Advanced Programming 2024

Reexam Problem

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

Preamble

For all questions regarding the exam, contact Mikkel Kragh Mathiesen <`mkm@di.ku.dk`>.

This document consists of 15 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 focussed 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: arrays, array comprehensions, `compose`, message passing (`sendl`, `sendr`, `recvl`, `recvr`), 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

Operators	**Associativity**
Application	Left
`!`	Left
`*` `/`	Left
`+` `-`	Left
`==`	Left
`<->`	Left

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

a slightly simplified version of the SPC job scheduler (which you must 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 chapter 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 ")"
       | "[" "]"
       | "[" Exps "]"
       | "[" Exp "|" var "<-" Exp "]"
       | "compose" Atom
       | "sendl" Atom
       | "sendr" Atom
       | "recvl"
       | "recvr"

Exps ::= Exp
       | Exps "," Exps

FExp ::= FExp FExp
       | Atom

LExp ::= "if" Exp "then" Exp "else" Exp
       | "\" var "->" Exp
       | "let" var "=" Exp "in" Exp
       | FExp

Exp ::= LExp
      | 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 arrays

This task depends on no other tasks.

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

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

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

```
"[" "]"
```

corresponding to an array with no elements, and by the production

```
"[" Exps "]"
```

corresponding to an array with at least one element.

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

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

For example, the input `[a, b]` corresponds to the following `Exp`:

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

An array expression `[`e_0`,...,`e_{N-1}`]` is evaluated by evaluating the components from left to right, then constructing a `ValArray` 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 array also fails and the first error encountered should be reported.

The elements of an array `x` can be indexed (accessed) with the syntax `x ! i`, where `i` is an expression which evaluates to an integer, such as in `x ! 0` which indexes element 0 of the array. For example, `[`e_0`,...,`e_{N-1}`] ! 0` = e_0. If `x` has N elements, then `i` must evaluate to an integer between 0 and $N-1$; using an index outside of this range is an error. The corresponding AST constructor is `Index`. It is an error to try to index an element from a non-array.

Your task: Implement parsing of array construction and indexing in `APL.Parser` and evaluation of arrays and indexing in `APL.Eval`.

Task B: Implement array operations

This task depends on no other tasks.

For this task you must augment APL with array comprehensions and aggregations. An *array comprehension* expression

```
[head | x <- array]
```

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

1. Evaluate expression `array` to a value *a*. It is an error if *a* is not an array.
2. For each value *v* in the array *a* in order from first to last, bind variable `x` to *v* and evaluate `head`.
3. Return an array of the resulting values.

Essentially, array comprehensions behave like list comprehensions in Haskell.

A `compose` expression

```
compose e
```

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

1. Evaluate expression `e` to a value *a*. It is an error if *a* is not an array. Furthermore, each value in the array must be a function.
2. Return the composition of these functions, i.e. return `\x -> f1 (f2 (... (fn x)))` where `f1` to `fn` are the functions in the array *a*.

For example, the expression `compose [(\x -> x + 1), (\x -> x + 2)]` evaluates to `\x -> f1 (f2 x)` where `f1 x = x + 1` and `f2 x = x + 2`.

In particular `compose` can be used to define aggregations. For example, `compose [\x -> x + n | n <- a] 0` computes the sum of the integers in the array `a`.

Your task: Implement parsing of array constructs in `APL.Parser` and evaluation in `APL.Eval`. Treat `compose` as a keyword.

Task C: Implement concurrency operators

This task depends on no other tasks.

For this task you must implement the <-> and communication operators intended for concurrent programming.

The expression `e1 <-> e2` denotes concurrent execution of two expressions. To evaluate the expression `e1 <-> e2` we evaluate `e1` and `e2` in unspecified order. They must both evaluate to array values. The result is the concatenation of the arrays (like ++ for Haskell lists). If either subexpression fails, then the overall expression also fails. If both fail, either error may be reported.

The expressions `sendl e`, `sendr e`, `recvl` and `recvr` are meant for communication between the two computations of <->. This is not meaningful for the pure interpreter, however, so they should all fail with an error.

Your task: Transform the grammar of fig. 1 to eliminate left recursion and ambiguity. Then implement parsing of <->, `sendl`, `sendr`, `recvl` and `recvr` in `APL.Parser` and evaluation in `APL.Eval`. Evaluation of <-> is with the `SplitOp` effect (`evalSplit`), and evaluation of the communication primitives is with the correspondingly named effect.

Extend `APL.InterpPure` to handle the `SplitOp` effect. Also implement handling of the communication primitives (which should just fail with an error).

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 the possibility of communication between the two subcomputations of <->.

The `sendl` (resp. `sendr`) operator sends a message to the computation on the left (resp. right), and then returns a copy of the sent value. The `recvl` (resp. `recvr`) operator receives a message from the left (resp. right). However, communication is *synchronous* so the message is only sent when the neighbouring computation is ready to receive a message from the appropriate direction. A simple example is `sendr [42] <-> recvl` where the left computation will send the message `[42]` to the right computation; it is thus equivalent to `[42] <-> [42]` which evaluates to `[42, 42]`.

If the left (resp. right) subcomputation of a <-> sends/receives to the left (resp. right) then the whole computation may send/receive to the left. For instance in the program `sendr [42] <-> (recvl <-> [666])` the message `[42]` will also be sent from the first subcomputation to the second one. Note that expressions like `recvl <-> recvr` may interact both to the left and to the right; if both are possible it is unspecified which happens first.

In no other cases will messages be sent. In particular an expression like `f (sendr 1) recvr <-> g (sendl 2) recvr` is forever stuck because both subcomputations are trying to send before receiving, so it is impossible to match a sender and a receiver.

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

Hints: You might want to define a helper function to search an effect for sends/receives, which might be under an arbitrary number of <-> operators.

Task E: Implement a message exchange server

This task depends on no other tasks.

For this task you must implement a simple message exchange server (MES) that allows processes to exchange messages synchronously. Your implementation must be in the MES module and implement the following API:

```
-- | A reference to an MES instance that deals with processes
-- of type 'p' and messages of type 'v'.
data MES p v

-- | Start a new MES instance.
startMES :: (Ord p) => IO (MES p v)

-- | The call ‘newChannel mes p1 p2‘ establishes
-- a communication channel between p1 and p2, with p1
-- being the ‘left‘ process and p2 being the ‘right‘ process.
-- Any previous channels for which p1 is ‘left‘ or p2 is ‘right‘
-- are deleted. Previous channels where p1 is ‘right‘ or p2 is
-- ‘left‘ are left intact.
newChannel :: MES p v -> p -> p -> IO ()

-- | Send a value from the given process, which should be the
-- ‘right‘ part of a channel, to the process on the ‘left‘ side.
-- Blocks until the message is delivered to the recipient.
sendLeft :: MES p v -> p -> v -> IO ()

-- | Send a value from the given process, which should be the
-- ‘left‘ part of a channel, to the process on the ‘right‘ side.
-- Blocks until the message is delivered to the recipient.
sendRight :: MES p v -> p -> v -> IO ()

-- | Receive a value for the given process, which should be the
-- ‘right‘ part of a channel, from the process on the ‘left‘ side.
recvLeft :: MES p v -> p -> IO v

-- | Receive a value for the given process, which should be the
-- ‘left‘ part of a channel, from the process on the ‘right‘ side.
recvRight :: MES p v -> p -> IO v
```

Note that the API is polymorphic—MES can deal with processes of any type that is an instance of Ord, and values of any type.

Your task: Implement the MES interface. You must make use of the `GenServer` module. It is up to you to design the state and protocol used to implement MES. You must also define appropriate tests in `MES_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 <-> involves 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 the communication primitives as in Task D. In particular, message passing is still synchronous.

Your task: Finish the implementation of the concurrent interpreter in `APL.InterpConcurrent`. For handling <->, you must make use of `SPC`. For handling communication primitives you must make use of MES.

Hints: Since SPC jobs are executed purely for side effects and have no return value as such, you will need to use `IORefs` to actually store the results of executing tasks.

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, which you implement in task D.
- `src/APL/*_tests.hs`: Skeleton test suites.
- `src/APL/Monad.hs`: Definition of the `EvalM` monad and the `Val` type. **Do not modify this file.**
- `src/GenServer.hs`: The GenServer module. **Do not modify this file.**
- `src/MES.hs`: Skeleton for the message exchange server 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. Explain how you disambiguate array literals and array comprehensions. Show relevant parts of your transformed grammar.
2. Which of your tests demonstrate the evaluation order of array comprehensions, and how?
3. How can the concurrency primitives be used to simulate mutable state?
4. Is <-> associative? That is, do `e1 <-> (e2 <-> e3)` and `(e1 <-> e2) <-> e3` behave the same?
5. When the expression `[1] <-> [2]` is run using the concurrent interpreter, which processes are involved?
6. Is it possible for the simulated concurrent interpreter (`APL.InterpSim`) to go into a deadlock due to `send`/`recv`? Is this easy to detect in some cases? Is it easy to detect in all cases?
7. Is it possible for the concurrent interpreter (`APL.InterpConcurrent`) to go into a deadlock due to `send`/`recv`? Is this easy to detect in some cases? Is it easy to detect in all cases?

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 Arrays

Syntax: `let x = [1,2] in x!0`

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

Evaluation: `ValInt 1`

Syntax: `[1,2]!1`

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

Evaluation: `ValInt 2`

2 Array comprehensions

Syntax: `[x + 1 | x <- [1, 2, 3]]`

AST: `Map (Add (Var "x") (CstInt 1)) "x" (Array [CstInt 1, CstInt 2, CstInt 3])`

Evaluation: `ValArray [ValInt 2, ValInt 3, ValInt 4]`

Syntax: `compose [\x -> x + n | n <- [8, 13, 21]] 0`

AST: `Apply (Compose (Map (Lambda "x" (Add (Var "x") (Var "n"))) "n" (Array [CstInt 8, CstInt 13, CstInt 21]))) (CstInt 0)`

Evaluation: `CstInt 42`.

---

3 Concurrency operators

Syntax: `[true] <-> [false]`

AST: `Split (Array [CstBool True]) (Array [CstBool False])`

Evaluation: `ValArray [ValBool True, ValBool False]`

Syntax: `sendr [42] <-> recvl`

AST: `Split (SendRight (Array [CstInt 42])) RecvLeft`

Evaluation (pure): an error

Evaluation (simulated): `Array [ValInt 42, ValInt 42]`