Closed world instances

From HaskellWiki

By default, Haskell type classes use the open world assumption: It is possible to add instances for more and more types. Sometimes you need a type class with a restricted set of instances, that is, a closed world assumption. If the set of instances is restricted then you can do a complete case analysis of all types that inhabit the class. The reverse is also true: If you can do a complete case analysis on all instance types then you have a restricted set of instances. This is the key for declaring such a class. We only declare one method, namely the method for doing a complete case analysis.

1 Method

As an example we use type-level natural numbers. We want to define a class that only allows type-level natural numbers as instances. Here it is:

```
data Zero
data Succ n

class Nat n where
   switch ::
      f Zero ->
      (forall m. Nat m => f (Succ m)) ->
      f n

instance Nat Zero where
   switch x _ = x

instance Nat n => Nat (Succ n) where
   switch _ x = x
```

That's all. You will not be able to define more instances of Nat,

since you will not be able to implement the
`switch`
method for them.

I do not need more methods in Nat,

since I can express everything by the type case analysis provided by
`switch`
.

I can implement any method on Nat types

using a newtype around the method which instantiates the
`f`
.

This also means that users of the Nat class can write new methods without extending the Nat class.

Here is an example of how to use
`switch`

```
:
type family Add n m :: *
type instance Add Zero m = m
type instance Add (Succ n) m = Succ (Add n m)


data Vec n a = Vec

decons :: Vec (Succ n) a -> (a, Vec n a)

cons :: a -> Vec n a -> Vec (Succ n) a


newtype
   Append m a n =
      Append {runAppend :: Vec n a -> Vec m a -> Vec (Add n m) a}

append :: Nat n => Vec n a -> Vec m a -> Vec (Add n m) a
append =
   runAppend $
   switch
      (Append $ \_empty x -> x)
      (Append $ \x y ->
          case decons x of
             (a,as) -> cons a (append as y))
```

Please note, that the type parameter `n` must be the last one! The entire purpose of the `Append` type is to tell `switch` on which type it shall perform the case analysis.

2 Proofs

Since the set of class instances is restricted we can conduct proofs about types that are instances of the class. In the open world assumption it is not possible to conduct most proofs since every statement could be invalidated by new class instances. Here are some examples on commutativity and associativity:

```
newtype
   RightZeroForth a n =
      RightZeroForth
         {runRightZeroForth :: Vec (Add n Zero) a -> Vec n a}

rightZeroForth ::
   (Nat n) => Vec (Add n Zero) a -> Vec n a
rightZeroForth =
   runRightZeroForth $
   switch
      (RightZeroForth id)
      (RightZeroForth $ \x ->
         case decons x of (y,ys) -> cons y $ rightZeroForth ys)


newtype
   RightZeroBack a n =
      RightZeroBack
         {runRightZeroBack :: Vec n a -> Vec (Add n Zero) a}
```

```
rightZeroBack ::
   (Nat n) => Vec n a -> Vec (Add n Zero) a
rightZeroBack =
   runRightZeroBack $
   switch
      (RightZeroBack id)
      (RightZeroBack $ \x ->
         case decons x of (y,ys) -> cons y $ rightZeroBack ys)


data Forall n = Forall

forallPred :: Forall (Succ n) -> Forall n
forallPred Forall = Forall


newtype
   RightSuccBack m a n =
      RightSuccBack
         {runRightSuccBack ::
            Forall n -> Forall m ->
            Vec (Succ (Add n m)) a -> Vec (Add n (Succ m)) a}

{-
Succ (Add (Succ n) m)
 -> Succ (Succ (Add n m))
 -> Succ (Add n (Succ m))
 -> Add (Succ n) (Succ m)
-}

rightSuccBack ::
   (Nat n) =>
   Forall n -> Forall m ->
   Vec (Succ (Add n m)) a -> Vec (Add n (Succ m)) a
rightSuccBack =
   runRightSuccBack $
   switch
      (RightSuccBack $ const $ const id)
      (RightSuccBack $ \n m x ->
         case decons x of (y,ys) -> cons y $ rightSuccBack (forallPred n) m ys)


newtype
   Commute m a n =
      Commute
         {runCommute ::
            Forall n -> Forall m ->
            Vec (Add n m) a -> Vec (Add m n) a}

{-
Add (Succ n) m
 -> Succ (Add n m)
 -> Succ (Add m n)
 -> Add m (Succ n)
-}

commute ::
   (Nat n, Nat m) =>
   Forall n -> Forall m -> Vec (Add n m) a -> Vec (Add m n) a
commute =
   runCommute $
   switch
      (Commute $ const $ const rightZeroBack)
      (Commute $ \n m x ->
```

```
         case decons x of
            (y,ys) ->
               rightSuccBack m (forallPred n) $
               cons y $ commute (forallPred n) m ys)


newtype
   AssociateRight m k a n =
      AssociateRight
         {runAssociateRight ::
            Forall n -> Forall m -> Forall k ->
            Vec (Add n (Add m k)) a -> Vec (Add (Add n m) k) a}

associateRight ::
   (Nat n, Nat m) =>
   Forall n -> Forall m -> Forall k ->
   Vec (Add n (Add m k)) a -> Vec (Add (Add n m) k) a
associateRight =
   runAssociateRight $
   switch
      (AssociateRight $ const $ const $ const id)
      (AssociateRight $ \n m k x ->
         case decons x of
            (y,ys) -> cons y $ associateRight (forallPred n) m k ys)


newtype
   AssociateLeft m k a n =
      AssociateLeft
         {runAssociateLeft ::
            Forall n -> Forall m -> Forall k ->
            Vec (Add (Add n m) k) a -> Vec (Add n (Add m k)) a}

associateLeft ::
   (Nat n, Nat m) =>
   Forall n -> Forall m -> Forall k ->
   Vec (Add (Add n m) k) a -> Vec (Add n (Add m k)) a
associateLeft =
   runAssociateLeft $
   switch
      (AssociateLeft $ const $ const $ const id)
      (AssociateLeft $ \n m k x ->
         case decons x of
            (y,ys) -> cons y $ associateLeft (forallPred n) m k ys)
```

3 See also

- Haskell-Cafe on closed world instances, closed type families (http://www.haskell.org/pipermail/haskell-cafe/2013-April/107344.html)
- Haskell-Cafe on Lightweight type-level dependent programming in Haskell (http://www.haskell.org/pipermail/haskell-cafe/2009-June/062692.html)
- Haskell-Cafe on Typed Lambda-Expressions withOUT GADTs (http://www.haskell.org/pipermail/haskell-cafe/2005-January/008239.html)
- Bruno Oliveira and Jeremy Gibbons: TypeCase: A Design Pattern for Type-Indexed Functions (http://ropas.snu.ac.kr/~bruno/papers/Typecase.pdf)

Categories:

- Type-level programming

- Idioms
