Advanced Programming 2024

Exam Problem

Synopsis: Extending APL with programming constructs for concurrent and nondeterministic programming, executed both with simulation and multi-threading.

Preamble

This document consists of 17 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `cabal test` (i.e. include the non-modified parts of the handout as well).

You are expected to upload *two* files in total for the entire exam. The report must have the specific structure outlined in section 4.

Make sure to read the entire text before starting your work. There are useful hints at the end.

The following rules apply to the exam.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- Do not share or discuss your solution with anyone else until the exam is finished. Note that some students have received a timeline extension. Do not discuss the exam until an announcement has been made on Absalon that the exam is over.
- Posting your solution publicly, including in public Git repositories, is in violation of the rules concerning plagiarism.
- If you believe there is an error or inconsistency in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.
- The specification may have some intentional ambiguities in order for you to demonstrate your problem solving skills. These are not errors. Your report should state how you resolved them.

- Your solution to the exam problem is evaluated holistically. You are graded based on how well you demonstrate your mastery of the learning goals stated in the course description. You are also evaluated based on the elegance and style of your solution, including compiler warnings, inconsistent indentation, whether you include unnecessary code or files in your submission, and so on.

1 Introduction

The starting point for this exam will be a slightly cut down variant of the *AP Language* (APL) that was the topic of several assignments this year. The most significant simplification is that printing is not supported. The `localEnv` function is also implemented in a more direct way.

Your overall goal is to extend APL with various new features foccused on support for concurrency. This is partitioned into several tasks. Some of the tasks have dependencies on each other, which we will note explicitly. If you find yourself struggling with a task, consider attempting another (non-dependent) task. You do not need to solve (or even attempt) all tasks in order to pass, although a good performance across all tasks is necessary for a top grade.

The full ambiguous grammar, including features to be introduced later in this text, is shown in fig. 1, with disambiguation by the operator priorities listed in table 1. The new language constructs you will implement are these: pairs, `fst`, `get`, `loop`, `&&`, and `||`. The semantics of these language constructs will be discussed in the related tasks.

You are given a code handout with a complete AST definition and a partial implementation of a parser and an evaluator. The code handout corresponds roughly to the features developed during the course exercises. The evaluator expresses evaluation through the free monad `EvalM`, for which you will write three interpreters:

`APL.InterpPure`: the *pure interpreter*, which simply executes the given program sequentially and straightforwardly. Most of this interpreter is already complete in the code handout, although you are asked to make some extensions.

`APL.InterpSim`: the *simulated concurrent interpreter*, which simulates concurrency without using `IO`. You are asked to implement this interpreter in task D.

`APL.InterpConcurrent`: the *concurrent interpreter*, which uses true `IO`-based concurrency, based on the SPC job scheduler.

You do not need to define new effect types—these are already included in the handout. The code handout also includes a complete definition of a slightly simplified version of the SPC job scheduler (which you must

Operators	Associativity
* /	Left
+ -	Left
==	Left
&&	Left
||	Left

Table 1: APL table of operators in decreasing order of priority.

not modify), as well as some other skeleton files that you will extend as part of the exam tasks.

You should read the code handout carefully. It contains helpful comments. The handout contains a small collection of tests, of which some will initially fail. A comment connected to each test will mention after which task the test is supposed to work. You are *strongly* advised to add more tests of your own.

Non-normative examples of APL parsing and evaluation can be found in appendix A. Consult these if you find the semantics of some of the language constructs unclear.

When the semantics for a language construct states that something "is an error", it means that you must report an appropriate runtime error if that situation occurs (similar to division by zero). No specific error message is required.

```
Atom ::= var
       | int
       | bool
       | "(" ")"
       | "(" Exp ")"
       | "(" Exp Exps ")"
       | "put" Atom Atom
       | "get" Atom
       | Atom "." int

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

FExp ::= FExp FExp
       | Atom

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

Exp ::= Atom
      | Exp "==" Exp
      | Exp "+" Exp
      | Exp "-" Exp
      | Exp "*" Exp
      | Exp "/" Exp
      | Exp "&&" Exp
      | Exp "||" Exp
```

Figure 1: APL syntax in EBNF. This grammar is ambiguous and contains left recursion, which you will be asked to address. See table 1 for operator priority and associativity. The new language constructs are described in the corresponding tasks. Whitespace is permitted between all terminals.

2 Tasks

Task A: Implement tuples

This task depends on no other tasks.

For this task you must implement *tuples* in APL. A tuple is a value that comprises zero or more values. They are defined as follows in `APL.Monad`:

```
data Val
  = ...
  | ValTuple [Val]
```

Tuples are constructed with the syntax given in fig. 1 by the production

```
"(" ")"
```

corresponding to a tuple with no elements, and by the production

```
"(" Exp "," Exps ")"
```

corresponding to a tuple with two or more elements. Single-element tuples are thus not syntactically legal (just as in Haskell), although the AST representation can express them—the production `"(" Exp ")"` in fig. 1 is a parenthesized expression and not a single-element tuple.

The corresponding AST constructor is defined in `APL.AST` as follows:

```
data Exp
  = ...
  | Tuple [Exp]
```

For example, the input `(a, b)` corresponds to the following `Exp`:

```
Tuple [Var "a", Var "b"]
```

A tuple expression $(e_0, \ldots, e_{N-1})$ is evaluated by evaluating the components from left to right, then constructing a `ValTuple` with the resulting values in the same order as their corresponding expressions. If the evaluation of any expression fails, the result of the evaluation of the tuple also fails and the first error encountered should be reported.

The elements of a tuple `x` can be projected (accessed) with the syntax `x.i`, where `i` is a literal integer, such as in `x.0`, which projects element

0 of the tuple. For example, `(`$\texttt{e}_0$`,...,`$\texttt{e}_{N-1}$`).0` = $\texttt{e}_0$. If `x` has N elements, then `i` must be between 0 and $N-1$; using an index outside of this range is an error. The corresponding AST constructor is `Project`. It is an error to try to project an element from a non-tuple.

Your task: Implement parsing of tuple construction and projection in `APL.Parser` and evaluation of tuples and projections in `APL.Eval`.

Task B: Implement `loop` and stepping

This task depends on no other tasks.

For this task you must augment APL with two constructs for sequential looping. A `for`-loop expression

```
loop p = init for i < bound do body
```

is represented with the `ForLoop` AST constructor and is evaluated as follows:

1. Evaluate expression `init` to a value v.
2. Evaluate expression `bound` to an integer n. It is an error if `bound` does not evaluate to an integer.
3. Bind variable `i` to the integer 0.
4. Bind variable `p` to v.
5. While `i` $< n$: evaluate expression `body` and bind `p` to the result. Increment `i`.
6. Return the final value of `p`.

Essentially, `for`-loops evaluate the given `body` expression a number of times given by `bound`, each time binding the loop parameter (`p` above) to the result of the body.

A `while`-loop expression

```
loop p = init while cond do body
```

is represented with the `WhileLoop` AST constructor and is evaluated as follows:

1. Evaluate expression `init` to a value v.
2. Bind variable `p` to v.
3. Evaluate expression `cond` to a boolean c. It is an error if c does not evaluate to a boolean. If c is false, the loop stops and returns the value of `p`. Otherwise, if c is true, evaluate expression `body` and bind `p` to the result.

Note that in a `while`-loop `p` is in scope when evaluating `cond`, but in a `for`-loop `p` is not in scope when evaluating `bound`.

Stepping is an effect represented by the `StepOp` constructor of `EvalOp`. In later tasks we will use it to interrupt computations that do not otherwise have any effects. For `APL.InterpPure`, interpretation of `StepOp` is simply by recursively executing its continuation. For now, you must modify `APL.Eval` to use the `StepOp` effect (via `evalStep`) in the following cases: just before a loop body is executed (for both `for`- and `while`-loops), and just before a function value is applied.

Your task: Implement parsing of loops in `APL.Parser` and evaluation in `APL.Eval`. Treat `loop`, `for`, `do` and `while` as keywords. Add uses of `evalStep` as stated above. Extend `APL.InterpPure` to handle the `StepOp` effect.

Task C: Implement && and ||

This task depends on no other tasks.

For this task you must implement operators intended for concurrent programming.

The expression `e1 && e2` denotes concurrent execution of two expressions, returning the result of both. To evaluate the expression `e1 && e2` we evaluate `e1` and `e2` in unspecified order, then return a pair of their results. If either subexpression fails, then the overall expression also fails. If both fail, either error may be reported. Using the pure interpreter, `e1 && e2` is similar to `(e1,e2)` (except for evaluation order), but it will behave differently when using the concurrent interpreters.

The expression `e1 || e2` denotes concurrent execution of two expressions, returning the result of the one that finishes "first" (the meaning of which depends on the interpreter). To evaluate the expression `e1 || e2` we evaluate either `e1` or `e2` and return one of their results. The pure interpreter must start by evaluating `e1`, returning its value if evaluation succeeds. If `e1` fails, the pure interpreter should return the result of evaluating `e2`.

Your task: Transform the grammar of fig. 1 to eliminate left recursion and ambiguity. Then implement parsing of `&&` and `||` in `APL.Parser` and evaluation in `APL.Eval`. Evaluation of `&&` is with the `BothOfOp` effect (`evalBothOf`), and evaluation of `||` is with `OneOfOp` (`evalOneOf`).

Extend `APL.InterpPure` to handle the `BothOfOp` and `OneOfOp` effects. For `OneOfOp`, execute the first computation and return its value. If execution of the first computation results in an error, execute the second computation and return its value. If the second computation also results in an error, then the overall computation fails with either the first or second error message.

Task D: Implement a simulated concurrent interpreter

This task depends on task C.

For this task you must implement an interpretation function for `EvalM` that simulates concurrent execution. The main feature will be that `e1 || e2` can be implemented such that the expression that finishes *first* (i.e., in the shortest number of steps) has its value returned. As a special case, it means that `e1 || e2` can terminate even if one of `e1` or `e2` is an infinite loop (but not both).

The key idea is writing a function `step` that evaluates a `StepM` computation up to (and including) the next `StepOp` effect if possible *but no further*. By repeatedly "stepping", we can progress a computation arbitrarily. But importantly, we can interleave steppings of different concurrent computations.

Further, we also refine how the `KvGetOp`/`KvPutOp` effects work in a concurrent setting. When a `KvGetOp` effect requests the value of a key that does not exist in the state, then that is not an error. Rather, execution is just stuck until some other conconcurrent computation (if any) performs

a `KvPutOp` with the desired key. It does, however, remain an error in the pure interpreter.

Your task: Finish the implementation of the simulated concurrent interpreter in `APL.InterpSim`.

Hints: Consider postponing the treatment of `KvGetOp`/`KvPutOp` until the basics work.

Task E: Implement a key-value database

This task depends on no other tasks.

For this task you must implement a simple server for managing a concurrent key-value database *KVDB*. Your implementation must be in the `KVDB` module and implement the following API:

```
-- | A reference to a KVDB instance that stores keys
-- of type 'k' and corresponding values of type 'v'.
data KVDB k v

-- | Start a new KVDB instance.
startKVDB :: (Ord k) => IO (KVDB k v)

-- | Retrieve the value corresponding to a given key.
-- If that key does not exist in the store, then this
-- function blocks until another thread writes the desired
-- key with 'kvPut', after which this function returns
-- the now available value.
kvGet :: KVDB k v -> k -> IO v

-- | Write a key-value mapping to the database.
-- Replaces any prior mapping of the key.
kvPut :: KVDB k v -> k -> v -> IO ()
```

Note that the API is polymorphic—KVDB can store keys of any type that is an instance of `Ord`, and values of any type.

Your task: Implement the KVDB interface. You must make use of the `GenServer` module. It is up to you to design the state and protocol used to implement KVDB. You must also define appropriate tests in `KVDB_Tests`.

Task F: Implement a concurrent interpreter

This task depends on tasks C and E.

In this task you will implement a truly concurrent interpreter that potentially uses multiple Haskell threads to execute the `EvalM` monad. In particular, execution of the `BothOfOp` and `OneOfOp` involve actual IO-level concurrency. You must implement this concurrency by enqueuing jobs in `SPC`, *not* by using `forkIO` directly.

You must implement the same semantics for `KvGetOp`/`KvPutOp` as in Task D, meaning that trying to retrieve a value for a key that does not exist causes the thread to block until the key is available. Interpretation of `StepOp` is simply by recursively executing its continuation.

Your task: Finish the implementation of the concurrent interpreter in `APL.InterpConcurrent`. For handling `BothOfOp` and `OneOfOp`, you must make use of `SPC`. For handling `KvPutOp` and `KVGetOp` you must make use of KVDB.

Hints: Since SPC jobs are executed purely for side effects and have no return value as such, you will need to use `IORef`s to actually store the results of executing tasks. As in Task D, consider postponing the treatment of `KvGetOp`/`KvPutOp` until the basics work.

Your interpreter may crash with an error message "`thread blocked indefinitely in an MVar operation`", signalled by the Haskell runtime system. This is not necessarily an error in your implementation, but can be due to the APL program you are testing being invalid.

3 Code handout

The code handout consists of the following nontrivial files.

- `exam.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test execution program. **Do not modify this file.**
- `src/APL/AST.hs`: The APL AST definition. **Do not modify this file.**
- `src/APL/Parser.hs`: The APL parser, which you will modify in tasks A, B, and C.
- `src/APL/Eval.hs`: The APL evaluator, which you will modify in tasks A, B, and C.
- `src/APL/InterpConcurrent.hs`: Skeleton for an interpretation function for `EvalM` that uses true concurrency, which you implement in task F.
- `src/APL/InterpPure.hs`: A pure and non-concurrent interpretation function for `EvalM`, which you will modify in tasks B and C.
- `src/APL/InterpSim.hs`: Skeleton for a an `EvalM` interpretation function that simulates concurrency through stepping, which you implement in task D.
- `src/GenServer.hs`: The GenServer module. **Do not modify this file.**
- `src/Monad.hs`: Definition of the `EvalM` monad and the `Val` type. **Do not modify this file.**
- `src/KVDB.hs`: Skeleton for the key-value database implemented in task E.
- `src/SPC.hs`: Implementation of the simplified SPC. **Do not modify this file.**

4 Your Report

Your report should be no more than ten pages in length and must be structured exactly as follows:

Introduction: Briefly mention very general concerns, any ambiguities in the exam text and how you resolved them, and your own estimation of the quality of your solution. Briefly mention whether your solution is functional, which test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong.

A section answering the following numbered questions:

1. Show the final APL grammar with all ambiguities and left recursion removed, and explain to which extent it resembles your parser implementation.
2. Which of your tests demonstrate the evaluation order of tuples, and how?
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`)?
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?
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?
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?
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?

Your answers must be as precise and concrete as possible, with reference to specific programming techniques and/or code snippets when applicable. All else being equal, **a short report is a good report**.

A Examples

1 Tuples

Syntax: `let x = (1,2) in x.0`

AST: `Let "x" (Tuple [CstInt 1,CstInt 2]) (Project (Var "x") 0)`

Evaluation: `ValInt 1`

Syntax: `(1,2).1`

AST: `Project (Tuple [CstInt 1,CstInt 2]) 1`

Evaluation: `ValInt 2`

2 Loops

Syntax: `loop x = 1 for i < 5 do x * 2`

AST: `ForLoop ("x",CstInt 1) ("i",CstInt 5) (Mul (Var "x") (CstInt 2))`

Evaluation: `ValInt 32`

Syntax: `loop x = 1 while x == 1 do x`

AST: `WhileLoop ("x",CstInt 1) (Eql (Var "x") (CstInt 1)) (Var "x")`

Evaluation: Goes into an infinite loop.

Syntax: `loop x = (1,10) while if (x.1 == 0) then false else true do (x.0*2,x.1-1)`

AST: `WhileLoop ("x",Tuple [CstInt 1,CstInt 10]) (If (Eql (Project (Var "x") 1) (CstInt 0)) (CstBool False) (CstBool True)) (Tuple [Mul (Project (Var "x") 0) (CstInt 2),Sub (Project (Var "x") 1) (CstInt 1)])`

Evaluation: `ValTuple [ValInt 1024,ValInt 0]`

3 Concurrency operators

Due to nondeterminism, expressions may produce different results in the pure and simulated/concurrent interpreters. This is noted where relevant. The concurrent interpreter may further produce results that differ from the simulated interpreter, due to nondeterminism—only a single possible result is listed below.

Syntax: `(1+2) && (3+4)`

AST: `BothOf (Add (CstInt 1) (CstInt 2)) (Add (CstInt 3) (CstInt 4))`

Evaluation: `ValTuple [ValInt 3,ValInt 7]`

Syntax: `(1+2) || (3+4+5+6)`

AST: `OneOf (Add (CstInt 1) (CstInt 2)) (Add (Add (Add (CstInt 3) (CstInt 4)) (CstInt 5)) (CstInt 6))`

Evaluation (pure): `ValInt 18`

Evaluation (simulated): `ValInt 3`

4 Concurrent `put/get`

The evaluation results for the following examples apply only *for the simulated and concurrent interpretation functions.*

Syntax: `get 0 && put 0 true`

AST: `BothOf (KvGet (CstInt 0)) (KvPut (CstInt 0) (CstBool True))`

Evaluation: `ValTuple [ValBool True,ValBool True]`

Syntax: `get 0 + 1 && put 0 2`

AST: `BothOf (Add (KvGet (CstInt 0)) (CstInt 1)) (KvPut (CstInt 0) (CstInt 2))`

Evaluation: `ValTuple [ValInt 3,ValInt 2]`

Syntax: `put (get 0) 1 && let x = put 0 2 in get 2`

AST: `BothOf (KvPut (KvGet (CstInt 0)) (CstInt 1)) (Let "x" (KvPut (CstInt 0) (CstInt 2)) (KvGet (CstInt 2)))`

Evaluation: `ValTuple [ValInt 1,ValInt 1]`