Advanced Programming
Exam 2015

Victor Petrén Bach Hansen - grn762

September 20, 2017

Contents

1 Subscript, Parser

DISCLAIMER: The grammar used for this question is the 1.2 version!

The source code for the `SubsParser.hs` discussed in this section can be found in appendix **??**. The associated test can be found in appendix **??**.

In order to compile `SubsParser.hs`, the package `MissingH` has to be installed, which is need for the `Data.String.Utils` library. This library is used to strip a string of its trailing whitespaces and replacing newlines with spaces, i.e. string manipulation.

1.1 Parser library

The parser is built using the SimpleParse parser combinator library that was provided. It is constructed by combining various parsers that both checks that the input is well formed, based on the grammar, and builds the internal SubsAst representation.

Simple parse was chosen since it was the library I was familiar with from the previous assignment. A downside of using the `SimpleParse` library is the lack of error messages, only the following is currently supported:

1. `Error NoParse`: Indicating that the parser was not able to construct any ASTs from the given string/file
2. `Error AmbiguousParse`: Indicating that the parser was able to construct multiple ASTs from the given string/file

1.2 Updated grammar

Multiple places in the grammar, namely in `Expr1`, were there occurrences of self left recursiveness. The grammar being left recursive is bad for top down parsing, since we can't determine where to go by looking at the first symbol.

`Expr1` also needed to account for the precedence and associativity of the various operators, i.e. the fact that the arithmetic operators `*` and `/` binds stronger than `+` and `-`, which in turn binds stronger than the `<` operator etc. The transformations regarding precedence and associativity was done according to Table 1 in the assignment text.

Below, the transformed parts of the grammar can be seen (I do apologize in advance for the, at times, confusing naming convention):

```
Expr     ::= Expr1 ExprOpt

ExprOpt ::= "," Expr1 ExprOpt
          | e

Expr1    ::= T1 E_EQ
```

```
          | Ident E_AsStr

E_AsExp ::= Ident E_AsStr E_AsExp
          | e

E_AsStr ::= "=" Expr1 E_AsExp
          | e

E_EQ    ::= "===" T1 E_EQ
          | e

T1      ::= T2 E_LT

E_LT    ::= "<" T2 E_LT
          | e

T2      ::= T3 E_PM

E_PM    ::= "+" T3 E_PM
          | "-" T3 E_PM
          | e

T3      ::= T4 E_MM

E_MM    ::= "*" T4 E_MM
          | "%" T4 E_MM
          | e

T4      ::= Number
          | String
          | true
          | false
          | undefined
          | '[' Exprs  ']'
          | '(' Expr ')'
          | '[' 'for' '(' Ident 'of' Expr ')' ArrayCompr Expr ']'
          | Ident AfterIdent

AfterIdent := FunCall
            | e
```

Due to the associativy and precedence of the "=" operator, this was was extracted from the `AfterIdent`.

Left recursiong could also have been dealt with by using the functions `chainl`, but I had a much clearer idea of the approach that was taken.
Implementations wise, it is pretty straightforward so I have chosen not to include any Haskell code.

1.3 Testing

With the parser, there is also a `Hunit` test suite, that can execute a number of defined tests based on assertions.
To run the test, simply load the testmodule `ParserTest.hs` and run the following command

```
ParserTest> runTestTT testsParser
```

The test cases covers various scenarios where the parser should fail and succeed, such as:

1. Precedence of operators
2. Associativity of operators
3. List comprehension

4. Functions calls
5. Programs of various sizes
6. Reserved keywords

I chose not to do property based testing with QuickCheck, as it proved to be too time consuming with my limited experience, but doing this could have revealed many quirky edge cases where my parser also fails. Unit testing can also be very time consuming if every case has to be covered, but does a good job of illustrating if common use cases (or obvious error) parses as they should.

1.4 Assessment

Compiling the parser with the flags: `-Wall -fno-warn-unused-do-bind` does not result in any warnings or errors, and running it with `hlint` produces 3 suggestions of reductions, but I have chosen to ignore these as they don't make much sense in this context.

The testing yielded no errors (that I know of!) in the parser, which is an indicator of that it generally works pretty well, as the parser correctly parses valid input. It accounts for precedence and associativity of operations and fails when it should. The code itself, is pretty readable and not too obscure, maybe apart from some function names (is explained in the comments, though).

As stated in the previous subsection, QuickChecking the program for edge cases would have been a stronger indicator of the robustness of the parser, but was not done due to the aforementioned reasons.

2 Subscript, Interpreter

The source code for the `SubsInterpreter.hs` discussed in this section can be found in appendix **??**. The associated test can be found in appendix **??**.

The interpreter can be run by running `runProg` with a given program (of the type `Program`) or in combination with the parser by running `runhaskell Subs.hs somefile.js` where `somefile.js` is the path to a file using the syntax described in the assignment.

The SubsM monad proved to be, quite like the Salsa monad from the weekly assignment, similar to the State monad, as we maintain some environments that we pass along to future computations. To construct a SubsM instance of type `a`, a function of the type

$$SubsM\ a \rightarrow Context \rightarrow Either\ Error\ (a,\ Env)$$

has to be provided. To deconstruct the instance, the function `runSubsM` wrapped inside the type constructor can be invoked, by providing a context.
The `Context` data type contains the variable and function environments, where the functions is a read-only environment containing the functions available for use, and the variable environment contains the value of the variables declared so far.

2.1 Implementation

Generally, the implementation of the interpreter was pretty straightforward, apart from the list comprehension expression, which I will describe in section **??**. The most relevant parts of the interpreter is as follows:

`evalExpr` This function evaluates some expression and returns a SubsM Value. Given some expressions, it recursively evaluates the expression until it reaches a leaf in the AST provided. The expressions can be e.g. some arithmetic operation, comparison operation or variable assignments. `evalExpr` can update the current environment if a legal assignment is performed. A more detailed description of my implementation of the list comprehension can be found in section **??**.

Failing scenarios: If one attempts to assign a variable which has not been declared before, call functions with the wrong types, wrong number of arguments, functions that does not exists or performing list comprehension on something that is not a string or list will result in an error.

`stm` In `stm` we have the option to either do a variable declaration, the only time where we add to the variable environment, or simply execute an expression, that can possibly update the variable environment. If a variable is declared with an expression following it, this is assigned to the given variable name, and if not the name is assigned `UndefinedVal`.

`runProg` This function takes a Subs program as input (program being a list of statements). We take the initial context and provide it to the `runSubsM` function along with a statement. This either generates an error or it succeeds. If it succeeds, we update the environment and repeat this for the next statement as well. This is repeated until there are no more statements in the program and if this succeeds without any errors, the resulting environment from performing these statements is returned.

If an error is encountered when interpreting the program, will stop at first error received, and return it. This error will most likely be a operator type error, or a failure to either locate a given function or variable in the environments.

2.2 List comprehension

The list comprehension proved to be the most difficult part of the interpreter to implement, so only a partial solution was provided, described in the following section.

2.2.1 Simplifications

The list comprehension in my implementation is a bit amputated compared to its intended behaviour. As of now, it only works for a single `for` expression and it does not allow any `if` statements as well, so in essence, the supported list comprehension is on the following form:

```
[ for (n of e1) e2]
```

Any additional `for` and `if` statements will simply be ignored at the moment, but they can still be included if one wishes to do so.

It works by first making a 'local' copy of the current environment, evaluating the list/string `e1` and then recursively looking through each element of `e1`, saving the values as a variable declaration with the ident `n` in the local environment. In case that `e1` contains a variable with ident `n`, `e1` can be evaluated to get the correct expression. The ident `n` will shadow previous declarations of same names, but as it resets af the list comprehension finshes, it will not mess with the original scope of the program.

An idea for implementing the `if` guards, would to just not include any values satisfying the predicate when computing the evaluated list/string.

2.3 Testing

The interpreter also comes with a `Hunit` testsuite, that can execute a number of defined tests based on assertions.
To run the test, simply load the testmodule `InteInterpreterTest.hs` and run the following command

```
InterpreterTest> runTestTT testsInterpreter
```

The test cases covers various scenarios where the interpreter should fail and succeed, such as:

1. List comprehension
2. Basic environment scopes
3. Programs of various sizes
4. Function calls

The test cases do not cover all the different combinations of expressions that can be made in general, but the unit tests give a pretty good overview of the capabilities of the implementation. They also don't quite cover all cases where type errors in the function calls occur, but a sample of them has been provided, in order to illustrate that they do in fact return an error.

I have also included a failing test case that shows that the implementation is not able to interpret more advanced list comprehension expressions correctly, but this is to be expected.

For the same reasons as stated in the parser, QuickCheck was not done for the interpreter.

2.4 Assessment

Testing showed that the interpreter generally works as intended, apart from the missing list comprehension implementation. The partial list comprehension implemented also works according to the simplifications made, correctly producing a resulting list by updating the environment locally, and resetting it to its original state, when finished.

It will, in case an error occurs, fail to interpret the statements, and return an error with a message informing the user what went wrong. The error types returned are as described previously.

3 Generic Replicated Server Library

The source code for the `gen_replicated.erl` and `replica.erl` discussed in this section can be found in appendix **??** and **??**. Their associated test can be found in appendix **??**

The implementation of the generic replicated server can be found the in file `gen_replicated.erl` with the corresponding callback module `replica.erl`.

3.1 Structure

The current implementation is not a complete one, according to the specification in the assignment text. The simplifications that I made, can be found in section **??**.

The `gen_server` is defined as a custom behaviour, which means that importing it will result in warnings if the functions: `handle_read/2`, `handle_write/2` and `init/0` are not implemented in the module.

The main structure of the actual implementation is as follows:

Coordinator The coordinator handles incoming requests from clients and forwards them to the replicas. When initialized, the coordinator initializes a state from a given callback module and recursively spawns the desired amount of replicas under its control, and keeps the `Pid`s in a list. It then enters a loop, waiting for incoming requests. The requests that the coordinator can receive is:

`begin_read` Informs the coordinator that it should begin some read request. The request is forwarded to an available replica, and it re-enters the receiving loop.

`begin_write` Informs the coordinator that it should begin some write request. The write request is forwarded to an available replica, and then enters a new loop, where it waits for a response from the writer replica. If the writer responds with `update` and a new state, the coordinator sends out a copy to all other replicas, telling them to update their own respective state and re-enters the main receiving loop. If it received `noupdate` no new state should be sent to the replicas, and it simply re-enters the main receiving loop. All other requests received while waiting for the writer to finish, is simply informed that the server is busy.

`done_read` Informs the coordinator that a replica has finished a read request, and it forwards the result to the client and re-enters the receiving loop again.

`stop` Informs the coordinator that is should terminate its state instance and stop looping. The coordinator will send the same message to all replicas, before terminating itself.

replica The replica server receives tasks delegated by the coordinator and it can be assigned to be either a reader or writer. Replicas are spawned by the coordinator with an associated callback module, and are all initialized in the same inital state (where the concrete state depends on the callback module implementation). When spawned, a replica enters a receiving loop where i can get one of the following requests:

`read` Informs the replica to do a read job with a given request. The replica calls `handle_read/2`, implemented by the callback module, where it can either receive a reply in form of a result or a `stop`. If it gets a result, it is sent back to the coordinator, which then returns it to the client. If it gets `stop` it tells the coordinator to stop, which will terminate all replicas and the coordinator.

`write` Informs the replica to do a write job with a given request. It will call `handle_write/2`, implemented by the callback module. This call can return 3 values: `noupdate` which wont do anything, and the replica returns to its main loop after notifying the coordinator it is finished writing, `update` which will cause the replica to update its state, and tell the coordinator to do the same before returning to its main loop, and `stop` which is as previously described.

`replicate` Informs the replica to update its state with a given copy, sent by the coordinator, and return to the main loop

`stop` Informs the replica to terminate its state and not loop any more.

Callback module The callback module (`replica.erl` in my case) is the module that does the actual concrete computations assigned by the coordinator. It implements the following functions:

`start/1` calls the `gen_replicated` function `start` which starts a new generic replicated server.

`init/0` Initializes a fresh state. My implementation uses a dictionary data structure, for storing data in form of key/value pairs.

`handle_read/2` Handles read requests. In my simple implementation of the callback module, it only accepts a single request, `tolist`, which converts the current dictionary state into a list and returns it.

`handle_write/2` Handles a write request. In my implementation it only supports the request {`add_to_dict, K, V`} where a key `K` will be added to the dictionary with the value `V`, if it doesn't exist (returns `update`) or simply nothing happens if it already does (returns `noupdate`).

The requests accepted by the functions `handle_read/2` and `handle_write/2`, can also contain an extra argument `Time` (i.e. {`tolist, Time`} and {`add_to_dict, K, V, Time`}), if we want the calls to sleep for `Time` amount of ms. This is only used for testing purposes though.

3.1.1 Simplifications

In my implementation i have made some few restrictions in order to simply the `gen_replicated` server:

1. No blocking while writing to the state (and no queue for incoming read requests while performing a write)
2. No waiting for readers to finish before starting a write operation
3. No error handling in general. If something goes wrong, the server more or less crash and burn (for example, by sending more requests than available replicas).

An implementation that would avoid starvation of readers and writers would be to maintain some form of queue that represents the operations in the order they were received. If a write request is received while reading, we should stop accepting any more reads while writing. If read requests were received while writing, these should be added to a queue and executed in the corresponding order, before allowing any write requests that comes after.

3.2 Testing

In the test file `genTest.erl` resides a few functions that makes some assertions, using the EUnit unit testing library. These tests checks that the basic read/write operations works as intended, by assigning a writer replica and a reader replica and confirming that it actually writes and reads. There is also a small test for testing the concurrency of the read operation. It works by starting an `timer:exit_after` operation which will throw an error after a specified amount of time, and then immediately spawning a number of processes which will make a sleeping read call. The time is chosen such that if the read operations was done sequentially, it would exceed and time out, but if done concurrently, the test will make the appropriate amount of assertions before timing out.

Running the tests can be done with the following commands:

```
c(replica),
c(gen_replicated),
c(genTests),
genTest:test().
```

Which returns that all (albeit only 4) tests passes.

3.3 Assessment

From what I have tested, basic reading and writing to a state works as intended, but can easily be broken if not limiting oneself to to using it according to the simplifications stated. Testing also concluded that concurrent read works as intended.

There is still some work left if the implementation is to be labelled as a robust system, but due to the fast approaching deadline, this was not prioritized.

4 AlzheimerDB

The source code for the `alzheimer.erl` and `dbtest.erl` discussed in this section can be found in appendix **??** and **??**. Their associated test can be found in appendix **??**

I chose the implement the `alzheimer` callback module, before I attempted to implement the `gen_replicated` module, which means that `alzheimer` uses the `gen_server` behaviour. For storage in the callback module, a simple dictionary was used, as the functionality provided by one is more than enough for my intended purpose, and it makes some operations, such as lookups, simple.

4.1 Error handling

The `alzheimers` module handles errors or exceptions that might occur by using `try ... catch` expressions where they should be properly handled, i.e. in the `handle_call` function. That is, when the function `F` for some reason throws a `throw` exception or when `P` throws any kind of exception. When such an exception is caught, the `alzheimer` module will then forward the error, to whatever module is calling it.

4.2 Querying efficiently

Doing several concurrent reads would have been ideal for a generic replicated server model, where each read request would be distributed to a replica server by the coordinator and then return the result, when done.
My implementation, as stated above, unfortunately uses the `gen_server` behaviour, so the concurrency handling would be like any Erlang process. If a `gen_server` receives multiple requests at the same time, they are placed in a processes message queue and processes handle their message one-by-one. My implementation, I realized a little bit too late, is not concurrent, and thus do not support concurrent reads. Concurrency when querying with `handle_call` could have been handled in conjunction with `handle_info` by sending a message back to the module where it would then spawn a new process to do the job.

4.3 Testing

Testing the `alzheimer` module, consisted of 2 parts. First to create a module that used `alzheimer.erl` and then test this module. Testing the module was done with unit tests, using the EUnit library.

Testing Program: For testing the `alzheimer` callback module, I have implemented another small module, `dbtest.erl`, that utilizes the `alzheimer` functions `query/2` and `upsert/3`. In `dbtest`, the function `insert/3` calls `upsert/3` with a `Pid`, a key `K` and a value `V` which then calls `upsert` with the predefined function `F`. `F` will then, based on its input, notify the alzheimers module whether to simply insert `V` in the database if it doesn't exist or tell it to modify it with `V` if it does. So no matter what, the dictionary will contain the value `V` after `insert/3` is executed.

The other function `getRowsError/2` call the `query/2` function, to obtain all the rows in the database that satisfies a predefined predicate `P`. It will simply check if a given `LookupVal` is equal to any values

in the dictionary and return true if they are and false if not. To also test that alzheimers catches an exception, `P` will throw and exception if any value in the dictionary is `throw_error`.

EUnit tests: In the test file `alzTest.erl` resides a some functions that makes some assertions, using the EUnit unit testing library. This includes assuring that `dbtest` correctly inserts/updates the values in the database, and correctly returns an error if an exception was thrown at some point.

Running the tests can be done with the following commands:

```
c(dbtest),
c(alzTest),
alzTest:test().
```

Where you will get the output:

```
All 6 tests passed.
```

4.4 Assessment

The `alzheimer` callback module generally works alright when doing performing the various `upsert` and `query` requests. It inserts/updates the database as it should with the correct key/value, and returns the correct results when querying it. The error handling, in general, also seems to work pretty well, since it catches exceptions raised by either `P` or `F` and returns the error to the user.

The callback module lacks concurrency, which means that it does not utilize the fact that query operations are read-only, which of course is unfortunate. If time allowed it, it could be implemented as briefly described before.

Testing with the `dbtest` module reveals that `alzheimer` throws errors and passes when supposed to. The test cases can be found in **??**.

A Source code

A.0.1 Parser

A.1 Interpreter

A.2 Generic replicated server

A.2.1 gen_replicated.erl

A.2.2 replica.erl

A.3 AlzheimerDB

A.3.1 alzheimer.erl

A.3.2 dbtest.erl

B Test files

B.1 Parser tests

B.1.1 ParserTest.hs

B.2 Interpreter tests

B.2.1 InterpreterTest.hs

B.3 Generic replicated server tests

B.3.1 genTests.erl

B.4 AlzheimerDB tests

B.4.1 alzTests.erl