Advanced Programming (I00032) 2018
Generics by overloading

Assignment 2

Goals of this exercise

In this exercise you program the transformation from datatypes to their generic representation and back to implement serialization in a generic way. This gives you a proper understanding of the generic mechanism.

1 Review Questions

Answer the following questions before implementing classes and instances in section 2. You can include the answers as a comment in your `.icl` file.

1. The definition of==for `UNIT` in the slides is

   ```
   instance == UNIT where (==) UNIT UNIT = TRUE
   ```

   This looks odd, there is a pattern match, but no alternative. Is it better to write

   ```
   instance == UNIT where
       (==) UNIT UNIT = TRUE
       (==) x    y    = FALSE
   ```

   or can we write

   ```
   instance == UNIT where (==) x y = TRUE
   ```

2. The definition for (==) for `CONS` is

   ```
   instance == (CONS a) | == a where (==) (CONS _ x) (CONS _ y) = x==y
   ```

   shouldn't we check the equality of constructor names as in

   ```
   instance == (CONS a) | == a where (==) (CONS a x) (CONS b y) = a==b && x==y
   ```

3. Given

   ```
   :: Bin   a = Leaf | Bin (Bin a) a (Bin a)
   :: BinG  a :== EITHER (CONS UNIT) (CONS (PAIR (Bin a) (PAIR a (Bin a))))
   :: ListG a :== EITHER (CONS UNIT) (CONS (PAIR a [a]))
   ```

   What are the generic representations of the values [] and `leaf`?

   Does this imply that `Leaf` == [] yields `True` if we define it in the generic way?

2 Generic serialization

In exercise 1 we defined serialization of objects by a class

```
class serialize a where
  write :: a [String] → [String]
  read  :: [String] → Maybe (a,[String])
```

In this assignment we reimplement this class using the generic representation of data types:

```
:: UNIT       = UNIT
:: EITHER a b = LEFT a | RIGHT b
:: PAIR   a b = PAIR a b
:: CONS   a   = CONS String a
```

The types to be serialized are the native lists of Clean, and binary trees `Bin`. The generic representations `ListG`, and `BinG` are listed above. Define the transformation functions

```
fromList :: [a] → ListG a
toList   :: (ListG a) → [a]
fromBin  :: (Bin a) → BinG a
toBin    :: (BinG a) → Bin a
```

similar to transformations defined in the lecture slides.

2.1 With generic information

Define instances of `serialize` for `[a]` and `Bin a` based on their generic representation. Include all generic information in the serialized versions of data types, the serialized version contains strings like `"UNIT"` and `"LEFT"`.

2.2 Without generic information

For humans it is nicer to omit the generic information. Only the basic types and the constructor names from the data types (like `Leaf`, `Bin`, `Nil`, and `Cons`) are included. For the `write` part this very simple. In the `read` it is at some places a little more advanced, especially in the instance for `EITHER` you need to backtrack. Be sure to include enough parenthesis whenever necessary.

Optional: Prettier serialization

Most likely your serialization of single constructor values like `Leaf` contains parenthesis, e.g. `["(","Leaf",")"]` of the value `Leaf`. Beautify the serialization by omitting these parenthesis for constructors without arguments, e.g. `["Leaf"]`.

3 Reflection

When you implement everything correctly this will work fine for the listed types and all tests should pass. Can you come up with an example that breaks this system?

Although it is allowed to solve this problem, this is certainly not required.

Deadline

The deadline for this exercise is September 24 2018, 10:30h (just before the next lecture).