Prof. Dr. Peter Thiemann
Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 13

Exercise 1 (Generic Programming)

In the lecture, we saw that algebraic data types can be expressed as fixpoints of functors. In this exercise, we will build on this machinery to do generic programming, i.e. writing algorithms, which work uniformly for all algebraic datatypes.

A classical application for generic programming is serialization, e.g. converting between data types and some serialization format like JSON or binary code. This application is relevant in many programs, e.g. when sending data over a network socket or saving it in a file or database.

The following shows a first attempt at a serialization API:

```
toBits   :: a -> [Bool]         -- Convert an `a` to a list of bits
fromBits :: [Bool] -> Maybe a   -- Try to convert a list of bits to an `a`.
                                -- Might fail for bad bits.
```

These functions could clearly be used with arbitrary argument types, however it is impossible to implement them, because the implementation depends on the concrete structure of the type we put in for `a`. As a second attempt, we might use a type class:

```
class Serialize a where
  toBits   :: a -> [Bool]
  fromBits :: [Bool] -> Maybe a
```

Using a type class allows us to have different behavior for different types, but someone needs to manually write `Serialize`-instances for every new algebraic datatype that is defined.

This is rather annoying, because all of those instances follow the same pattern. In particular, we could capture this pattern by writing an algorithm that given the description of an algebraic data type (how many constructors, types of the fields of the constructors, etc.) produces us a `Serialize`-instance for that type. The problem is that we cannot directly implement this algorithm in Haskell because it is a function from Haskell code to Haskell code.

What we would like is to simply annotate a data type with `deriving (Serialize)`, but without needing to submit a pull request to the Haskell compiler to hardcode our generic algorithm.

In Haskell there are two main ways of achieving that:

1. *Template Haskell* is a macro system[1], which literally allows writing Haskell functions, which map Haskell syntax trees to Haskell syntax trees. Using Template Haskell divides the compilation of Haskell into two stages: in the first stage, the Template Haskell code is executed, which transforms the Haskell code written by the user to Haskell code without Template Haskell; and in the second stage that Haskell code is then compiled as usually.

   While Template Haskell is very powerful and avoids performance overhead at runtime, it also adds some cognitive overhead and it is rather cumbersome to write algorithms as code transformations and to debug them.

[1]Comparable with Rust's proc-macros, but significantly more powerful than C/C++ preprocessor macros.

2. *Generic Programming techniques* do not require multiple stages, but allow expressing generic algorithms directly in Haskell without the need for syntax tree transformations. This is achieved by finding a way to encode arbitrary algebraic data types in just a finite amount of fixed datatypes, e.g. sum types `Either a b` for the constructors and product types `(a, b)` for the different fields within a constructor.

   A generic algorithm can then be expressed as a type class with an instance that says how it behaves on product types, sum types, etc. To use the generic algorithm, we first encode its generic arguments, then run it, and then decode its generic results back to the actual datatype.

   If we have a new algebraic data type, then all we need to do is to implement the encoding and decoding functions for it, and in return all generic algorithms can be used on it. In practice, the encoding and decoding functions are automatically derived either directly by the compiler[2] or via a macro system like Template Haskell, so new algebraic datatypes can be used directly with all generic algorithms without needing to write any boilerplate code.

In this exercise, we are going to look at a generic programming technique, which is based on the functor fixpoints that we have already explored in the lecture. It comes without bells and whistles (like accessing constructor names and deriving the encoding and decoding functions), but its simplicity and transparency makes it suitable for getting your feet wet with generic programming. In practice, you probably want to use one of the above mentioned techniques.

The idea of this technique is that each algebraic datatype is isomorphic to a fixpoint of a specific functor. This functor can be expressed as a combination of two primitive functors (`Id` and `Const b`) and two functor combinators (`:+:` and `:*:`)[3].

The relevant definitions are:

```
newtype Const b a = Const b                 -- Constant functor
newtype Id a = Id a                         -- Identity functor

data (:+:) f g a = Inl (f a) | Inr (g a)   -- Sum functor (aka Coproduct)
data (:*:) f g a = Pair (f a) (g a)        -- Product functor (not covered in lecture)

newtype Fix f = Fix (f (Fix f))            -- this was called `Mu` in the lecture
```

Computing the functor from an algebraic datatype definition works as follows:

- Each field of a constructor corresponds to a primitive functor. If the type `b` of the field is the algebraic datatype itself, then the functor is `Id`, otherwise it is `Const b`.
- Each constructor is a tuple of its fields. Consequently, the functor for a constructor is the product functor of its field functors, e.g. `f1 :*: f2 :*: ... :*: fn`. Constructors without fields can be treated as if they have a single field of type `()`.
- A value of the datatype is one of its constructors, so the functor for the whole datatype is the sum functor of its constructor functors `c1 :+: c2 :+: ... :+: cn`.

Let's say `f` is the functor obtained from a datatype in this manner, then `Fix f` is isomorphic to the original datatype.

For example, we can define an algebraic datatype for integer lists as

```
data IntList = Nil | Cons Int IntList
```

and encode them as

[2]See Haskell's `Generic` type class and `deriving (Generic)`

[3]This is true for all algebraic datatypes except for those, which contain functions where the type itself appears recursively, e.g. `data D = C (Int -> D)`. We omit these cases for simplicity, but it is also possible to extend the technique to make it work with them.

```
Fix (Const () :+: (Const Int :*: Id))
```

The Nil constructor has no fields, so we treat it as if it has a single field of type (), which we represent as Const (). Since there is only one field, there isn't really a product to compute so we represent the whole Nil constructor as Const (). The Cons constructor has two fields, the first one has type Int, so we represent it as Const Int, and the second one has type IntList so we represent it as Id.

Note that if we apply the functor to an argument type a, then the resulting type

```
(Const () :+: (Const Int :*: Id)) a
```

is isomorphic to

```
Either () (Int, a)
```

because Const b a is isomorphic to b and Id a is isomorphic to a. If Haskell would support type-level lambdas, then we could encode the type instead simply as

```
Fix (\a -> Either () (Int, a))
```

which conveys more clearly what we want to do here.

To capture algebraic data types for which we defined such an encoding, we define a type class:

```
class Functor f => GenRepr a f | a -> f where
  toGen   :: a -> Fix f
  fromGen :: Fix f -> a
```

A constraint GenRepr a f means that the type a is isomorphic to Fix f. The functional dependency | a -> f forbids us from defining two GenRepr instances for the same a. This improves type inference, because for a constraint GenRepr a f the type checker only needs to know a, but not f, which leads to less ambiguity errors.

Your task is to use the code supplement from the website, implement GenRepr for certain datatypes and write a few generic algorithms, which then work for all datatypes that have a GenRepr instance. In particular:

1. Figure out the functor f for the following algebraic datatypes a and write corresponding GenRepr a f instances:

   - data Bool = True | False
   - lists
   - data Tree a = Leaf | Branch (Tree a) a (Tree a)

   For lists and trees the functor f will have the functor of the element type e as an additional parameter. This is necessary, because for example [[Bool]] should encode also the inner lists and booleans and not just the outer list. Consequently, your list instance should look like

   ```
   instance GenRepr a e => GenRepr [a] (ListF e) where
   ```

   Example:

   ```
   >>> toGen [True, False]
   (Fix (Inr (Pair (Const (Fix (Inr (Const ()))))
                   (Id (Fix (Inr (Pair (Const (Fix (Inl (Const ()))))
                                       (Id (Fix (Inl (Const ())))))))))))
   ```

2. Write a generic function `depth`, which calculates the recursion depth of an algebraic datatype:

   - For lists, `depth` is the length of the list;
   - For trees, `depth` is the depth of the tree, i.e. the length of the longest path from root to leaf; and
   - For booleans, `depth` is always `0`.

   To implement `depth` generically, we need to create a typeclass and instanciate it for each of the primitive functors and functor combinators:

   ```
   class Functor f => GenDepth f where
     genDepth :: f Int -> Int

   instance GenDepth (Const b) where ...
   instance GenDepth Id where ...
   instance (GenDepth f, GenDepth g) => GenDepth (f :+: g) where ...
   instance (GenDepth f, GenDepth g) => GenDepth (f :*: g) where ...
   ```

   We can then define `depth` as follows:

   ```
   depth :: (GenRepr a f, GenDepth f) => a -> Int
   depth x = foldFix genDepth $ toGen x
   ```

   Note that since we implemented `GenDepth` for exactly those functors to which we limited ourselves when implementing `GenRepr`, the `GenDepth f` constraint is trivially satisfied when `GenRepr a f` is satisfied.

   Note that `depth` should not recurse into substructures, i.e. calling `depth` on a nested list returns the length of the outer list just like the real `length` function. This can be easily achieved by simply ignoring all `Const b` functors by assigning them depth `0`.

3. Write a generic function `toBits`, which serializes arbitrary algebraic datatypes to bits.

   We recommend to use the following datatype instead of booleans, because it has a much more readable `Show` instance for large lists of bits:

   ```
   data Bit = O | I deriving (Show, Eq)
   type Bits = [Bit]
   ```

   The implementation is similar to the `depth` function from the previous exercise part, with one important difference: nested datastructures need to be serialized recursively, e.g. to serialize a `[[Bool]]` it is not sufficient to serialize the outer list, but also the inner bool lists and the bools inside of them.

   This concerns the instance for `Const e`, which now needs to be split into multiple instances:

   ```
   instance GenToBits f => GenToBits (Const (Fix f)) where ...
   instance GenToBits (Const ()) where ...
   ```

   The first instance is to cover other algebraic datatypes, e.g. when a list of trees is serialized, then in the generic representation of the list each tree will be a `Const (Fix f)` where `f` is the functor for trees. The second instance is relevant for constructors without any fields, like the nil constructor `[]`.

   Examples:

```
>>> toBits False
[O]
>>> toBits True
[I]
>>> toBits [False]
[I,O,O]
>>> toBits [True]
[I,I,O]
>>> toBits [False, False]
[I,O,I,O,O]
```

In `toBits [False] == [I,O,O]`, the `I` stands for taking the `Inr` constructor (`(:)`), the middle `O` stands for `False`, and the last `O` stands for taking the `Inl` constructor (`[]`).

4. (Bonus) Write a generic function `fromBits`, which deserializes arbitrary algebraic datatypes from bits.

   This implementation is quite a bit different from the previous two, which is mainly caused by the fact that it is not defined by recursing over the `Fix f`, but instead we recurse over `Bits` and try to build a `Fix f` as output.

   We recommend to implement the `fromBits` function in terms of a `fromBits'` function:

```
fromBits' :: (GenRepr a f, GenFromBits f) => Bits -> Maybe (a, Bits)
fromBits' = map' (fromGen . Fix) . fromBitsGen
```

   where `Bits -> Maybe (a, Bits)` is basically a simple form of the parser monad, which does not support ambiguity (`Maybe` instead of lists). It is not necessary to implement any parser combinators for this exercise, but it suffices to use the functor and monad instances for `Maybe` at a few places.

   The actual `fromBits` function can then be defined in terms of `fromBits'` and additionally asserts that no bits are left-over:

```
fromBits :: (GenRepr a f, GenFromBits f) => Bits -> Maybe a
fromBits bs = case fromBits' bs of
  Just (x, []) -> Just x
  _            -> Nothing
```

   We recommend the following structure for the type class:

```
class Functor f => GenFromBits f where
  fromBitsGen :: GenFromBits g => Bits -> Maybe (f (Fix g), Bits)

instance GenFromBits f => GenFromBits (Const (Fix f)) where ...
instance GenFromBits (Const ()) where ...
instance GenFromBits Id where ...
instance (GenFromBits f, GenFromBits g) => GenFromBits (f :+: g) where ...
instance (GenFromBits f, GenFromBits g) => GenFromBits (f :*: g) where ...
```