Advanced Programming
Exam

Exam number 167

November 2024

1 Introduction

The functionality is in general as desired, and any minor problems are discussed in the subsections below. I have made tests for all the edge cases (and generic cases) that I could think of, I furthermore asked two LLM's (ChatGPT4o and Claude3-5-Sonnet) to find further edge cases that might not be handled, as well as using the `--enable-coverage` functionality to make sure all code-branches are checked. This does not guarantee all edge cases are handled but does reduce the likelihood of such existing (and not being handled correctly). I also have one issue when dealing with deadlocks/stuck computations, where I can only test that they timeout (after 3 seconds), and thus assume that they will be stuck forever, which I have deemed to be reasonable. Stuck tests are commented out in the test suite, but are left if the reader wants to ensure they are stuck.

1.1 Tuples

For the implementation of the tuples, I first made sure it could parse empty tuples and tuples with > 1 elements correctly, and also with or without projections. For the evaluation, I use a `mapM` to evaluate all the sub-expressions within the tuple. For the projection, I first make sure that it projects on a tuple, and that the index is positive but smaller than the length of the tuple. I have tested parsing and evaluating for many cases such as nested tuples, nested projections, long tuples, empty tuples, etc. I therefore deem that the base functionality of the `Tuples`/`Projection` to be correct.

1.2 Loops and stepping

For the implementation of the `for` and `while` loop I follow the evaluation implementation structure as presented in the assignment. However, in the `for` loop, it is not specified if `i` should be accessible within the body, but I assume that it should be accessible and thus tested it with

```
evalTest
   "For loop with 'i' in body"
```

```
    (ForLoop ("x", CstInt 0) ("i", CstInt 5) (Add (Var "x") (Var "i")))
    (ValInt 10)
```

For parsing, the two loops have the same prefix of `loop`, and therefore I needed to include a `try` within the parser as

```
try $ ForLoop -- try since the for loops were too greedy
       <$> ((,) <$> (lKeyword "loop" *> lVName) <*> (lString "=" *> pExp))
       <*> ((,) <$> (lKeyword "for" *> lVName) <*> (lString "<" *> pExp))
       <*> (lKeyword "do" *> pExp),
   WhileLoop
    <$> ((,) <$> (lKeyword "loop" *> lVName) <*> (lString "=" *> pExp))
    <*> (lKeyword "while" *> pExp)
    <*> (lKeyword "do" *> pExp)
```

For both parsing and the evaluating of the loops, I have tested the base functionality, and edge cases such as incomplete definitions, errors in the `bound`, `body` and `init`, false `bound`, negative `bound`, etc. I therefore deem the implementation to be functional within generic use cases.

1.3 `&&` and `||`

For the parsing of `&&` and `||` I first removed left recursion and ambiguity, which can be seen in question 1. To keep the hierarchy of the operators, I had to create separate parser groups for `&&` and `||`, with `||` having the lower precedents. Implementing the evaluation of the pure interpreter mainly requires controlling for any errors. As the evaluation order is not given, my implementation evaluates from left to right, and thus in e.g. `e1 || e2`, if `e1` succeeds, it will always be returned, as it "finishes" first.
Testing in the pure case, I mostly focused on parsing and basic evaluation, and more specifically the hierarchy of the operators is followed. As all the tests passed as expected, I deem the basic functionality of `&&` and `||` to be okay.

1.4 Simulated concurrent interpreter

For the simulated concurrent interpreter, I made the `step` function using a helper function `step'` that contains a boolean flag `stepDone`, such that I can make sure each (sub-)expression in `&&` and `||` are both only evaluated once. Further, the `&&` steps until both expressions are done, or if one fails and just returns that. For `||` it steps until one is done first, and returns that. There is however an issue with `e1 || e2`, in that when stepping, it steps both `e1` and `e2` and then looks to see if any finishes, thus if `e1` finished the next step of `e2` should not have happened. This could be fixed by first stepping `e1` and then casing on that result, but that gives rise to a new issue: If `e1` gets stuck e.g. in an infinite `KvGet` the other expression `e2` is never able to evaluate and return. This could in turn also be fixed with other mechanisms (such as looking for progress in the expressions) but was deemed to be unnecessary for the time being, but worth keeping in mind. With the minor problem mentioned in mind, the implementation does work for all other edge cases I could find, and thus I deem at least the general functionality of the implementation to be okay.

1.5 KVDB

For the KVDB I implemented the API, where the main implementation is the serverLoop, which is spawned using the genServer module. The serverLoop handles the KvGet and KvPut by defining the messages the server can receive to be GetMsg for KvGet and PutMsg for KvPut. The serverLoop then cases on that, such that for the GetMsg we simply give back the value if the requested key is in the DB. If the key is not yet put, we save it in a list of pending requests. This leads to the PutMsg, that whenever a key-value is inserted, we look if there are any pending requests for that key, and if so, send the value to them, and then remove them from the pending requests. I made several test cases to ensure the basic functionality worked, as well as blocking of threads when kvGet does not receive a value, which I tested like

```
testCase "kvGet blocks a thread if the key is not yet present" $ do
   db <- startKVDB :: IO (KVDB Int String)
   resultVar <- newEmptyMVar
   resultVar2 <- newEmptyMVar
   _ <- forkIO $ do
      val <- kvGet db 2
      putMVar resultVar val
      kvPut db 3 "three"
   _ <- forkIO $ do
      val <- kvGet db 3
      putMVar resultVar2 val
   threadDelay 1000000
   isEmpty <- isEmptyMVar resultVar2
   isEmpty @?= True
   kvPut db 2 "two"
   threadDelay 100000
   val <- takeMVar resultVar
   val @?= "two"
   val2 <- takeMVar resultVar2
   val2 @?= "three"
```

This test ensures that a thread can not continue until kvGet gets its value. This could lead to infinite loops and could be avoided by e.g. implementing a timeout in either the server loop or directly in the kvGet, but as we wanted the kvGet to block, I decided not to implement this.

1.6 Concurrent interpreter

The concurrent interpreter is implemented using the jobAdd function from the SPC to evaluate the && and || concurrently. Both && keep waiting until the SPC reports back that both the jobs are done, and then returns that. The || simply waits until one expression is successful, and then returns that. The killing of threads is discussed in question 7.
The implementation is tested for basic functionality, such as handling failures,

and that all the normal computations work. I also test the concurrency with regard to deadlocks, which are discussed in question 7. I thus deem the implementation to work for the given cases I could find.

2 Questions

2.1 1

The full grammar represents the parser with no ambiguity and the left recursion removed can be seen below:

```
Exp := Exp1

Exp1 := Exp2 Exp1'
Exp1' := "||" Exp2 Exp1' | (* empty *)

Exp2 := Exp3 Exp2'
Exp2' := "&&" Exp3 Exp2' | (* empty *)

Exp3 := Exp4 Exp3'
Exp3' := "==" Exp4 Exp3' | (* empty *)

Exp4 := Exp5 Exp4'
Exp4' := "+" Exp5 Exp4' | "-" Exp5 Exp4' | (* empty *)

Exp5 := LExp Exp5'
Exp5' := "*" LExp Exp5' | "/" LExp Exp5' | (* empty *)

LExp := "if" Exp "then" Exp "else" Exp
      | "\\" var "->" Exp
      | "let" var "=" Exp "in" Exp
      | "loop" var "=" Exp "for" var "<" Exp "do" Exp
      | "loop" var "=" Exp "while" Exp "do" Exp
      | FExp

FExp := Atom FExp'
FExp' := Atom FExp' | "." int FExp' | (* empty *)

Atom := baseAtom Atom'
Atom' := "." int Atom' |(* empty *)

baseAtom := var
          | int
          | bool
          | "(" ")"
          | "(" Exp ")"
```

```
          | "(" Exp Exps ")"
          | "put" Atom Atom
          | "get" Atom

Exps := "," Exp Exps | "," Exp
```

I use helper non-terminals like `Exp1'` to avoid left recursion. In the implementation of the parser, it follows the grammar by making separate parsers for each priority group, such that e.g. `||` has the lowest priority and `*` & `/` has the highest operator priority. The grammar does not explicitly denote white-space handling and that variable names must not be a keyword, but the parser is implemented to handle that.

2.2 2

In my implementation of the `Tuple` case in `eval`, I use the `mapM` to evaluate the expressions, thus they are evaluated left to right (as `mapM` goes left to right), as can be seen in the code:

```
eval (Tuple es) = do
  vals <- mapM eval es
  pure $ ValTuple vals
```

To make sure the evaluation order of tuples is left from right, such that in `(e1, e2)` `e1` is evaluated before `e2`. I made some tests where `e1` depends on `e2` and where `e2` depends on `e1`. I further made longer tuples like `(e1,...,e5)`, where the next expression depends on the previous. I make the dependencies using `KvGet` and `KvPut`. As an example see below from the pure interpreter:

```
evalTestFail
    "Order of evaluation in tuple (KvGet before KvPut)"
    (Tuple [KvGet (CstInt 1), KvPut (CstInt 1) (CstInt 2)]),
evalTest
    "Order of evaluation in tuple (KvPut before KvGet)"
    (Tuple [KvPut (CstInt 1) (CstInt 2), KvGet (CstInt 1)])
    (ValTuple [ValInt 2, ValInt 2])
```

2.3 3

Regarding the normal operations, the implementation I have of the evaluation method `eval` all errors regarding operations are handled within the `evalM` monad and captured using `failure` (such as `failure "Division by zero"`), thus they result in an `ErrorOp` and returns `Left e` and not as an exception, and the only way a job is marked as `DoneCrashed` is when an exception is being thrown within the job, as we can see below

```
let doJob = do
      jobAction job
      send (spcChan state) $ MsgJobDone jobid
```

```
  onException :: SomeException -> IO ()
  onException _ =
    send (spcChan state) $ MsgJobCrashed jobid
```

Regarding using the IO functions, such as `KvGet` and `KvPut`, the implementation of the KVDB does not throw any exceptions, and will e.g. on an unknown key wait, and not throw any exception. And the way `IORef` is used is controlled, like in the `BothOfOp` I make sure to initialize the `newIORef` before writing or reading to it. However given we are in the inherent chaotic IO, there might be some unforeseen errors leading to an exception, but not something I would deem as likely or often to happen.

2.4 4

In my implementation of the simulated concurrent interpreter, using a `KvGet` on a key never put, or if the put is dependent on the `KvGet` we end in a deadlock. However, since the simulated interpreter is deterministic, and since we know how the implementation is made, we can quite easily determine if an expression will be deadlocked. Some are more trivial than others to notice, e.g. in the following example where there is only one `KvGet`:

```
evalTestFail
  "Test of deadlock with only one KvGet"
  (KvGet (CstInt 1))
```

Another quite trivial example is when two `KvGet`'s are dependent on each other like in this case

```
evalTestFail
  "Test BothOf deadlock with conditional KvGet and KvPut"
  (BothOf
    (If (Eql (KvGet (CstInt 1)) (CstInt 2))
        (KvPut (CstInt 2) (CstInt 1))
        (CstInt 0))
      (If (Eql (KvGet (CstInt 2)) (CstInt 1))
        (KvPut (CstInt 1) (CstInt 2))
        (CstInt 0))
    )
```

We know that both `KvGet`'s can never get the values, since the put for the key they're waiting for is dependent on the other, and thus stuck forever. One that is a bit more tricky to spot is this example

```
evalTestFail
  "Nested BothOf with conditional KvPut that results in deadlock"
  (BothOf -- 1
    (BothOf -- 2
      (If (Eql (KvGet (CstInt 1))(CstInt 1)) -- 3
        (KvPut (CstInt 2) (CstInt 1)) -- 4
        (CstInt 0)) -- 5
      (KvPut (CstInt 1) (CstInt 1))) -- 6
    (BothOf -- 7
      (KvPut (CstInt 1) (CstInt 2)) -- 8
```

```
      (KvGet (CstInt 2))) -- 9
  )
```

Here we see that `9` depends on `4`, and line `4` is only run if the `KvGet (CstInt 1)` in `3` is equal to 1. There are two `KvPut` with a key of 1 but with different values: `6` & `8`. Thus if need to have the condition in `3` to be true, we need to have `6` executed and then `3`. In this simulated case, we will always have the condition false, and thus we end with `9` being stuck. However, if we e.g. just swap `3-5` with `6` to get this:

```
evalTest
  "Nested BothOf with conditional KvPut that never results in deadlock (3-5 and
      6 swap)"
  (BothOf -- 1
    (BothOf -- 2
      (KvPut (CstInt 1) (CstInt 1)) -- 6
      (If (Eql (KvGet (CstInt 1)) (CstInt 1)) -- 3
          (KvPut (CstInt 2) (CstInt 1)) -- 4
          (CstInt 0)) -- 5
    )
    (BothOf -- 7
      (KvPut (CstInt 1) (CstInt 2)) -- 8
      (KvGet (CstInt 2))) -- 9
  )
  (ValTuple [ValTuple [ValInt 1, ValInt 1], ValTuple [ValInt 2, ValInt 1]])
```

This test never gets into a deadlock and will pass (with slightly different results although). So as long as you know the deterministic order of evaluation, it is easy to determine any deadlocks.

2.5 5

Yes, the concurrent interpreter can go into a deadlock, but it is non-deterministic due to the somewhat unpredictable concurrency. Of course, we have the deadlocks that are trivial as with the simulated interpreter, e.g. when a `KvGet` does not have a corresponding texttKvPut, or if we try to `KvPut` which is dependent on the corresponding `KvGet`. These are easy to find, as they don't necessarily depend on the order of evaluation. But as we saw from a test case from the previous question:

```
evalTest
  "Nested BothOf with conditional KvPut that sometimes results in deadlock"
  (BothOf -- 1
    (BothOf -- 2
      (If (Eql (KvGet (CstInt 1)) (CstInt 1)) -- 3
          (KvPut (CstInt 2) (CstInt 1)) -- 4
          (CstInt 0)) -- 5
      (KvPut (CstInt 1) (CstInt 1)) -- 6
    )
    (BothOf -- 7
      (KvPut (CstInt 1) (CstInt 2)) -- 8
      (KvGet (CstInt 2))) -- 9
  )
  (ValTuple [ValTuple [ValInt 1, ValInt 1], ValTuple [ValInt 2, ValInt 1]])
```

In this case, the order of evaluation mattered about it going into a deadlock, as discussed in the last question. But as we don't exactly know which are evaluated in what order, we have a non-deterministic case that might lead to a deadlock. So in the cases where the order of evaluation matters, it can be difficult and/or impossible to know if it goes into a deadlock.

2.6 6

In my simulated concurrent implementation of `e1 || e2` I use a helper function `step'` to recursively step each nested expression in the computation. In the case of a `OneOfOp`, the function looks to see if `e1` or `e2` is done with their evaluations, and if so, it just calls itself recursively with only that expression, and therefore effectively discards the other expression. In the following code snippet we see the case where `e1` is done (there is a similar case for when `e2` is done discarding `e1`):

```
in case (e1', e2') of
  (Pure v1, _) ->
    let m' = c v1
    in step' stepDone' env state' m'
```

And since the infinite expression is no longer in the computation, we no longer use any resources on the infinite loop in `e2`.

2.7 7

In my concurrent implementation of `e1 || e2`, I utilize the SPC's `jobAdd` function that creates separate threads to evaluate the two expressions concurrently. Thus if `e1` finishes first, `e2` will keep getting evaluated if not explicitly stopped. Thus in my code, whenever `e1` or `e2` finishes, I call SPC's `cancelJob` on the other expression job-id to cancel it and kill the thread evaluating the expression. Whenever a job is done, I case on the result, and if it succeeds it ends in the following case:

```
case result of
  Just (Right v) -> do
    let otherJobId = if doneJobId == jobId1 then jobId2 else jobId1
    jobCancel spc otherJobId
    runEvalM spc kvdb env (c v)
```

This makes sure to explicitly stop the possible infinite loop in `e2`, and we therefore don't use any resources after at least one of the expressions is evaluated.