Exam Report
Advanced Programming 2024

November 7, 2024

1 Introduction

I completed all the exam tasks and implemented all the extension requirements of the APL language. I extended parser and evaluator for tuple construction and projection operations, for/while loop structures, and concurrent operators (&&, ||). I implemented three interpreters, including pure functional interpreter, simulated concurrent interpreter and concurrent interpreter. I implemented concurrent key-value database (KVDB) to support for concurrent interpreter.

During the implementation, the main ambiguity that I encountered was the state transfer in the simulated concurrent interpreter when the step function handles `BothOfOp` and `OneOfOp`. First, there is a clear principle when dealing with concurrent operations: the state generated by `step m1` should be used in `step m2`. This ensures that the state changes generated by m1 can be used when executing m2, such as the key-value pairs generated by the put operation in m1. For `BothOfOp`, when one of the executions fails, it is necessary to decide which state to use as the state for error propagation: the initial state, the state of successful execution, or the state after two executions? I chose to use the state after two executions, which ensures that even if one execution fails, the state changes generated by the other execution will not be lost. Similarly, for `OneOfOp`, no matter which execution succeeds (obtaining `Pure`), I choose to use the state updated after two executions to perform subsequent executions, to maintain the continuity of the state. In the pure interpreter, the state transfer of `BothOf` and `OneOf` is performed in the order of evaluation. `BothOf` evaluates m1 first and then uses the state updated by m1 to evaluate m2; `OneOf` also evaluates m1 first, and if it fails, it uses the state after executing m1 to continue evaluating m2. This state transfer method matches its sequential execution characteristics. In the concurrent interpreter, since the state is saved in the KVDB, there is no state selection problem.

Another ambiguity is that it is stated in the exam text that: to evaluate the expression `e1 && e2` we evaluate e1 and e2 in unspecified order, then return a pair of their results. In the pure interpreter, when `e1 = get 0`, `e2 = put 0 true`, the order in which e1 and e2 are evaluated has an impact on the final result, if we evaluate e2 first, then the result will be `ValTuple [ValBool True, ValBool True]`, if we evaluate e1 first, an error will occur. Finally, I chose to evaluate e1 first and then evaluate e2, because pure interpreter does not contain concurrent execution, only sequential execution.

In addition, although not explicitly required in the exam text, I also implemented parsing of nested tuple projection operations, such as `((e0,e1),(e2, e3)).0.0`. In my implementation, empty tuples and tuples with two or more elements are legal, and tuples with only one element will be parsed as parenthesized exp.

My implementation of KVDB does not use read-write locks. Otherwise, programs like `BothOf (KvGet 0) (KvPut 0 True)` will block due to mutual exclusion.

I wrote corresponding test cases for the parser, the three interpreters, and KVDB. These test cases fully covered their main functions. All components passed these tests successfully. Although no special tests were written for the evaluator, since the tests of the three interpreters need to be implemented by calling `eval`, the result of these tests also indirectly verifies the correctness of the evaluator. Therefore, it can be considered that all parts of my implementation are functionally correct.

2 Q&A

2.1 Show the final APL grammar with all ambiguities and left recursion removed, and explain to which extent it resembles your parser implementation.

My parser implementation has the same grammar structure as the final APL grammar after eliminating left recursion and ambiguities. I added priority levels for `OneOf` and `BothOf`, and implemented the priority of && and || through the parser chain of `pOneOf -> pBothOf -> pExp1 -> ...`. I used the chain method and recursive calls (such as `pBothOf' $ BothOf e1 e2`) to implement the left associativity of && and ||, and support parsing consecutive && and || operations. I handled tuples and projects in `pAtom`. `pTuple` parses tuples of at least two elements `(e1, e2, ...)`, and `pTupleProject` supports consecutive projection operations `e.1.2` through recursion.

```
Exp    ::= OneOf

OneOf   ::= BothOf OneOf'
OneOf' ::= "||" BothOf OneOf'
        | epsilon

BothOf ::= Eq BothOf'
BothOf'::= "&&" Eq BothOf'
        | epsilon

Eq     ::= Add Eq'
Eq'    ::= "==" Add Eq'
        | epsilon

Add    ::= Mul Add'
Add'   ::= "+" Mul Add'
        | "-" Mul Add'
        | epsilon

Mul    ::= LExp Mul'
Mul'   ::= "*" LExp Mul'
        | "/" LExp Mul'
        | epsilon

LExp   ::= FExp
        | "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   ::= Atom FExp'
FExp'  ::= Atom FExp'
        | epsilon

Atom   ::= var
        | int
        | bool
        | "()"                 -- empty tuple
        | "(" Exp ")"          -- parenthesized expression
        | "(" TupleExps ")"    -- tuple
        | "put" Atom Atom
        | "get" Atom
        | AtomProject

TupleExps ::= Exp "," Exp TupleExps'
TupleExps'::= "," Exp TupleExps'
           | epsilon

AtomProject ::= Atom AtomProject'
AtomProject'::= "." int AtomProject'
          | epsilon
```

2.2 Which of your tests demonstrate the evaluation order of tuples, and how?

I used these 2 tests to check the evaluation order of tuples. In each test, the first tuple element is a `KvPut` with a key $k1$, the second tuple element is a let binding and needs to evaluate a `KvGet` with $k1$ and another `KvPut` with a key $k2$, the third tuple element is a `KvGet` with $k2$. In the first test $k1 \neq k2$, and it shows that the evaluation order is from left to right: the first `KvPut` must be executed first, otherwise the subsequent `KvGet` cannot obtain the correct value. In the second test $k1 = k2$, not only verifies the order of tuple evaluation, but also verifies the correctness of the state update.

```
   , evalTest "Tuple evaluation order with key-value dependencies"
        (Tuple
          [ KvPut (CstInt 1) (CstInt 10)
          , Let "x"
              (KvGet (CstInt 1))
              (KvPut (CstInt 2) (Var "x"))
          , KvGet (CstInt 2)
          ])
        (ValTuple [ValInt 10, ValInt 10, ValInt 10])

    , evalTest "Tuple evaluation order with key-value dependencies and updates"
        (Tuple
          [ KvPut (CstInt 1) (CstInt 100)
          , Let "x"
              (KvGet (CstInt 1))
              (KvPut (CstInt 1) (Add (Var "x") (CstInt 1)))
          , KvGet (CstInt 1)
          ])
        (ValTuple [ValInt 100, ValInt 101, ValInt 101])
```

2.3 With the way you use SPC in the concurrent interpreter (APL.InterpConcurrent), is it possible for a job to crash (i.e., finish with DoneCrashed)?

In my implementation, the job may finish with `DoneCrashed`. For example, when `writeIORef ref result` is executed, since the `IORef` operation may fail, the SPC will catch the exception and end in `DoneCrashed`. But I have handled this situation by checking `JobDoneReason` returned by `jobWaitAny`: in `BothOf`, I require both executions to be completed with `Done`, otherwise the error "Not both tasks done" is returned; in `OneOf`, if the first completed execution does not finish with `Done`, it will wait for the second execution, and if the second execution also fails to end normally (`Done`),the error "Both tasks not done" is returned. In this way, I can distinguish between normal completion and abnormal crash, and provide a suitable error handling mechanism.

2.4 Is it possible for the simulated concurrent interpreter (APL.InterpSim) to go into a deadlock due to get/put? Is this easy to detect in some cases? Is it easy to detect in all cases?

It is possible for the simulated concurrent interpreter to go into a deadlock due to get/put. For example:

```
evalTest
  "Dead␣lock␣1"
  (BothOf
    (Tuple [KvGet (CstInt 0), KvPut (CstInt 1) (CstInt 2)])
    (Tuple [KvGet (CstInt 1), KvPut (CstInt 0) (CstInt 1)]))
  (ValTuple [ValTuple [ValInt 1, ValInt 2],
             ValTuple [ValInt 2, ValInt 1]])
```

In this case, the first branch executes `KvGet (CstInt 0)`, stuck, and cannot execute subsequent `KvPut`. The second branch executes `KvGet (CstInt 1)`, stuck, and cannot execute subsequent `KvPut`. Since the tuple is evaluated from left to right, the two branches each wait for the other's `KvPut`, resulting in a deadlock and program timeout. In simulated concurrent interpreter, deadlock manifests itself as computation stagnation. The deadlock is deterministic because the execution order is controlled in the `step` function.

It is easier to detect deadlock when there is an obvious circular wait between branches. When multiple nested `BothOf`/`OneOf` and complex programs are involved, deadlock detection becomes difficult. One way to help in all cases is to add a counter in the `step` function. If an execution does not progress for too many steps, it may be a deadlock.

2.5 Is it possible for the concurrent interpreter (APL. InterpConcurrent) to go into a deadlock due to get/put? Is this easy to detect in some cases? Is it easy to detect in all cases?

It is possible for the concurrent interpreter to go into a deadlock due to get/put. The example 2.4 used in the previous question also applies in this question. The difference is that in concurrent interpreter, deadlock manifests itself as thread blocking.

Deadlocks are easier to detect when they are caused by obvious circular waits. However, since the execution order of threads is affected by SPC scheduling, the deadlock state may be non-deterministic. Deadlock detection needs to rely on the runtime detection. In complex concurrent scenarios, deadlocks may be intermittent and timing-dependent, making deadlock detection difficult and not easily detectable in all cases.

In my opinion, the example 2.4 is more like a thread starvation than a deadlock, since there is no mutual exclusive access to resources, and there is no situation where threads are waiting for each other to hold resources. The operation `KvGet` waits for a certain value to be written. If other threads put the correct value, these operations can continue to execute.

2.6 With your simulated concurrent interpreter, when evaluating an expression e1 || e2 where e1 finishes quickly and e2 goes into an infinite loop, will e2 continue consuming resource indefinitely? If yes, how could this be fixed?

e2 will not continue consuming resource indefinitely. In my implementation:

```
step _ s (Free (StepOp m)) = (m, s)
step r s (Free (OneOfOp m1 m2 k)) =
  let (m1', s1) = step r s m1
  in case m1' of
    Pure v1 -> step r s1 (k v1)
    _ -> let (m2', s2) = step r s1 m2
         in case m2' of
              Pure v2 -> step r s2 (k v2)
              _ -> (Free $ OneOfOp m1' m2' k, s2)
```

The `step` function check before execution, first try to execute one step on m1, if it is found that m1 has been completed (`Pure v1` is obtained), it will directly step on this result for subsequent execution, and will not execute m2 at all. This can prevent continuing to execute the calculation that may be an infinite loop when there is already a result. And when encountering `StepOp m`, the `step` function will directly return the continuation `m` without further evaluation to ensure only execute at most one `StepOp` effect. This also ensures that infinite loops will not consume resources indefinitely.

2.7 With your concurrent interpreter, when evaluating an expression e1 || e2 where e1 finishes quickly and e2 goes into an infinite loop, will e2 continue consuming resource indefinitely? If yes, how could this be fixed?

e2 will not continue consuming resource indefinitely. In my implementation:

```
    interpConcurrent' env (Free (OneOfOp m1 m2 k)) = do
        ref1 <- newIORef (Left "Task 1 not done")
        ref2 <- newIORef (Left "Task 2 not done")
```

```
jobid1 <- jobAdd spc $ Job $ do
    result1 <- interpConcurrent kvdb spc m1
    writeIORef ref1 result1
jobid2 <- jobAdd spc $ Job $ do
    result2 <- interpConcurrent kvdb spc m2
    writeIORef ref2 result2
(finishedJobId, reason1) <- jobWaitAny spc [jobid1, jobid2]
let (resultRef, runningJobId) = if finishedJobId == jobid1
    then (ref1, jobid2)
    else (ref2, jobid1)
let runningRef = if finishedJobId == jobid1 then ref2 else ref1
firstResult <- readIORef resultRef
case (firstResult, reason1) of
    (Right v, Done) -> do
        jobCancel spc runningJobId
        interpConcurrent' env (k v)
    _ -> do
        (_, reason2) <- jobWaitAny spc [runningJobId]
        case reason2 of
            Done -> do
                secondResult <- readIORef runningRef
                case secondResult of
                    Right v -> interpConcurrent' env (k v)
                    Left e -> pure $ Left e
            _ -> pure $ Left "Both tasks not done"
```

I check `firstResult` and `reason1` to determine whether a job is completed successfully, if it is completed successfully, I will use `jobCancel spc runningJobId` to cancel another running job. Even if e2 is an infinite loop, it will be terminated by SPC's cancellation function and stop consuming resources. Only when the first completed job fails will it wait for the second job to complete.

3 Reference

I used the AI tool Cursor to generate some of the test cases. The generated tests are carefully checked to ensure the reliability.