Composing instance- and type level functions

Joop Ringelberg 03-04-20 Version: 1

The problem

Consider the following two functions:

```
contextType :: ContextInstance ~~> ContextType
contextRole :: ContextType ~~~> RoleType
```

The former gives us the type of a context instance; the latter gives us all `RoleType`s defined for a context type. A reasonable use case is to compose these two:

```
Z = contextType >=> contextRole
```

However, the Purescript compiler flags a type error. The first function is *defined on the instance level*, while the latter is *defined on the type level*.

We will explore what these two terms mean and how we can reconcile both types so we can safely compose these functions.

Instance level functions

Let's examine the type of `contextType`. It expands according to the following types:

```
infixl 5 type TrackingObjectsGetter as ~~>
type TrackingObjectsGetter s o = s -> MonadPerspectivesQuery o
type MonadPerspectivesQuery =  ArrayT (WriterT (Array Assumption) MonadPerspectives)
```

to:

```
contextType :: ContextInstance -> ArrayT (WriterT (Array Assumption) MonadPerspectives) ContextType
```

So we see that this function computes a result in a monad stack that has `MonadPerspectives` at the bottom[1], `WriterT` above that and on top `ArrayT`. The `WriterT` monad transformer allows us to gather `Assumptions`. This is for query dependency tracking, explained elsewhere. As a spoiler: we do not track dependencies on the type level (because we do not allow type level queries through the API).

Furthermore, `ArrayT` allows us to work with Arrays of values and still compose functions as if they yielded single values. It abstracts away *un-determinedness* with respect to functional result, as it would be put in FP terms.

Let's turn to type level functions next

Type level functions

The type of contextRole expands as follows:

```
infixl 0 type TypeLevelGetter as ~~~>
type TypeLevelGetter s o = s -> ArrayT MonadPerspectives o
```

to:

```
contextRole :: ContextType -> ArrayT MonadPerspectives RoleType
```

This type is less involved. We merely get an undetermined (Array) result in `MonadPerspectives`. As said before, this type allows us to use Kleisli composition on such functions.

The problem, revisited

We can now state the problem clearly. We want to compose two functions with the following types:

```
contextType :: ContextInstance -> ArrayT (WriterT (Array Assumption) MonadPerspectives) ContextType
contextRole :: ContextType -> ArrayT MonadPerspectives RoleType
```

[1] Well, not really: `MonadPerspectives` itself is `ReaderT (AVar PerspectivesState) Aff`, but this need not concern us here as we will dig no deeper than `MonadPerspectives`.

The problem is that both stacks have `ArrayT` on top and `MonadPerspectives` as the bottom, but that the first stack has `WriterT (Array Assumption)` in between.

Notice that we apply the instance level function first and that it results in a type. This type is input for the type level computation. In other words, we move from instance- to type level. While the other way round is certainly possible, there seem to be far less use cases as we seldom ask for the instances of a type.

How do we make these types compatible?

runArrayT

We have function runArrayT to strip away the ArrayT layer:

```
runArrayT :: forall m a. ArrayT m a -> m (Array a)
```

Applying these to our functions, we get:

```
runArrayT <<< contextType :: ContextInstance -> (WriterT (Array Assumption) MonadPerspectives) (Array ContextType)
runArrayT <<< contextRole :: ContextType -> MonadPerspectives (Array RoleType)
```

Now, remembering we want to move from the instance- to the type level, it seems appropriate to reframe our problem as: how can we lift `runArrayT <<< contextRole` to the type of `runArrayT <<< contextType`? So, we want to lift

```
MonadPerspectives (Array RoleType)
```

To:

```
WriterT (Array Assumption) MonadPerspectives) (Array RoleType)
```

That is actually not too difficult, we can just lift `runArrayT <<< contextRole`:

```
lift <<< runArrayT <<< contextRole :: ContextType -> (WriterT (Array Assumption) MonadPerspectives) (Array RoleType)
```

We're almost done. We now want to abstract over the `Array RoleType` result, so we can deal implicitly with the un-determinedness of the result. That's what `ArrayT` is for:

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType -> ArrayT (WriterT (Array Assumption)
MonadPerspectives) RoleType
```

Reducing this long expression using our type definitions, we get:

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType -> MonadPerspectivesQuery RoleType
```

Or

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType ~~> RoleType
```

And we're done: we now have a function we can compose with contextType:

```
f :: ContextInstance ~~> RoleType
f = contextType >=> ArrayT <<< lift <<< runArrayT <<< contextRole
```

End result: lifting from instance- to type level

As this is a pattern of composition we will encounter quite a few times, it is worthwhile to introduce a special lifting function to abstract the pattern:

```
liftToInstanceLevel :: forall s o. (s ~~~> o) -> (s ~~> o)
liftToInstanceLevel f = ArrayT <<< lift <<< runArrayT <<< f
```

So we can rewrite our function f above as:

```
f :: ContextInstance ~~> RoleType
f = contextType >=> liftToInstanceLevel contextRole
```