Advanced Programming 2024
Deep Embedded DSL
Assignment 8

April 17 2025

1 Goal

The goal of this exercise is to understand the possibilities and limitations of a deep embedded DSL using (our version of) GADT's. For this purpose we implement a DSL that controls a harbour crane. Try to prevent run-time errors by adding type arguments.

The harbour crane loads containers on and off a ship. For simplicity, we assume there is only a single stack of containers on the ship and a single stack on the quay. These stacks are able to accommodate any number of containers. The crane is able to execute the following actions.

```
:: Action
 = MoveToShip                 // move the crane to the ship
 | MoveToQuay                 // move the crane to the quay
 | MoveUp                     // moves the crane up
 | MoveDown                   // moves the crane down
 | Lock                       // locks the top container of the stack under the crane
 | Unlock                     // unlocks the container the crane is carrying, put it on the stack
 | Wait                       // do nothing
 | (:.) infixl 1 Action Action // sequence of two actions
 | WhileContainerBelow  Action // repeat action while there is a container at current position
```

The following program loads all containers from the quay on the ship, given that the crane is initially up, does not carry a container and is above the quay.

```
loadShip
   = WhileContainerBelow (
         MoveDown :.
         Lock :.
         MoveUp :.
         MoveToShip :.
         Wait :.
         MoveDown :.
         Wait :.
         Unlock :.
         MoveUp :.
         MoveToQuay
     )
```

Most actions should only be applied in specific states. Some violations of this rule are pretty harmless, e.g., `MoveToShip` when the crane is already above the ship. These actions can be considered as a no-operation. Other actions are very dangerous, like unlocking a container while the crane is `Up`, or moving the crane horizontally when it is `Down`. Finally, there are actions that are impossible to execute, for instance `Unlock` when the crane is not carrying a container or `Lock` when the crane is up, already carrying a container, or the stack is empty.

2 Assignment

2.1 Adding Safety

As indicated above we can write very dangerous programs in this DSL, e.g.:

```
MoveDown :. Lock :. MoveUp :. Unlock
```

It is only human to make these kind of errors when writing a program for the crane. A solution would be to construct the evaluator of the DSL in such a way that the dangerous actions are not executed. A safer approach is to improve the design of the DSL such that dangerous programs are ill-typed and hence cannot be executed.

Improve the type `Action` such that moving the crane when it is low, or (un)locking it when it is high is a type error in the DSL using the poor man's GADT approach outlined in the lecture.

Hint: introduce two type arguments in `Action` indicating the initial and final position (low/high) of the crane. The following types were used in our solution.

```
:: High = High
:: Low  = Low
```

2.2 Evaluator

Write an evaluator for your improved type `Action` using

```
:: ErrorOk s = Error String | Ok s
```

We use `String` to represent errors. The evaluator should produce such an error whenever that action cannot be executed in the current state, e.g., locking a container on an empty stack. The state can be modeled as:

```
:: State highLow
  = { onShip      :: [Container]
    , onQuay      :: [Container]
    , craneOnQuay :: Bool
    , locked      :: ? Container
    }
:: Container :== String

state0 :: State High
state0
  = { onShip      = []
    , onQuay      = ["apples","beer","cheese"]
    , craneOnQuay = True
    , locked      = ?None
    }

:: Step i f :== (ErrorOk (State i)) -> ErrorOk (State f)
```

Monads will save you some checking on error or okay. You will need the Monadic instances for `ErrorOk` and implement an `eval :: (Action i f) -> Step i f`. It is perfectly fine to start with a version without Monads. It is not required, but encouraged, to transform the evaluator to a Monadic version when you start with a plain implementation.

2.3 Printing

Define a function `print` that turns an `Action` in a `[String]` representing it.

2.4 Optional: optimizer

Write an optimizer for actions that removes all `Wait` steps from an `Action`.

Deadline

To receive feedback, hand in your solution before May 7 23:59h.