Advanced Programming (I00032)
A DSL for Crane Control with GADT's

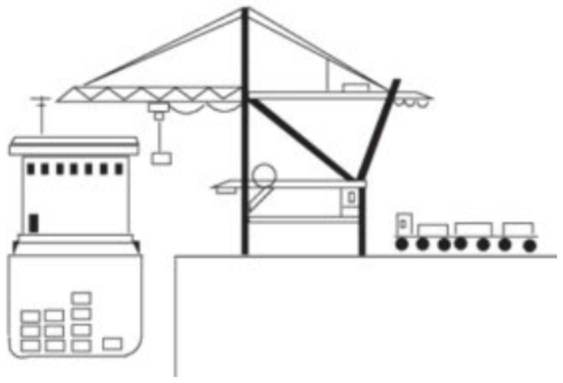
Assignment 11

Goal

The purpose of this assignment is to understand the possibilities and limitations of a deep embedded DSL using (our version of) GADT's. For this purpose we implement the DSL controlling a physical device. Try to prevent runtime type errors whenever possible.

Crane Control

To load the containers on and off a ship we have a harbor crane controlled by a DSL. This is illustrated by the following figure:



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 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, but 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.

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. A solution would be to construct the executor 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 in 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 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 Evaluation

Write an evaluator for your improved type action using

```
:: ErrorOrResult e r = Error e | Result r
```

It is fine to use `String` as representation of the actual errors, some tailor-made enumeration type might be even better. 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 modelled as:

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

initialState =
    { onShip      = []
    , onQuay      = ["apples","beer","camera's"]
    , craneUp     = True
    , craneOnQuay = True
    , locked      = Nothing
    }
```

3 Printing

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

4 *Bonus, you do not have to make this:* Optimization

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

Deadline

The deadline for this assignment is Thursday, May 27 at 13:30h. This is just before the next lecture. Due to Whit Monday this lecture is moved to Thursday.