecl] -> EM TCEnv
declare [] = Right []
declare ds = do
  decz <- doDeclarations ds
  return $ tce0 ++ decz

doDeclarations :: [TDecl] -> EM TCEnv
doDeclarations [] = Right []
doDeclarations (a: _as) =
  case a of
    TDSyn (_cn, _vs) pt -> do
      -- check vs has distinct variables
      -- check variables in pt are in vs
      case pt of
        PTVar _ -> Left "incomplete"
        PTApp _c _ -> Left "incomplete"
    TDRcd (_cn, _vs) _rc _fs -> Left "incomplete"
```

Listing 2: ResolverImpl.hs

Coder

```haskell
module CoderImpl where

import Defs
import Control.Monad (ap, liftM)

instance Functor Tree where fmap = liftM
instance Applicative Tree where pure = return; (<*>) = ap

instance Monad Tree where
  return = Found
  Found a >>= f = f a
  Choice [] >>= _ = Choice []
  Choice a >>= f = Choice $ [x >>= f | x <- a]

pick :: [a] -> Tree a
pick [] = Choice []
pick [x] = return x
pick as = Choice $ [pick [x] | x <- as]

solutions :: Tree a -> Int -> Maybe a -> [a]
solutions (Found a) _ _ = [a]
solutions (Choice []) _ _ = []
solutions (Choice (a:as)) n d =
  case a of
    Found x ->
      if n == 1
        then [x]
        else [x] ++ solutions (Choice as) (n-1) d
    Choice _ ->
      let l = solutions (Choice as) n d
          len = length l
      in if len >= n
        then case d of
          Nothing -> l
          Just d' -> l ++ [d']
        else l ++ solutions a (n-len) d

produce :: [(String,SType)] -> SType -> Tree Exp
produce = undefined

-- recommended, but not mandated, helper function:
extract :: [(String,SType)] -> SType -> SType -> Tree (Exp -> Exp)
extract = undefined
```

Listing 3: CoderImpl.hs

BlackBox Testing

```haskell
-- Sample black-box test suite. Feel free to adapt, or start from scratch.

-- Do NOT import from your ModImpl files here. These tests should work with
-- any implementation of the AutoProg APIs. Put any white-box tests in
-- suite1/WhiteBox.hs.
import Defs
import Parser
import Resolver
import Coder

import Test.Tasty
import Test.Tasty.HUnit

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

tests = testGroup "All tests" [
  testGroup "Parser" [

    testGroup "Name Tests" [
      testCase "Simple var name" $
        parseStringType "a" @?= Right pt2,
      testCase "Var name with all possible chars" $
        parseStringType "b1_'23" @?= Right (PTVar "b1_'23"),
      testCase "Var name with illegal char" $
        case parseStringType "b/" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
      testCase "Simple con name" $
        parseStringType "A" @?= Right (PTApp "A" []),
      testCase "Con name with all possible chars" $
        parseStringType "B1_'23" @?= Right (PTApp "B1_'23" []),
      testCase "Con name with illegal char" $
        case parseStringType "B/" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
      testCase "Keyword type" $
        case parseStringType "type" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
      testCase "Keyword newtype" $
        case parseStringType "newtype" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
      testCase "Keyword data" $
        case parseStringType "data" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
```

```
    ,
      testCase "Keyword type in middle of name" $
        parseStringType "type24" @?= Right (PTVar "type24"),
      testCase "Keyword newtype in middle of name" $
        parseStringType "newtype24" @?= Right (PTVar "newtype24"),
      testCase "Keyword data in middle of name" $
        parseStringType "data24" @?= Right (PTVar "data24"),
      testCase "pType with all illegal names" $
        case parseStringType "F* x* -> (y*, A*)" of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
      testCase "tDeclz with all illegal names" $
        case parseStringTDeclz "data T* = C* {f*, f* :: a* -> a*}"
    of
          Left _ -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ],

    testGroup "PType tests" [
      testCase "Simple assigning" $
        parseStringType "a->a" @?= Right pt0,
      testCase "Con with var" $
        parseStringType "A a" @?= Right pt1,
      testCase "Simple parentheses" $
        parseStringType "(a)" @?= Right pt2,
      testCase "Simple tuple" $
        parseStringType "(a,b)" @?= Right pt3,
      testCase "Tuple with constructor" $
        parseStringType "(a,B)" @?= Right (PTApp "(,)" [PTVar "a",
    PTApp "B" []]),
      testCase "Con with multiple types" $
        parseStringType "F (X y)" @?= Right (PTApp "F" [PTApp "X" [
    PTVar "y"]]),
      testCase "Complex 1" $
        parseStringType "F x->(y,A)" @?= Right pt4,
      testCase "Complex 2" $
        parseStringType "F (F x y)->(T (y,f5),A b)" @?= Right pt5
    ],

    testGroup "TDeclz tests" [
      testCase "Empty list of declarations" $
        parseStringTDeclz "" @?= Right [],
      testCase "Parse declare type" $
        parseStringTDeclz "type X a=a->a" @?= Right [td0],
      testCase "Parse declare newtype" $
        parseStringTDeclz "newtype T=B{c ::a}" @?= Right [TDRcd ("T
    ", []) "B" [("c", PTVar "a")]],
      testCase "Parse declare data no fields" $
        parseStringTDeclz "data T=C {}" @?= Right [TDRcd ("T", [])
    "C" []],
      testCase "Parse declare data one field" $
        parseStringTDeclz "data T=C {f1::a->a}" @?= Right [TDRcd ("
    T", []) "C" [("f1", pt0)]],
      testCase "Parse data multiple fields" $
        parseStringTDeclz "data T=C {f1, f2::a->a}" @?= Right [
    TDRcd ("T", []) "C" [("f1", pt0), ("f2", pt0)]],
```

```
97        testCase "Parse data duplicate name fields" $
98          parseStringTDeclz "data T=C {f, f::a->a}" @?= Right [TDRcd
    ("T", []) "C" [("f", pt0), ("f", pt0)]],
99        testCase "Prase synonym types" $
100         parseStringTDeclz "type T a b c = a" @?= Right [TDSyn ("T",
     ["a", "b", "c"]) (PTVar "a")],
101       testCase "Prase two declarations" $
102         parseStringTDeclz "type T=a; type T=b;" @?= Right [TDSyn ("
    T", []) (PTVar "a"), TDSyn ("T", []) (PTVar "b")],
103       testCase "Ignore semicolon before declaration" $
104         parseStringTDeclz ";type X a=a->a" @?= Right [td0],
105       testCase "Parse declare with bigger pType" $
106         parseStringTDeclz "type T a = F x->(y,A)" @?= Right [TDSyn
    ("T", ["a"]) pt4]
107   ],
108
109   testGroup "Whitespace Tests" [
110     testCase "Space between assigning" $
111       parseStringType "a -> a" @?= Right pt0,
112     testCase "Tab between assigning" $
113       parseStringType "a\t" @?= Right pt2,
114     testCase "Newline between assigning" $
115       parseStringType "a\n" @?= Right pt2,
116     testCase "A lof of whitespace between assigning" $
117       parseStringType "\na\n     -> \t\t\n  a  \n" @?= Right pt0,
118     testCase "Space at start" $
119       parseStringType " a" @?= Right pt2,
120     testCase "Space at end" $
121       parseStringType "a " @?= Right pt2,
122     testCase "Space between parantheses" $
123       parseStringType " ( a ) " @?= Right pt2,
124     testCase "Space between comma" $
125       parseStringType "(a , b)" @?= Right pt3,
126     testCase "Comment at start" $
127       parseStringType "{-Test-}a" @?= Right pt2,
128     testCase "Comment at end" $
129       parseStringType "a{-Test-}" @?= Right pt2,
130     testCase "Comment in middle" $
131       parseStringType "a{-Test-}-> a" @?= Right pt0,
132     testCase "Comment that does not end" $
133       case parseStringType "a {-" of
134         Left _ -> return ()
135         Right p -> assertFailure $ "Unexpected parse: " ++ show p
    ,
136     testCase "Comment in middle of declaration" $
137       parseStringTDeclz "data T=C {{-TODO: add fields-}}" @?=
    Right [TDRcd ("T", []) "C" []],
138     testCase "Space between everything pType" $
139       parseStringType " F ( F x y ) -> ( T ( y , f5 ) , A b ) " @
    ?= Right pt5,
140     testCase "Space between everything declare type" $
141       parseStringTDeclz "   type T  a = a -> a " @?= Right [td0],
142     testCase "Space between everything declare newtype" $
143       parseStringTDeclz " newtype   T = B { c :: a } " @?= Right
    [TDRcd ("T", []) "B" [("c", PTVar "a")]],
144     testCase "Space between everything declare data" $
145       parseStringTDeclz "  \t data T a t1 d = C \n {  f1 ,  f2 ::
```

```
    a -> a }   " @?= Right [TDRcd ("T", ["a", "t1", "d"]) "C" [("
  f1", pt0), ("f2", pt0)]]
146   ],
147
148   testGroup "Disambiguation Tests" [
149     testCase "Constructor parentheses" $
150       parseStringType "F x y" @?= Right (PTApp "F" [PTVar "x",
  PTVar "y"]),
151     testCase "Constructor tighter than infix" $
152       parseStringType "F x y -> z" @?= Right (PTApp "(->)" [PTApp
   "F" [PTVar "x", PTVar "y"], PTVar "z"]),
153     testCase "Right-associative arrow" $
154       parseStringType "a -> b -> c" @?= Right (PTApp "(->)" [
  PTVar "a", PTApp "(->)" [PTVar "b", PTVar "c"]])
155   ]
156 ],
157 testGroup "Resolver" [
158   testGroup "resolve" [
159     testCase "Test PTVar" $
160       resolve tce0 (\x -> return $ STVar (x++"'")) pt2 @?= Right
  (STVar "a'"),
161     testCase "Test STProd" $
162       resolve tce0 (\x -> return $ STVar (x++"'")) pt3 @?= Right
  (STProd (STVar "a'") (STVar "b'")),
163     testCase "Test STArrow" $
164       resolve tce0 (\x -> return $ STVar (x++"'")) pt0 @?= Right
  (STArrow (STVar "a'") (STVar "a")),
165     testCase "Test Nested" $
166       resolve tce0 (\x -> return $ STVar (x++"'")) pt6 @?= Right
  (STArrow (STArrow (STVar "a'") (STVar "a")) (STProd (STArrow (
  STVar "a'") (STVar "a")) (STVar "a'"))),
167     testCase "Test non-existent constructor" $
168       case resolve tce0 (\x -> return $ STVar (x++"'")) pt1 of
169         Left _ -> return ()
170         Right p -> assertFailure $ "Unexpected resolve: " ++ show
   p,
171     testCase "Test bad args in constructor" $
172       case resolve tce0 (\x -> return $ STVar (x++"'")) (PTApp "
  (,)" [PTVar "x"]) of
173         Left _ -> return ()
174         Right p -> assertFailure $ "Unexpected resolve: " ++ show
   p,
175     testCase "Test bad variable environment" $
176       case resolve tce0 (\_ -> Left "nope") pt2 of
177         Left _ -> return ()
178         Right p -> assertFailure $ "Unexpected resolve: " ++ show
   p,
179     testCase "Example from spec" $
180       resolve tce1 (\x -> return $ STVar x) (PTApp "T" [PTApp "
  (,)" [PTVar "b", PTVar "b"]]) @?= Right st1
181   ],
182   testGroup "Declare" [
183     testCase "Empty declaration list" $
184       case testDeclare [] "X" [STVar "a"] of
185         Left _ -> return ()
186         Right p -> assertFailure $ "Unexpected resolve: " ++ show
   p,
```

```
187       testCase "Declare synonym with var" $
188         testDeclare [td2] "T" [STVar "a'"] @?= Right (STVar "a'"),
189       testCase "Declare synonym for (->)" $
190         testDeclare [td0] "X" [STVar "a'"] @?= Right st0,
191       testCase "Declare synonym for (,)" $
192         testDeclare [td3] "Y" [STVar "a'"] @?= Right (STProd (STVar
     "a'") (STVar "a')),
193       testCase "Declare constructor with two type variables" $
194         testDeclare [td16] "T" [STVar "a'"] @?= Right (STVar "a'"),
195       testCase "Example from specs" $
196         testDeclare [td1] "Z" [STProd (STVar "b") (STVar "b")] @?=
    Right st1,
197       testCase "Four declarations" $
198         testDeclare [td0, td1, td2, td3] "Z" [STProd (STVar "b") (
    STVar "b")] @?= Right st1,
199       testGroup "Semantics" [
200         testCase "Non-distinct type constructor (fail)" $
201           case testDeclare [td0, td0] "X" [STVar "a"] of
202             Left _ -> return ()
203             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
204         testCase "Refer to declaration ahead in list (fail)" $
205           case testDeclare [td14, td15] "T" [STVar "a"] of
206             Left _ -> return ()
207             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
208         testCase "Refer to previous declaration (success)" $
209           testDeclare [td15, td4] "T" [STVar "a"] @?= Right (STProd
     (STVar "a") (STVar "x")),
210         testCase "Non-distinct LHS type-variables (fail)" $
211           case testDeclare [td4] "X" [STVar "a"] of
212             Left _ -> return ()
213             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
214         testCase "RHS type-variable not on LHS (fail)" $
215           case testDeclare [td5] "X" [STVar "a"] of
216             Left _ -> return ()
217             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
218         testCase "Non-distinct field names (fail)" $
219           case testDeclare [td6] "X" [STVar "a"] of
220             Left _ -> return ()
221             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
222         testCase "Same field name in two declarations (fail)" $
223           case testDeclare [td7, td8] "A" [STVar "a"] of
224             Left _ -> return ()
225             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
226         testCase "Field name keyword fst (fail)" $
227           case testDeclare [td9] "X" [STVar "a"] of
228             Left _ -> return ()
229             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
230         testCase "Field name keyword snd (fail)" $
231           case testDeclare [td10] "X" [STVar "a"] of
232             Left _ -> return ()
```

```
233             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p,
234         testCase "Matching type and record constructors (succeed)"
    $
235           testDeclare [td11] "T" [STVar "a"] @?= Right (STRcd "T"
    [("a", STVar "a")]),
236         testCase "Matching variable and field names (succeed)" $
237           testDeclare [td12] "T" [STVar "a"] @?= Right (STRcd "T"
    [("t", STVar "a")]),
238         testCase "Recursive declaration (fail)" $
239           case testDeclare [td13] "T" [STVar "a"] of
240             Left _ -> return ()
241             Right p -> assertFailure $ "Unexpected resolve: " ++
    show p
242       ]
243     ],
244     testCase "Declare, then Resolve example from specs" $
245       do tce <- declare [td1]
246          st <- resolve tce (\x -> return $ STVar x) (PTApp "T" [
    PTApp "(,)" [PTVar "b", PTVar "b"]])
247          return st
248       @?= Right st1
249   ],
250   testGroup "Coder" [
251     testGroup "Pick" [
252       testCase "Pick empty" $
253         pick ([] :: [Char]) @?= Choice [],
254       testCase "Pick one choice" $
255         pick [3] @?= Found 3,
256       testCase "Pick multiple choices" $
257         pick ['a', 'b', 'c', 'd'] @?= Choice [Found 'a', Found 'b',
     Found 'c', Found 'd'],
258       testCase "Pick multiple levels" $
259           do n <- pick [0,3]
260              if n > 0 then return n
261              else pick []
262           @?= Choice [Choice [], Found 3],
263       testCase "Pick bigger tree" $
264           do n <- pick [0,3]
265              if n > 0 then return n
266              else do m <- pick [4,0]
267                      if m > 0 then return m else pick []
268           @?= tr0,
269       testCase "Pick huge tree" $
270           do n <- pick [0,3]
271              if n > 0 then return n
272              else do m <- pick [4,6]
273                      if m > 4 then return m
274                      else do o <- pick [0,3,5,2]
275                              if o > 0 then return o
276                              else do p <- pick [7,0]
277                                      if p > 0 then return p
278                                      else do q <- pick [7,0]
279                                              if q > 0 then return q
280                                              else pick [8, 9]
281           @?= Choice [Choice [Choice [Choice [Found 7,Choice [Found
     7,Choice [Found 8,Found 9]]],Found 3,Found 5,Found 2],Found
```

```
    6],Found 3]
282    ],
283    testGroup "solutions" [
284      testCase "Simple found 3" $
285        solutions (Found "a") 10 Nothing @?= ["a"],
286      testCase "Simple empty" $
287        solutions (Choice [] :: Tree Int) 10 Nothing @?= [],
288      testCase "Nested choices (BFS check)" $
289        solutions tr0 10 Nothing @?= [3,4],
290      testCase "Nested choices, empty list" $
291        solutions (Choice [Choice [], Choice [Choice [], Choice
    []], Choice [Choice []]] :: Tree Int) 10 Nothing @?= [],
292      testCase "Nested 1-5 #1" $
293        solutions tr1 10 Nothing @?= [1,2,3,4,5],
294      testCase "Nested 1-5 #2" $
295        solutions tr2 10 Nothing @?= [1,2,3,4,5],
296      testCase "Exceeding n with d is nothing" $
297        solutions tr0 1 Nothing @?= [3],
298      testCase "Exceeding n with d is 5" $
299        solutions tr0 1 (Just 5) @?= [3,5],
300      testCase "Exceeding n with bigger tree #1" $
301        solutions tr1 4 (Just 6) @?= [1,2,3,4,6],
302      testCase "Exceeding n with bigger tree #2" $
303        solutions tr2 3 Nothing @?= [1,2,3],
304      testCase "n is exact size of list, no d added #1" $
305        solutions tr1 5 (Just 6) @?= [1,2,3,4,5],
306      testCase "n is exact size of list, no d added #2" $
307        solutions tr2 5 (Just 6) @?= [1,2,3,4,5],
308      testCase "n solutions to infinite tree" $
309        solutions tr3 5 (Just 0) @?= [1,1,1,1,1,0]
310    ],
311    testGroup "Produce" [
312      testCase "produce" $
313        do e <- dfs (produce [] st0)
314           return $ case e of
315                      Lam x (Var x') | x' == x -> e0
316                      _ -> e
317        @?= [e0]
318    ]
319    -- TODO: test combination of modules!!
320    ]]
321  where pt0 = PTApp "(->)" [PTVar "a", PTVar "a"]
322        pt1 = PTApp "A" [PTVar "a"]
323        pt2 = PTVar "a"
324        pt3 = PTApp "(,)" [PTVar "a", PTVar "b"]
325        pt4 = PTApp "(->)" [PTApp "F" [PTVar "x"], PTApp "(,)" [
    PTVar "y", PTApp "A" []]]
326        pt5 = PTApp "(->)" [PTApp "F" [PTApp "F" [PTVar "x", PTVar "
    y"]], PTApp "(,)" [PTApp "T" [PTApp "(,)" [PTVar "y", PTVar "f5
    "]], PTApp "A" [PTVar "b"]]]
327        pt6 = PTApp "(->)" [PTApp "(->)" [PTVar "a", PTVar "a"],
    PTApp "(,)" [PTApp "(->)" [PTVar "a", PTVar "a"], PTVar "a"]]
328        td0 = TDSyn ("X", ["a"]) pt0
329        td1 = TDRcd ("Z", ["a"]) "C" [("x", PTVar "a"), ("f", PTApp
    "(->)" [PTVar "a", PTVar "a"])]
330        td2 = TDSyn ("T", ["a"]) pt2
331        td3 = TDSyn ("Y", ["a"]) (PTApp "(,)" [PTVar "a", PTVar "a"
```

```
    ])
332       td4 = TDSyn ("X", ["a, a"]) (PTApp "(,)" [PTVar "a", PTVar "
    a"])
333       td5 = TDSyn ("X", ["a"]) (PTApp "(,)" [PTVar "b", PTVar "b"
    ])
334       td6 = TDRcd ("X", ["a"]) "C" [("x", PTVar "a"), ("x", PTVar
    "a")]
335       td7 = TDRcd ("A", ["a"]) "C" [("x", PTVar "a")]
336       td8 = TDRcd ("B", ["a"]) "C" [("x", PTVar "a")]
337       td9 = TDRcd ("X", ["a"]) "C" [("fst", PTVar "a")]
338       td10 = TDRcd ("X", ["a"]) "C" [("snd", PTVar "a")]
339       td11 = TDRcd ("T", ["a"]) "T" [("t", PTVar "a")]
340       td12 = TDRcd ("T", ["t"]) "T" [("t", PTVar "t")]
341       td13 = TDSyn ("T", ["a"]) (PTApp "(->)" [PTVar "a", PTApp "T
    " [PTVar "a"]])
342       td14 = TDSyn ("T", ["a"]) (PTApp "(->)" [PTVar "a", PTApp "U
    " [PTVar "a"]])
343       td15 = TDSyn ("U", ["x"]) (PTVar "x")
344       td16 = TDSyn ("T", ["a", "b"]) (PTApp "(->)" [PTVar "a",
    PTVar "b"])
345       st0 = STArrow (STVar "a'") (STVar "a'")
346       st1 = STRcd "C" [("x", STProd (STVar "b") (STVar "b")), ("f"
    , STArrow (STProd (STVar "b") (STVar "b")) (STProd (STVar "b")
    (STVar "b")))]
347       tr0 = Choice [Choice [Found 4, Choice []], Found 3]
348       tr1 = Choice [Found 1, Choice [Found 3, Choice [Found 5],
    Found 4], Found 2]
349       tr2 = Choice [Choice [Choice [Choice [Choice [Found 5],
    Found 4], Found 3], Found 2], Found 1]
350       tr3 = Choice [tr3, Found 1]
351       dfs (Found a) = [a]
352       dfs (Choice ts) = concatMap dfs ts
353       e0 = Lam "X" (Var "X")
354
355 tce1 = tce0 ++ [("T", \ts ->
356   case ts of
357     [t] -> return $ STRcd "C" [("x", t), ("f", STArrow t t)]
358     _ -> Left "bad args for T")]
359
360
361 testDeclare :: [TDecl] -> TCName -> [SType] -> EM SType
362 testDeclare ds tc st = do
363   tce <- declare ds
364   tf <- case lookup tc tce of Just tf -> return tf; _ -> Left "no T
        "
365   tf st
```

Listing 4: BlackBox.hs

BlackBox Results

```
All tests
  Parser
    Name Tests
      Simple var name:                                    OK
      Var name with all possible chars:                   OK
      Var name with illegal char:                         FAIL
        tests\BlackBox.hs:28:
        Unexpected parse: PTVar "b"
        Use -p '/Var name with illegal char/' to rerun this test only.
      Simple con name:                                    OK
      Con name with all possible chars:                   OK
      Con name with illegal char:                         FAIL
        tests\BlackBox.hs:36:
        Unexpected parse: PTApp "B" []
        Use -p '/Con name with illegal char/' to rerun this test only.
      Keyword type:                                       OK
      Keyword newtype:                                    OK
      Keyword data:                                       OK
      Keyword type in middle of name:                     OK
      Keyword newtype in middle of name:                  OK
      Keyword data in middle of name:                     OK
      pType with all illegal names:                       FAIL
        tests\BlackBox.hs:58:
        Unexpected parse: PTApp "F" []
        Use -p '/pType with all illegal names/' to rerun this test only.
      tDeclz with all illegal names:                      OK
    PType tests
      Simple assigning:                                   OK
      Con with var:                                       OK
      Simple parentheses:                                 OK
      Simple tuple:                                       OK
      Tuple with constructor:                             OK
      Con with multiple types:                            OK
      Complex 1:                                          OK
      Complex 2:                                          OK
    TDeclz tests
      Empty list of declarations:                         OK
      Parse declare type:                                 OK
      Parse declare newtype:                              FAIL
        tests\BlackBox.hs:90:
        expected: Right [TDRcd ("T",[]) "B" [("c",PTVar "a")]]
         but got: Left "\"parse error\" (line 1, column 12):\
    nunexpected \"c\"\nexpecting \"{-\""
        Use -p '/Parse declare newtype/' to rerun this test only.
      Parse declare data no fields:                       OK
      Parse declare data one field:                       OK
      Parse data multiple fields:                         OK
      Parse data duplicate name fields:                   OK
      Prase synonym types:                                OK
      Prase two declarations:                             OK
      Ignore semicolon before declaration:                OK
      Parse declare with bigger pType:                    OK
```

```
    Whitespace Tests
      Space between assigning:                              OK
      Tab between assigning:                                OK
      Newline between assigning:                            OK
      A lof of whitespace between assigning:                OK
      Space at start:                                       OK
      Space at end:                                         OK
      Space between parantheses:                            OK
      Space between comma:                                  OK
      Comment at start:                                     OK
      Comment at end:                                       OK
      Comment in middle:                                    OK
      Comment that does not end:                            FAIL
        tests\BlackBox.hs:135:
        Unexpected parse: PTVar "a"
        Use -p '/Comment that does not end/' to rerun this test
    only.
      Comment in middle of declaration:                     OK
      Space between everything pType:                       OK
      Space between everything declare type:                FAIL
        tests\BlackBox.hs:141:
        expected: Right [TDSyn ("X",["a"]) (PTApp "(->)" [PTVar "a
    ",PTVar "a"])]
         but got: Right [TDSyn ("T",["a"]) (PTApp "(->)" [PTVar "a
    ",PTVar "a"])]
        Use -p '/Space between everything declare type/' to rerun
    this test only.
      Space between everything declare newtype:             OK
      Space between everything declare data:                OK
    Disambiguation Tests
      Constructor parentheses:                              OK
      Constructor tighter than infix:                       OK
      Right-associative arrow:                              OK
  Resolver
    resolve
      Test PTVar:                                           OK
      Test STProd:                                          OK
      Test STArrow:                                         OK
      Test Nested:                                          OK
      Test non-existent constructor:                        OK
      Test bad args in constructor:                         OK
      Test bad variable environment:                        OK
      Example from spec:                                    OK
    Declare
      Empty declaration list:                               OK
      Declare synonym with var:                             FAIL
        tests\BlackBox.hs:188:
        expected: Right (STVar "a'")
         but got: Left "incomplete"
        Use -p '/Declare synonym with var/' to rerun this test only
    .
      Declare synonym for (->):                             FAIL
        tests\BlackBox.hs:190:
        expected: Right (STArrow (STVar "a'") (STVar "a'"))
         but got: Left "incomplete"
        Use -p '/Declare synonym for (->)/' to rerun this test only
    .
```

```
    Declare synonym for (,):                              FAIL
      tests\BlackBox.hs:192:
      expected: Right (STProd (STVar "a'") (STVar "a'"))
       but got: Left "incomplete"
      Use -p '/Declare synonym for (,)/' to rerun this test only.
    Declare constructor with two type variables:          FAIL
      tests\BlackBox.hs:194:
      expected: Right (STVar "a'")
       but got: Left "incomplete"
      Use -p '/Declare constructor with two type variables/' to
rerun this test only.
    Example from specs:                                   FAIL
      tests\BlackBox.hs:196:
      expected: Right (STRcd "C" [("x",STProd (STVar "b") (STVar
"b")),("f",STArrow (STProd (STVar "b") (STVar "b")) (STProd (
STVar "b") (STVar "b")))])
       but got: Left "incomplete"
      Use -p '/Example from specs/' to rerun this test only.
    Four declarations:                                    FAIL
      tests\BlackBox.hs:198:
      expected: Right (STRcd "C" [("x",STProd (STVar "b") (STVar
"b")),("f",STArrow (STProd (STVar "b") (STVar "b")) (STProd (
STVar "b") (STVar "b")))])
       but got: Left "incomplete"
      Use -p '/Four declarations/' to rerun this test only.
    Semantics
      Non-distinct type constructor (fail):               OK
      Refer to declaration ahead in list (fail):          OK
      Refer to previous declaration (success):            FAIL
        tests\BlackBox.hs:209:
        expected: Right (STProd (STVar "a") (STVar "x"))
         but got: Left "incomplete"
        Use -p '/Refer to previous declaration (success)/' to
rerun this test only.
      Non-distinct LHS type-variables (fail):             OK
      RHS type-variable not on LHS (fail):                OK
      Non-distinct field names (fail):                    OK
      Same field name in two declarations (fail):         OK
      Field name keyword fst (fail):                      OK
      Field name keyword snd (fail):                      OK
      Matching type and record constructors (succeed): FAIL
        tests\BlackBox.hs:235:
        expected: Right (STRcd "T" [("a",STVar "a")])
         but got: Left "incomplete"
        Use -p '/Matching type and record constructors (succeed)
/' to rerun this test only.
      Matching variable and field names (succeed):     FAIL
        tests\BlackBox.hs:237:
        expected: Right (STRcd "T" [("t",STVar "a")])
         but got: Left "incomplete"
        Use -p '/Matching variable and field names (succeed)/' to
 rerun this test only.
      Recursive declaration (fail):                       OK
  Declare, then Resolve example from specs:               FAIL
    tests\BlackBox.hs:245:
    expected: Right (STRcd "C" [("x",STProd (STVar "b") (STVar "b
")),("f",STArrow (STProd (STVar "b") (STVar "b")) (STProd (
```

```
    STVar "b") (STVar "b")))])
       but got: Left "incomplete"
      Use -p '/Declare, then Resolve example from specs/' to rerun this test only.
  Coder
    Pick
      Pick empty:                                          OK
      Pick one choice:                                     OK
      Pick multiple choices:                               OK
      Pick multiple levels:                                OK
      Pick bigger tree:                                    OK
      Pick huge tree:                                      OK
    solutions
      Simple found 3:                                      OK
      Simple empty:                                        OK
      Nested choices (BFS check):                          OK
      Nested choices, empty list:                          OK
      Nested 1-5 #1:                                       OK
      Nested 1-5 #2:                                       OK
      Exceeding n with d is nothing:                       OK
      Exceeding n with d is 5:                             OK
      Exceeding n with bigger tree #1:                     OK
      Exceeding n with bigger tree #2:                     OK
      n is exact size of list, no d added #1:              OK
      n is exact size of list, no d added #2:              OK
      n solutions to infinite tree:                        OK
    Produce
      produce:                                             FAIL
        Exception: Prelude.undefined
        CallStack (from HasCallStack):
          error, called at libraries\base\GHC\Err.hs:79:14 in base:GHC.Err
          undefined, called at src\CoderImpl.hs:43:11 in autoprog-0.0.0-47SS6bPL6nfL22mnydv0Fa:CoderImpl
        Use -p '/produce/' to rerun this test only.

17 out of 101 tests failed (0.03s)
```

Listing 5: BlackBox.txt

Frappe

```
-module(frappe).

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

% Export at least the API:
-export([fresh/1,
         set/4,
         read/2,
         insert/4,
         update/4,
         upsert/3,
         stable/3,
         all_items/1,
         stop/1
        ]).

-export([]).

fresh(Cap) ->
  server:start(Cap).

set(FS, Key, Value, C) ->
  server:set(FS, Key, Value, C).

read(FS, Key) ->
  server:read(FS, Key).

insert(FS, Key, Value, C) ->
  server:insert(FS, Key, Value, C).

update(FS, Key, Value, C) ->
  server:update(FS, Key, Value, C).

upsert(FS, Key, Fun) ->
  server:upsert(FS, Key, Fun).

stable(_FS, _Key, _Ref) ->
  not_implemented.

all_items(FS) ->
  server:all_items(FS).

stop(FS) ->
  server:stop(FS).
```

Listing 6: frappe.erl

Frappe Server

```erlang
-module(server).

-export([start/1, set/4, read/2, insert/4, update/4, upsert/3,
    all_items/1, stop/1]).

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

-behaviour(gen_server).

%% export functions
start(Cap) ->
  case Cap > 0 of
    true -> gen_server:start(?MODULE, Cap, []);
    false -> {error, "Capacity must be positive integer"}
  end.

set(FS, Key, Value, C) ->
  gen_server:call(FS, {set, {Key, Value, C}}).

read(FS, Key) ->
  gen_server:call(FS, {read, Key}).

insert(FS, Key, Value, C) ->
  gen_server:call(FS, {insert, {Key, Value, C}}).

update(FS, Key, Value, C) ->
  gen_server:call(FS, {update, {Key, Value, C}}).

upsert(FS, Key, Fun) ->
  gen_server:call(FS, {upsert, {Key, Fun}}).

all_items(FS) ->
  gen_server:call(FS, {all_items}).

stop(FS) ->
  gen_server:stop(FS).

%% server functions
init(Cap) ->
  {ok, {Cap, []}}.

handle_call(Request, _From, {Cap, Queue} = State) ->
  case Request of
    {insert, {Key, _Val, _C} = Item} ->
      % Check if key exists
      case lists:keymember(Key, 1, Queue) of
        true ->
          {reply, {error, "Key already exists"}, State};
        false ->
          insert_item(Item, State)
      end;
    {read, Key} ->
      % Get key
      case lists:keyfind(Key, 1, Queue) of
```

```
55          {Key, Val, _C} = Item ->
56            % Reorder queue
57            {reply, {ok, Val}, {Cap, lists:keydelete(Key, 1, Queue)
    ++ [Item]}};
58          false ->
59            {reply, nothing, State}
60        end;
61      {update, {Key, _Val, _C} = Item} ->
62        % Check if key exists
63        case lists:keymember(Key, 1, Queue) of
64          true ->
65            update_item(Item, State);
66          false ->
67            {reply, {error, "Key does not exists"}, State}
68        end;
69      {set, {Key, _Val, _C} = Item} ->
70        % Check if key exists
71        case lists:keymember(Key, 1, Queue) of
72          true ->
73            update_item(Item, State);
74          false ->
75            insert_item(Item, State)
76        end;
77      {upsert, {Key, Fun}} ->
78        upsert_item(Key, Fun, State);
79      {all_items} ->
80        {reply, Queue, State}
81    end.
82
83  handle_cast(_Request, _State) -> undefined.
84
85  %% separation of concerns
86  insert_item({_Key, _Val, C} = Item, {Cap, Queue} = State) ->
87    case C =< Cap of
88      true ->
89        NewQueue = Queue ++ [Item],
90        case is_exceeded_capacity(Cap, NewQueue) of
91          true ->
92            {_, PoppedQueue} = lists:split(1, NewQueue),
93            {reply, ok, {Cap, PoppedQueue}};
94          false ->
95            {reply, ok, {Cap, NewQueue}}
96        end;
97      false ->
98        {reply, {error, "Broken capacity invariant"}, State}
99    end.
100
101 update_item({Key, _Val, C} = Item, {Cap, Queue} = State) ->
102   case C =< Cap of
103     true ->
104       % Update and reorder in queue
105       NewQueue = lists:keydelete(Key, 1, Queue) ++ [Item],
106       case is_exceeded_capacity(Cap, NewQueue) of
107         true ->
108           {_, PoppedQueue} = lists:split(1, NewQueue),
109           {reply, ok, {Cap, PoppedQueue}};
110         false ->
```

```
          {reply, ok, {Cap, NewQueue}}
      end;
    false ->
      {reply, {error, "Broken capacity invariant"}, State}
  end.

upsert_item(Key, Fun, {_Cap, Queue} = State) ->
  % Get key
  case lists:keyfind(Key, 1, Queue) of
    {Key, Val, _C} ->
      Arg = {existing, Val};
    false ->
      Arg = new
  end,
  try
    upsert_handle_result(Fun(Arg), Arg, Key, State)
  catch
    _ : Ex ->
      upsert_handle_result(Ex, Arg, Key, State)
  end.

upsert_handle_result(Result, Arg, Key, State) ->
  case Result of
    {new_value, NewVal, NewC} ->
      case Arg of
        {existing, _} ->
          update_item({Key, NewVal, NewC}, State);
        new ->
          insert_item({Key, NewVal, NewC}, State)
      end;
    _ ->
      {reply, ok, State}
  end.

%% helper functions
is_exceeded_capacity(Cap, Queue) ->
  sum_capacities(Queue) > Cap.

sum_capacities([]) -> 0;
sum_capacities([{_, _, C} | Queue]) ->
  C + sum_capacities(Queue).
```

Listing 7: frappe.erl

Frappe QuickCheck

```
-module(test_frappe).

-export([test_all/0, test_everything/0]).
-export([initial_state/0, command/1, precondition/2, postcondition
    /3, next_state/3]).
-export([mktrans/2, terminating_transformation/1,
    prop_cache_under_capacity/0, prop_capacity_invariant/0,
    prop_unique_keys/0, prop_stopped_server/0]).

-export([prop_cache/0, frappe_fresh/0]).

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

-behaviour(apqc_statem).

test_all() ->
  eqc:quickcheck(test_frappe:prop_cache_under_capacity()),
  eqc:quickcheck(test_frappe:prop_capacity_invariant()),
  eqc:quickcheck(test_frappe:prop_unique_keys()),
  eqc:quickcheck(test_frappe:prop_stopped_server()),
  test_eunit:test_all().

test_everything() ->
  test_all().

%% TODO: use limited versions since I haven't implemented stable/2

% Cache behaves normally under capacity
prop_cache_under_capacity() -> undefined.

% Capacity invariant is never broken
prop_capacity_invariant() ->
  ?FORALL(FS, cache(50),
    capacity_invariant(FS, 50)).

% No duplicate keys
prop_unique_keys() ->
  ?FORALL(FS, cache(),
    no_duplicates(frappe:all_items(FS))). %% TODO: convert to list
    of keys

% Stopped servers cannot receive calls
prop_stopped_server() -> undefined.

% Transformation generator
mktrans(Opr, Args) ->
  case Opr of
    new ->
      fun(new) -> {new_value, Args, 5} end;
    add_val ->
      fun({existing, Val}) -> {new_value, Val+Args, 5} end;
    set_cap ->
      fun({existing, Val}) -> {new_value, Val, Args} end;
    val_is_cap ->
```

```
52        fun({existing, Val}) -> {new_value, Val, Val} end;
53      throw ->
54        fun({existing, _}) -> throw(Args) end
55    end.
56
57  terminating_transformation(KeyGen) ->
58    {KeyGen, ?LET(Fun, list(trans_fun()), {call, ?MODULE, mktrans,
        Fun})}.
59
60  trans_fun() ->
61    {oneof([new, add_val, set_cap, val_is_cap, throw]), value()}.
62
63  %% Attempt at cache generator #1:
64  cache() ->
65    {ok, FS} = frappe:fresh(int()),
66    eval(gen_cache(FS)),
67    FS.
68  cache(Cap) ->
69    {ok, FS} = frappe:fresh(Cap),
70    eval(gen_cache(FS)),
71    FS.
72
73  gen_cache(FS) ->
74    ?LAZY(
75        oneof([
76          ?LET({K,V,C}, {key(), value(), cap()}, {call, frappe, set,
        [FS,K,V,C]})
77        ])
78      ).
79
80  % Attempt at cache generator #2: (aqpc)
81  prop_cache() ->
82    ?FORALL(Cmds, commands(?MODULE),
83      begin
84        {_, FS, _} = Result = run_commands(?MODULE, Cmds),
85        cleanup(FS),
86        check_commands(Cmds, Result)
87      end).
88
89  check_commands(Cmds, {_,_,Res} = HSRes) ->
90    pretty_commands(?MODULE, Cmds, HSRes,
91      aggregate(command_names(Cmds),
92        equals(Res, ok))).
93
94  cleanup(FS) ->
95    frappe:stop(FS).
96
97  -record(state,{fs}).
98
99  initial_state() ->
100   #state{fs = none}.
101
102 command(#state{fs = none}) ->
103   return({call, ?MODULE, frappe_fresh, [[]]});
104 command(FS) ->
105   oneof([{call, frappe, set, [FS, key(), value(), cap()] }]).
106
```

```
107 next_state(S, FS, {call, frappe_fresh, _}) ->
108   S#state{fs = FS};
109 next_state(S, _, _) ->
110   S.
111
112 precondition(_, _) ->
113   true.
114
115 postcondition(_, _, _) ->
116   true.
117
118 frappe_fresh() ->
119   {ok, FS} = frappe:fresh(int()),
120   FS.
121
122 % Other generators
123 key() ->
124   oneof([int(), char()]).
125 value() ->
126   frequency([{4, int()}, {1, char()}]). % minimise chance of error
      in transformation function (eg. char+int, setting char as cap)
127 cap() ->
128   int().
129
130 % Helper functions
131 no_duplicates(Lst) ->
132   length(Lst) =:= length(lists:usort(Lst)). %% TODO: THIS IS WRONG,
        DONT USE USORT
133
134 capacity_invariant(FS, Cap) ->
135   sum_capacities(frappe:all_items(FS)) =< Cap.
136
137 sum_capacities([]) -> 0;
138 sum_capacities([{_, _, C} | Queue]) ->
139   C + sum_capacities(Queue).
```

Listing 8: test_frappe.erl

Frappe EUnit

```
-module(test_eunit).

-export([test_all/0]).

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

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

testsuite() ->
    [ {"Start/stop tests", spawn,
       [
         test_fresh(),
         test_fresh_multiple(),
         test_fresh_negative_cap(),
         test_stop(),
         test_stop_multiple()
       ]
      },
      {"Insert tests", spawn,
       [
         test_insert(),
         test_insert_over_cap(),
         test_insert_existing()
       ]
      },
      {"Read tests", spawn,
       [
         test_read_existing(),
         test_read_nonexisting()
       ]
      },
      {"Update tests", spawn,
       [
         test_update(),
         test_update_over_cap(),
         test_update_nonexisting()
       ]
      },
      {"Set tests", spawn,
       [
         test_set_nonexisting(),
         test_set_nonexisting_then_read(),
         test_set_nonexisting_over_cap(),
         test_set_existing_then_read(),
         test_set_existing_over_cap()
       ]
      },
      {"All_Items tests", spawn,
       [
         test_all_items(),
         test_all_items_empty()
       ]
      },
      {"Upsert tests", spawn,
       [
```

```
56          test_upsert_insert(),
57          test_upsert_insert_over_cap(),
58          test_upsert_insert_existing(),
59          test_upsert_insert_returns_wrong(),
60          test_upsert_insert_function_throws_newval(),
61          test_upsert_insert_function_throws_error(),
62
63          test_upsert_update(),
64          test_upsert_update_over_cap(),
65          test_upsert_update_nonexisting(),
66          test_upsert_update_returns_wrong(),
67          test_upsert_update_function_throws_newval(),
68          test_upsert_update_function_throws_error()
69        ]
70      },
71      {"Stable tests", spawn,
72        [
73          % TODO
74        ]
75      },
76      {"LRU tests", spawn,
77        [
78          test_LRU_set_removes_item(),
79          test_LRU_insert_removes_item(),
80          test_LRU_update_removes_item(),
81          test_LRU_upsert_insert_removes_item(),
82          test_LRU_upsert_update_removes_item(),
83
84          test_LRU_read_changes_order(),
85          test_LRU_update_changes_order(),
86          test_LRU_upsert_update_changes_order_on_success(),
87          test_LRU_upsert_no_update_does_not_change_order()
88         %  test_LRU_stable_changes_order()
89        ]
90      },
91      {"Key coherency tests", spawn,
92        [
93          test_key_coherency_set(),
94          test_key_coherency_update(),
95          test_key_coherency_upsert()
96        ]
97      },
98      {"Key concurrency tests", spawn,
99        [
100         % TODO
101       ]
102     },
103     {"Efficienty tests", spawn,
104       [
105         % TODO
106       ]
107     }
108   ].
109
110 test_fresh() ->
111   {"Start a frappe server",
112     fun () ->
```

```
113       ?assertMatch({ok, _}, frappe:fresh(5))
114     end }.
115
116 test_fresh_multiple() ->
117   {"Start multiple frappe servers",
118     fun () ->
119       ?assertMatch({ok, _}, frappe:fresh(5)),
120       ?assertMatch({ok, _}, frappe:fresh(6)),
121       ?assertMatch({ok, _}, frappe:fresh(12))
122     end }.
123
124 test_fresh_negative_cap() ->
125   {"Start frappe server with negative capacity",
126     fun () ->
127       ?assertMatch({error, _}, frappe:fresh(-5))
128     end }.
129
130 test_stop() ->
131   {"Start and stop a frappe server",
132     fun () ->
133       {ok, FS} = frappe:fresh(5),
134       ?assertMatch(ok, frappe:stop(FS))
135     end }.
136
137 test_stop_multiple() ->
138   {"Start and stop multiple frappe servers",
139     fun () ->
140       {ok, FS} = frappe:fresh(5),
141       {ok, FS2} = frappe:fresh(6),
142       {ok, FS3} = frappe:fresh(12),
143       ?assertMatch(ok, frappe:stop(FS)),
144       ?assertMatch(ok, frappe:stop(FS2)),
145       ?assertMatch(ok, frappe:stop(FS3))
146     end }.
147
148 test_insert() ->
149   {"Insert item",
150     fun () ->
151       {ok, FS} = frappe:fresh(5),
152       ?assertEqual(ok, frappe:insert(FS, key, val, 3))
153     end }.
154
155 test_insert_over_cap() ->
156   {"Insert that breaks capacity invariant",
157     fun () ->
158       {ok, FS} = frappe:fresh(5),
159       ?assertMatch({error, _}, frappe:insert(FS, key, val, 6))
160     end }.
161
162 test_insert_existing() ->
163   {"Insert item with key that exists",
164     fun () ->
165       {ok, FS} = frappe:fresh(5),
166       frappe:insert(FS, key, val, 3),
167       ?assertMatch({error, _}, frappe:insert(FS, key, val, 2))
168     end }.
169
```

```
170 test_read_existing() ->
171   {"Read existing item",
172     fun () ->
173       {ok, FS} = frappe:fresh(5),
174       frappe:insert(FS, key, val, 3),
175       ?assertEqual({ok, val}, frappe:read(FS, key))
176     end }.
177
178 test_read_nonexisting() ->
179   {"Read non-existing item",
180     fun () ->
181       {ok, FS} = frappe:fresh(5),
182       ?assertEqual(nothing, frappe:read(FS, key))
183     end }.
184
185 test_update() ->
186   {"Update existing item",
187     fun () ->
188       {ok, FS} = frappe:fresh(5),
189       frappe:insert(FS, key, val, 3),
190       frappe:update(FS, key, newval, 3),
191       ?assertEqual({ok, newval}, frappe:read(FS, key))
192     end }.
193
194 test_update_over_cap() ->
195   {"Update to break capacity invariant",
196     fun () ->
197       {ok, FS} = frappe:fresh(5),
198       frappe:insert(FS, key, val, 3),
199       ?assertMatch({error, _}, frappe:update(FS, key, val, 6))
200     end }.
201
202 test_update_nonexisting() ->
203   {"Update nonexisting item",
204     fun () ->
205       {ok, FS} = frappe:fresh(5),
206       ?assertMatch({error, _}, frappe:update(FS, key, val, 3))
207     end }.
208
209 test_set_nonexisting() ->
210   {"Set new item that does not break capacity invariant",
211     fun () ->
212       {ok, FS} = frappe:fresh(5),
213       ?assertEqual(ok, frappe:set(FS, key, val, 3))
214     end }.
215
216 test_set_nonexisting_over_cap() ->
217   {"Set new item that does break capacity invariant",
218     fun () ->
219       {ok, FS} = frappe:fresh(5),
220       ?assertMatch({error, _}, frappe:set(FS, key, val, 6))
221     end }.
222
223 test_set_nonexisting_then_read() ->
224   {"Set new item that does not break capacity invariant and read",
225     fun () ->
226       {ok, FS} = frappe:fresh(5),
```

```
227       frappe:set(FS, key, val, 3),
228       ?assertEqual({ok, val}, frappe:read(FS, key))
229     end }.
230
231 test_set_existing_then_read() ->
232   {"Set existing item so that it does not break capacity invariant"
      ,
233     fun () ->
234       {ok, FS} = frappe:fresh(5),
235       frappe:insert(FS, key, val, 3),
236       frappe:set(FS, key, newval, 3),
237       ?assertEqual({ok, newval}, frappe:read(FS, key))
238     end }.
239
240 test_set_existing_over_cap() ->
241   {"Set existing item so that it does break capacity invariant",
242     fun () ->
243       {ok, FS} = frappe:fresh(5),
244       frappe:insert(FS, key, val, 3),
245       ?assertMatch({error, _}, frappe:set(FS, key, val, 6))
246     end }.
247
248 test_all_items() ->
249   {"Insert three items then call all_items/1",
250     fun () ->
251       {ok, FS} = frappe:fresh(10),
252       frappe:insert(FS, key, val, 3),
253       frappe:insert(FS, key2, val, 3),
254       frappe:insert(FS, key3, val, 3),
255       ?assertEqual([{key,val,3},{key2,val,3},{key3,val,3}], frappe:
      all_items(FS))
256     end }.
257
258 test_all_items_empty() ->
259   {"Call all_items/1 on an empty cache",
260     fun () ->
261       {ok, FS} = frappe:fresh(5),
262       ?assertEqual([], frappe:all_items(FS))
263     end }.
264
265 test_upsert_insert() ->
266   {"Call upsert to insert a new item",
267     fun () ->
268       {ok, FS} = frappe:fresh(5),
269       ok = frappe:upsert(FS, key,
270                 fun(new) -> {new_value, val, 4} end),
271       ?assertEqual({ok, val}, frappe:read(FS, key))
272     end }.
273
274 test_upsert_insert_over_cap() ->
275   {"Call upsert to insert a new item and break capacity invariant",
276     fun () ->
277       {ok, FS} = frappe:fresh(5),
278       ?assertMatch({error, _}, frappe:upsert(FS, key,
279                 fun(new) -> {new_value, val, 6} end)),
280       ?assertEqual(nothing, frappe:read(FS, key))
281     end }.
```

```
282
283 test_upsert_insert_existing() ->
284   {"Call upsert to insert an already existing key (bad_arg to
        function)",
285     fun () ->
286       {ok, FS} = frappe:fresh(5),
287       ok = frappe:insert(FS, key, val, 3),
288       ?assertEqual(ok, frappe:upsert(FS, key,
289                 fun(new) -> {new_value, 2, 3} end)),
290       ?assertEqual({ok, val}, frappe:read(FS, key))
291     end }.
292
293 test_upsert_insert_returns_wrong() ->
294   {"Call upsert to insert new item with function that returns
        incorrect response",
295     fun () ->
296       {ok, FS} = frappe:fresh(5),
297       ?assertEqual(ok, frappe:upsert(FS, key, fun({new}) -> no end)
        )
298     end }.
299
300 test_upsert_insert_function_throws_newval() ->
301   {"Call upsert with function that throws a newvalue and then
        inserts",
302     fun () ->
303       {ok, FS} = frappe:fresh(5),
304       ?assertMatch(ok, frappe:upsert(FS, key,
305                 fun(new) -> throw({new_value, 1, 4}) end)),
306       ?assertEqual({ok, 1}, frappe:read(FS, key))
307     end }.
308
309 test_upsert_insert_function_throws_error() ->
310   {"Call upsert to insert new item with function that throws an
        error",
311     fun () ->
312       {ok, FS} = frappe:fresh(5),
313       ?assertMatch(ok, frappe:upsert(FS, key,
314                 fun(new) -> throw("no") end))
315     end }.
316
317 test_upsert_update() ->
318   {"Call upsert to update an existing item",
319     fun () ->
320       {ok, FS} = frappe:fresh(5),
321       ok = frappe:insert(FS, key, 23, 3),
322       ok = frappe:upsert(FS, key,
323                 fun({existing, Val}) -> {new_value, Val+2, 4} end),
324       ?assertEqual({ok, 25}, frappe:read(FS, key))
325     end }.
326
327 test_upsert_update_over_cap() ->
328   {"Call upsert to update an item and break capacity invariant",
329     fun () ->
330       {ok, FS} = frappe:fresh(5),
331       ok = frappe:insert(FS, key, 2, 3),
332       ?assertMatch({error, _}, frappe:upsert(FS, key,
333                 fun({existing, Val}) -> {new_value, Val+2, 6} end))
```

```
    ,
      ?assertEqual({ok, 2}, frappe:read(FS, key))
    end }.

test_upsert_update_nonexisting() ->
  {"Call upsert to update a nonexisting key (bad_arg to function)",
    fun () ->
      {ok, FS} = frappe:fresh(5),
      ?assertMatch(ok, frappe:upsert(FS, key,
                fun({existing, Val}) -> {new_value, Val+3, 3} end))
    ,
      ?assertEqual(nothing, frappe:read(FS, key))
    end }.

test_upsert_update_returns_wrong() ->
  {"Call upsert to update with function that returns incorrect
    response",
    fun () ->
      {ok, FS} = frappe:fresh(5),
      ok = frappe:insert(FS, key, 23, 3),
      ?assertEqual(ok, frappe:upsert(FS, key,
                fun({existing, _Val}) -> no end))
    end }.

test_upsert_update_function_throws_newval() ->
  {"Call upsert with function that throws a newvalue and then
    updates",
    fun () ->
      {ok, FS} = frappe:fresh(5),
      ok = frappe:insert(FS, key, 1, 3),
      ?assertMatch(ok, frappe:upsert(FS, key,
                fun({existing, Val}) -> throw({new_value, Val+1, 4}
    ) end)),
      ?assertEqual({ok, 2}, frappe:read(FS, key))
    end }.

test_upsert_update_function_throws_error() ->
  {"Call upsert to update with function that tries to add integer
    to string",
    fun () ->
      {ok, FS} = frappe:fresh(5),
      ok = frappe:insert(FS, key, val, 3),
      ?assertMatch(ok, frappe:upsert(FS, key,
                fun({existing, Val}) ->
                  New = Val+2,
                  {new_value, New, 4}
                end)),
      ?assertEqual({ok, val}, frappe:read(FS, key))
    end }.

test_LRU_set_removes_item() ->
  {"Test LRU holds for set function",
    fun () ->
      {ok, FS} = frappe:fresh(5),
      frappe:set(FS, key, val, 3),
      frappe:set(FS, key2, val, 3),
      ?assertEqual(nothing, frappe:read(FS, key)),
```

```
385       ?assertEqual({ok, val}, frappe:read(FS, key2))
386     end }.
387
388 test_LRU_insert_removes_item() ->
389   {"Test LRU holds for insert function",
390     fun () ->
391       {ok, FS} = frappe:fresh(5),
392       frappe:insert(FS, key, val, 3),
393       frappe:insert(FS, key2, val, 3),
394       ?assertEqual(nothing, frappe:read(FS, key)),
395       ?assertEqual({ok, val}, frappe:read(FS, key2))
396     end }.
397
398 test_LRU_update_removes_item() ->
399   {"Test LRU holds for update function",
400     fun () ->
401       {ok, FS} = frappe:fresh(5),
402       frappe:insert(FS, key, val, 3),
403       frappe:insert(FS, key2, val, 2),
404       frappe:update(FS, key2, newval, 3),
405       ?assertEqual(nothing, frappe:read(FS, key)),
406       ?assertEqual({ok, newval}, frappe:read(FS, key2))
407     end }.
408
409 test_LRU_upsert_insert_removes_item() ->
410   {"Test LRU holds for upsert function when inserting",
411     fun () ->
412       {ok, FS} = frappe:fresh(5),
413       ok = frappe:insert(FS, key, newval, 2),
414       ok = frappe:upsert(FS, key2, fun(new) -> {new_value, val, 4}
      end),
415       ?assertEqual(nothing, frappe:read(FS, key)),
416       ?assertEqual({ok, val}, frappe:read(FS, key2))
417     end }.
418
419 test_LRU_upsert_update_removes_item() ->
420   {"Test LRU holds for upsert function when updating",
421     fun () ->
422       {ok, FS} = frappe:fresh(5),
423       ok = frappe:insert(FS, key, newval, 2),
424       ok = frappe:insert(FS, key2, 4, 2),
425       ok = frappe:upsert(FS, key2, fun({existing, Val}) -> {
      new_value, Val+1, 4} end),
426       ?assertEqual(nothing, frappe:read(FS, key)),
427       ?assertEqual({ok, 5}, frappe:read(FS, key2))
428     end }.
429
430 test_LRU_read_changes_order() ->
431   {"Test changing of order by adding two items, reading LRU,
      inserting another item to break capacity, then attempt to read
      popped item",
432     fun () ->
433       {ok, FS} = frappe:fresh(5),
434       frappe:insert(FS, key, val, 3),
435       frappe:insert(FS, key2, val, 2),
436       frappe:read(FS, key),
437       frappe:insert(FS, key3, val, 1),
```

```
438       ?assertEqual(nothing, frappe:read(FS, key2))
439     end }.
440
441 test_LRU_update_changes_order() ->
442   {"Test changing of order by adding two items, updating LRU,
        inserting another item to break capacity, then attempt to read
        popped item",
443     fun () ->
444       {ok, FS} = frappe:fresh(5),
445       frappe:insert(FS, key, val, 3),
446       frappe:insert(FS, key2, val, 2),
447       frappe:update(FS, key, newval, 3),
448       frappe:insert(FS, key3, val, 1),
449       ?assertEqual(nothing, frappe:read(FS, key2))
450     end }.
451
452 test_LRU_upsert_update_changes_order_on_success() ->
453   {"Test changing of order by adding two items, successfully upsert
         updating LRU, inserting another item to break capacity, then
        attempt to read popped item",
454     fun () ->
455       {ok, FS} = frappe:fresh(5),
456       frappe:insert(FS, key, 3, 3),
457       frappe:insert(FS, key2, val, 2),
458       frappe:upsert(FS, key, fun({existing, Val}) -> {new_value,
      Val+2, 3} end),
459       frappe:insert(FS, key3, val, 1),
460       ?assertEqual(nothing, frappe:read(FS, key2))
461     end }.
462
463 test_LRU_upsert_no_update_does_not_change_order() ->
464   {"Test changing of order by adding two items, unsuccessfully
        upsert updating LRU, inserting another item to break capacity,
        then attempt to read popped item",
465     fun () ->
466       {ok, FS} = frappe:fresh(5),
467       frappe:insert(FS, key, 3, 3),
468       frappe:insert(FS, key2, val, 2),
469       frappe:upsert(FS, key, fun({existing, _Val}) -> no end),
470       frappe:insert(FS, key3, val, 1),
471       ?assertEqual({ok, val}, frappe:read(FS, key2))
472     end }.
473
474 test_key_coherency_set() ->
475   {"Make two set calls to same key in sequencial order",
476     fun () ->
477       {ok, FS} = frappe:fresh(5),
478       ok = frappe:insert(FS, key, 1, 3),
479       ok = frappe:set(FS, key, 2, 3),
480       ok = frappe:set(FS, key, 3, 3),
481       ?assertEqual({ok, 3}, frappe:read(FS, key))
482     end }.
483
484 test_key_coherency_update() ->
485   {"Make two update calls to same key in sequencial order",
486     fun () ->
487       {ok, FS} = frappe:fresh(5),
```

```
488       ok = frappe:insert(FS, key, 1, 3),
489       ok = frappe:update(FS, key, 2, 3),
490       ok = frappe:update(FS, key, 3, 3),
491       ?assertEqual({ok, 3}, frappe:read(FS, key))
492     end }.
493
494 test_key_coherency_upsert() ->
495   {"Make two upsert calls to same key in sequencial order",
496     fun () ->
497       {ok, FS} = frappe:fresh(5),
498       ok = frappe:insert(FS, key, 1, 3),
499       ok = frappe:upsert(FS, key, fun({existing, Val}) -> {
      new_value, Val+1, 3} end),
500       ok = frappe:upsert(FS, key, fun({existing, Val}) -> {
      new_value, Val+1, 3} end),
501       ?assertEqual({ok, 3}, frappe:read(FS, key))
502     end }.
```

Listing 9: test_eunit.erl

Frappe Testing Results

```
Failed! Reason:
{'EXIT',{bad_property,undefined}}
After 1 tests.
.....................................................................................................

OK, passed 100 tests
.....................................................................................................

OK, passed 100 tests
Failed! Reason:
{'EXIT',{bad_property,undefined}}
After 1 tests.
======================== EUnit ========================
Start/stop tests
  test_eunit: test_fresh (Start a frappe server)...ok
  test_eunit: test_fresh_multiple (Start multiple frappe servers)
    ...ok
  test_eunit: test_fresh_negative_cap (Start frappe server with
    negative capacity)...ok
  test_eunit: test_stop (Start and stop a frappe server)...ok
  test_eunit: test_stop_multiple (Start and stop multiple frappe
    servers)...ok
  [done in 0.078 s]
Insert tests
  test_eunit: test_insert (Insert item)...ok
  test_eunit: test_insert_over_cap (Insert that breaks capacity
    invariant)...ok
  test_eunit: test_insert_existing (Insert item with key that
    exists)...ok
  [done in 0.047 s]
Read tests
  test_eunit: test_read_existing (Read existing item)...ok
  test_eunit: test_read_nonexisting (Read non-existing item)...ok
  [done in 0.031 s]
Update tests
  test_eunit: test_update (Update existing item)...ok
  test_eunit: test_update_over_cap (Update to break capacity
    invariant)...ok
  test_eunit: test_update_nonexisting (Update nonexisting item)...
    ok
  [done in 0.047 s]
Set tests
  test_eunit: test_set_nonexisting (Set new item that does not
    break capacity invariant)...ok
  test_eunit: test_set_nonexisting_then_read (Set new item that
    does not break capacity invariant and read)...ok
  test_eunit: test_set_nonexisting_over_cap (Set new item that does
     break capacity invariant)...ok
  test_eunit: test_set_existing_then_read (Set existing item so
    that it does not break capacity invariant)...ok
  test_eunit: test_set_existing_over_cap (Set existing item so that
     it does break capacity invariant)...ok
  [done in 0.078 s]
All_Items tests
  test_eunit: test_all_items (Insert three items then call
```

```
    all_items/1)...ok
  test_eunit: test_all_items_empty (Call all_items/1 on an empty
    cache)...ok
  [done in 0.032 s]
Upsert tests
  test_eunit: test_upsert_insert (Call upsert to insert a new item)
    ...ok
  test_eunit: test_upsert_insert_over_cap (Call upsert to insert a
    new item and break capacity invariant)...ok
  test_eunit: test_upsert_insert_existing (Call upsert to insert an
     already existing key (bad_arg to function))...ok
  test_eunit: test_upsert_insert_returns_wrong (Call upsert to
    insert new item with function that returns incorrect response)
    ...ok
  test_eunit: test_upsert_insert_function_throws_newval (Call
    upsert with function that throws a newvalue and then inserts)
    ...ok
  test_eunit: test_upsert_insert_function_throws_error (Call upsert
     to insert new item with function that throws an error)...ok
  test_eunit: test_upsert_update (Call upsert to update an existing
     item)...ok
  test_eunit: test_upsert_update_over_cap (Call upsert to update an
     item and break capacity invariant)...ok
  test_eunit: test_upsert_update_nonexisting (Call upsert to update
     a nonexisting key (bad_arg to function))...ok
  test_eunit: test_upsert_update_returns_wrong (Call upsert to
    update with function that returns incorrect response)...ok
  test_eunit: test_upsert_update_function_throws_newval (Call
    upsert with function that throws a newvalue and then updates)
    ...ok
  test_eunit: test_upsert_update_function_throws_error (Call upsert
     to update with function that tries to add integer to string)
    ...ok
  [done in 0.187 s]
Stable tests
LRU tests
  test_eunit: test_LRU_set_removes_item (Test LRU holds for set
    function)...ok
  test_eunit: test_LRU_insert_removes_item (Test LRU holds for
    insert function)...ok
  test_eunit: test_LRU_update_removes_item (Test LRU holds for
    update function)...ok
  test_eunit: test_LRU_upsert_insert_removes_item (Test LRU holds
    for upsert function when inserting)...ok
  test_eunit: test_LRU_upsert_update_removes_item (Test LRU holds
    for upsert function when updating)...ok
  test_eunit: test_LRU_read_changes_order (Test changing of order
    by adding two items, reading LRU, inserting another item to
    break capacity, then attempt to read popped item)...ok
  test_eunit: test_LRU_update_changes_order (Test changing of order
     by adding two items, updating LRU, inserting another item to
    break capacity, then attempt to read popped item)...ok
  test_eunit: test_LRU_upsert_update_changes_order_on_success (Test
     changing of order by adding two items, successfully upsert
    updating LRU, inserting another item to break capacity, then
    attempt to read popped item)...ok
  test_eunit: test_LRU_upsert_no_update_does_not_change_order (Test
```

```
     changing of order by adding two items, unsuccessfully upsert
    updating LRU, inserting another item to break capacity, then
    attempt to read popped item)...ok
  [done in 0.156 s]
Key coherency tests
  test_eunit: test_key_coherency_set (Make two set calls to same
    key in sequencial order)...ok
  test_eunit: test_key_coherency_update (Make two update calls to
    same key in sequencial order)...ok
  test_eunit: test_key_coherency_upsert (Make two upsert calls to
    same key in sequencial order)...ok
  [done in 0.047 s]
Key concurrency tests
Efficienty tests
=======================================================
  All 44 tests passed.
```

Listing 10: Frappe_test.txt