Advanced Programming (I00032)
iTasksLite

Assignment 6

Preparation

Please, use the provided `.prj` files, also if you work on the console. The `clm` tool will not work, because some necessary modules make use of hirarchic module names (e.g. `Control.Monad`), which are not suppoted by `clm`. So compile your code with:

```
cpm skeleton6a.prj
```

1 iTasksLite Using a Unique State

In this assignment you will implement a simplified version of the iTask system, using the console for user interaction instead of webforms. We make use of the following state:

```
:: *TaskState = { console :: !*File
                , store   :: Map String Dynamic
                }
```

As discussed during the lecture, the console is represented as a unique `File`. Additionally, there is a store which is used to implement tasks for storing and retrieving values, similar to the shared stores you have used to store a list of ideas in the previous assignment. You do not have to care about how this store works as the following functions are provided in the skeleton:

```
store_    :: a (StoreID a) (Map String Dynamic) → Map String Dynamic | TC a
retrieve_ ::   (StoreID a) (Map String Dynamic) → a                  | TC a
```

However, if you are curious how the store works and are not familiar with Clean's dynamics, have a look at Chapter 8 of the language report (`<clean home>/doc/CleanLangRep.2.2.pdf`). The type `Map` is a key-value store and in this case maps String identifiers of stores to the dynamic representation of their value (see `<clean home>/lib/clean-platform/OS-Independent/Data/Map.dcl`)

1.1 `viewInformation`

Implement the task `viewInformation` with the following type:

```
viewInformation :: Description a TaskState → TaskResult a | iTasksLite a
```

The `Description` type is just a synonym for strings:

```
:: Description :== String
```

The result of type `TaskResult a` contains the unique state discussed above and a result of type `a`:

```
:: *TaskResult a :== (a, TaskState)
```

The class `iTasksLite` finally includes all classes neccessary for making the system work, in the same way as the `iTask` class in the real iTask system:

```
class iTasksLite a | print a & parse a & TC a
```

The printing and parsing functions are used for user interaction on the console. Instances for a few types are provided in the skeleton. Of course, it would be nicer to use generic functions here, but printing/parsing are not the focus in this assignment and the simple implementations work well enough. For printing to the console you can use the `<<<` operator, shown in the lecture.

With the implementation the first test task in the skeleton should work:

```
task1 :: (TaskState → TaskResult Int)
task1 = viewInformation "The answer is" 42
```

This should give something like:

```
Welcome to iTasksLite

The answer is: 42

The result of the task is 42.
```

Why is the type of `task1` (`TaskState → TaskResult Int`) and not `TaskState → TaskResult Int`?

1.2 `enterInformation`

Implement `enterInformation`:

```
enterInformation :: Description TaskState → TaskResult a | iTasksLite a
```

In case the parser fails (yields `Nothing`), ask the user to provide a value again. Use `freadline` to let the user input data.

The second test program should now work:

```
task2 :: TaskState → TaskResult Int
task2 st
   # (x, st) = enterInformation "Enter the answer" st
   =           viewInformation "The anser is" x st
```

The program should behave like this:

```
Welcome to iTasksLite

Enter the answer: there is none
Wrong format, try again.
Enter the answer: 42
The anser is: 42

The result of the task is 42.
```

1.3 `store` & `retrieve`

Implement the tasks to store and retrieve values from a store, using the provided functions `store_` and `retrieve_`:

```
store    :: a (StoreID a) TaskState → TaskResult a | iTasksLite a
retrieve ::   (StoreID a) TaskState → TaskResult a | iTasksLite a
```

The type `StoreID a` is actually just a string identifier, with an attached type, to indicate which type of values can be stored:

```
:: StoreID a :== String
```

The type `a` is also called a phantom type, as no data of this type is actually contained. Still such types can increase type safety.

Consider the following test program:

```
task3 :: TaskState → TaskResult Int
task3 st
    # (_, st) = store 1 intStore st
    =          retrieve intStore st
where
    intStore :: StoreID Int
    intStore = "intStore"
```

We define a store containing integers and use it to store and retrieve an integer. Running the program should give:

```
Welcome to iTasksLite



The result of the task is 1.
```

Using such stores can cause runtime errors. Consider the following program:

```
task3Fail = retrieve intStore
where
    intStore :: StoreID Int
    intStore = "intStore"
```

This causes an error, as the store is empty and has therefore no value one can retrieve. In `retrieve_` the program is just ended using an `abort`. As you can see by looking at the code, there is also another error message, for the case the type of the stored value does not match the asked one. **Write a task `task3TypeFail` that causes this error message.**

Finally, test `task4`, which lets the user enter one idea after another and adds them to a store:

```
task4 :: TaskState → TaskResult [Int]
task4 st
    # (_, st) = store [] ideaStore st
    = addIdea st
where
    addIdea st
        # (ideas, st) = retrieve ideaStore st
          (_,     st) = viewInformation "All ideas" ideas st
          (idea,  st) = enterInformation "Enter new idea" st
          (_,     st) = store (ideas ++ [toString (length ideas+1) +++ ". " +++ idea]) ideaStore st
        = addIdea st
```

```
ideaStore :: StoreID [String]
ideaStore = "ideas"
```

Remove the strictness annotation `!` of the `console` field and run `task4` again. **What does change and why?**

2 iTasksLite Using a Monad

In the task definitions above, the state is visible. It can be hidden by using a task monad. This allows task definitions similar to what you have seen before in iTasks. For instance the last task can then be written like:

```
task4 :: Task Void
task4 =
        store [] ideaStore
    >>| addIdea
where
    addIdea =
                     retrieve ideaStore
        >>= λideas → viewInformation "All ideas" ideas
        >>|          enterInformation "Enter new idea"
        >>= λidea  → store (ideas ++ [toString (length ideas+1) +++ ". " +++ idea]) ideaStore
        >>|          addIdea

    ideaStore :: StoreID [String]
    ideaStore = "ideas"
```

All test tasks from the first part are provided in this style in `skeleton6b.icl`.

2.1 The `Task` Monad

Define a type `Task a` that is suited to contain operations on the task state, similar to the state monad shown in the lecture. **Provide instances of `Functor`, `Applicative` and `Monad` for this type**. Note that this provides you also with `>>=`, `>>|` and `return` for tasks. The task `task0` should already work with a proper monad instance.

2.2 Task Implementations

Implement the tasks `viewInformation`, `enterInformation`, `store` and `retrieve`. For instance, the type of `viewInformation` becomes:

```
viewInformation :: Description a → Task a | iTasksLite a
```

Test your implementation with `task0` – `task4`. They should behave the same way as the tasks in the first part of the assignment.

3 Bonus: Exceptions

Add the possibility to throw exceptions with the task:

```
throw :: Exception → Task a | iTasksLite a
```

Exceptions can just be strings:

```
:: Exception :== String
```

Implement a task for catching exceptions:

```
tryCatch :: (Task a) (Exception → Task a) → Task a | iTasksLite a
```

The second argument is only executed in case an exception occurs. Adapt also the `retrieve` task, such that it does not crash using `abort`, but throws exceptions in case of errors. **Provide a small task that illustrates how exceptions are used.**

Deadline

The deadline for this exercise is October 19, 13:30h.