XMonad KeyMap QuasiQuoter

September 8, 2020

Abstract

Play around with the idea of using a QuasiQuoter for key bindings in xmonad configurations. Currently, it is extremely limited (it can only bind keys to spawn actions), has some very overcomplicated nonsense, and is not tested well.

Source files are literate Haskell primarily so I can keep notes of what all this junk is actually supposed to be doing and why it's doing it that way.

Not actually intended for practical use. Maybe after lots of expansion, simplification, and testing, but certainly not now.

Contents

Part I
Internals

1 Types

Custom data types and structures used throughout.
TODO: can these be split into two seperate modules?

```haskell
module Types where

import Data.Bifoldable

import Data.List.NonEmpty (NonEmpty (..), (<|), nonEmpty)
import Data.Map           (Map, elemAt, foldMapWithKey, mapKeysMonotonic, singleton, unionWith
```

1.1 Key actions

Data type representing actions so that they can be lifted to Template Haskell experessions later.

```haskell
data KeyAction = ActionSpawn String
                 deriving (Eq, Show)
```

1.2 Submaps

A nice feature of *XMonad.Util.EZConfig.additionalKeysP* is that it allows for the easy binding of sequences of keys, via some “sugar” on top of *XMonad.Actions.Submap.submap*. I want to recreate this feature here.
The current solution is way too complicated and needs to be simplified.

A key may be bound directly to an action or be bound to a submap. This choice is represented by the *ESM* type. *SM* wraps a plain *Map*'s values with *ESM*.

1.2.1 *ESM*

Either a submap (*LSM*) or just a value (*RSM*).
TODO: could a recursive type be used instead of this *and* a **newtype**?

$$\textbf{data}\ \mathit{ESM}\ k\ v = \mathit{LSM}\ (\mathit{SM}\ k\ v) \mid \mathit{RSM}\ v\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

1.2.2 *SM*

The *SM* type wraps the keys of a *Map* with *ESM*.

$$\textbf{newtype}\ \mathit{SM}\ k\ v = \mathit{SM}\ \{\mathit{unSM} :: \mathit{Map}\ k\ (\mathit{ESM}\ k\ v)\}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

1.2.3 Instances

Semigroup and monoid instances inherit an *Ord* constraint from *Map*.

```haskell
instance Ord k => Semigroup (ESM k v) where
  LSM x <> LSM y = LSM (x <> y)
instance Ord k => Semigroup (SM k v) where
  SM m <> SM n = SM $ unionWith (<>) m n
instance Ord k => Monoid (SM k v) where
  mempty = SM mempty
```

Folding, mapping, and traversing *ESM*s recurses on left values and uses rights directly. *SM* instances just have to apply functions recursively to their internal *Map*s and *ESM*s.

```haskell
instance Foldable    (ESM k) where foldMap f (RSM x)  =             f x
                                   foldMap f (LSM sm) =             foldMap f sm
instance Functor     (ESM k) where fmap    f (RSM x)  = RSM $   f x
                                   fmap    f (LSM sm) = LSM $   fmap f sm
instance Traversable (ESM k) where traverse f (RSM x)  = RSM <$> f x
                                   traverse f (LSM sm) = LSM <$> traverse f sm
instance Foldable    (SM k) where foldMap f (SM m) =          foldMap (foldMap f) m
instance Functor     (SM k) where fmap    f (SM m) = SM $   fmap    (fmap    f) m
instance Traversable (SM k) where traverse f (SM m) = SM <$> traverse (traverse f) m
```

1.2.4 Probably superfluous instances

I don't think any of these are used, will be ever used, or should ever be used, but they are defined here anyway for... fun.
ESM can be lifted to an applicative. When combining two maps, each of the right values of the "argument" map are passed to each of the right values of the "function" map. A similar method is used to lift *ESM* to a monad.

```
instance Applicative (ESM k) where
  pure = RSM
  RSM f  <*> RSM x        = RSM (f x)
  RSM f  <*> LSM sm       = LSM (fmap f sm)
  LSM sm <*> RSM x        = LSM (fmap ($x) sm)
  fs     <*> LSM (SM m)   = LSM (SM (fmap (fs<*>) m))
instance Monad (ESM k) where
  RSM x        >>= f = f x
  LSM (SM m)   >>= f = LSM (SM (fmap (>>=f) m))
```

Data.Map.foldMapWithKeys can easily be reworked into *bifoldMap* and a *Bifoldable* instance.

```
instance Bifoldable SM where
  bifoldMap f g (SM m) = foldMapWithKey (\k v -> f k <> bifoldMap f g v) m
instance Bifoldable ESM where
  bifoldMap _ g (RSM x)  = g x
  bifoldMap f g (LSM sm) = bifoldMap f g sm
```

In theory, a combination of *mapKeys* and plain *fmap* theoretically can be composed into *bimap*. However, *mapKeys* has an *Ord* constraint and will not work with *Bifunctor*.
Bitraversing is more insteresting. There is no function provided by *Data.Map* that allows for traversing over keys. Defining one, however, would not be too complicated. Two possibilities are:

Reconstructing the *Map* within the applicative. A function declared like this would work similarly to *mapKeysMonotonic*—and have the same restriction that function being applied must be monotonic, so its no good.

```
import Data.Map.Internal (Map (..))
bitraverseMonotonic :: Applicative f => (k -> f g) -> (v -> f w) -> Map k v -> f (Map g w)
bitraverseMonotonic _ _ Tip              = pure Tip
bitraverseMonotonic f g (Bin s k v l r) = Bin <$> pure s <*> f k <*> g v
                                              <*> bitraverseMonotonic f g l
                                              <*> bitraverseMonotonic f g r
```

Mapping each element into a singleton and folding results. Using *foldMapWithKey* and the *Ap* data type (from *Data.Monoid*), this task is trivial. Sequentially applying *singleton* to the new lifted keys, then wrapping this new

Map in *Ap* produces a monoid that is easily folded into a single *Map*, as long as *Map* itself is a monoid—which it only is if its keys are *Ord*erable, thus having the same constraint as the previous method and being equally useless for this purpose.

```
import Data.Monoid (Ap (..))
traverseKeysMonoid :: (Applicative f, Ord g) => (k -> f g) -> Map k (ESM k a) -> f (Map g (ESM
traverseKeysMonoid f = getAp . foldMapWithKey go
  where go k v = Ap (singleton <$> f k <*> recurse v)
          recurse (RSM x)  = RSM <$> x
          recurse (LSM sm) = LSM <$> traverseKeysMonoid f sm
```

1.2.5 Other *Map* functions for *SM*

mapKeysMonotonic; an unsafe function that applies a given function over the keys of a submap. It does not check if the ordering of the keys has changed, so if they do change, the *Map* will be broken, hence it being unsafe.

```
smMapKeysMonotonic :: (k -> g) -> SM k a -> SM g a
smMapKeysMonotonic f (SM m) = SM $ fmap recurse $ mapKeysMonotonic f m
  where recurse (RSM x) = RSM x
        recurse (LSM x) = LSM $ smMapKeysMonotonic f x
```

1.2.6 Creating and eliminating submaps

A "singleton" submap can be created by providing a "path" and its value, the path being a non-empty list of keys to the value.

```
toSM :: NonEmpty k -> t -> SM k t
toSM (k :| ks) = SM . singleton k . maybe pure (\p -> LSM . toSM p) (nonEmpty ks)
```

The inverse of the above, a path and its value can be extracted from a non-empty submap.

```
fromSM :: SM k t -> Maybe (NonEmpty k, t)
fromSM (SM m) | null m                          = Nothing
              | (k, RSM x)  <- head' m          = Just (k :| [], x)
              | (k, LSM sm) <- head' m,
                Just (ks, x) <- fromSM sm       = Just (k <| ks, x)
              | otherwise                       = Nothing
  where head' = elemAt 0
```

2 Parsing

Parsers used for producing data structres from input to the QuasiQuoter.
The syntax here is more similar to the ones found in vis or lf than *additionalKeysP*.
TODO: better error messages for just about everything here.

```
module Parsers where

import Control.Monad
import Control.Applicative
import Text.ParserCombinators.Parsec hiding ((<|>), many, optional)
import           Data.List
import           Data.List.NonEmpty (NonEmpty, some1)
import qualified Data.List.NonEmpty as NE
import           Data.Set           (Set)
import qualified Data.Set           as S
import XMonad
import Types
```

2.1 Utilities

```
someOf    = some  ∘ oneOf
tryMany   = many  ∘ try
trySome   = some1 ∘ try
testAhead = λp → lookAhead (True <$ p) <|> pure False
```

2.2 Parsing regular keys

Some keys are allowed to specified literally, i.e., by the actual character instead of the symbolic name.

```
allowedLiterally = ['!' .. '~'] \\ "<->"
```

Parser for a single, non-whitespace, ASCII character.

```
asciiKeysym :: CharParser st KeySym
asciiKeysym = choice xs <?> "an ASCII character"
  where xs = zipWith (λc s → s <$ char c) ['!' .. '~'] [xK_exclam .. xK_asciitilde]
```

Parser for known symbolic key name.

```
keysymName :: CharParser st KeySym
keysymName = do
  str ← someOf allowedLiterally
  let sym = stringToKeysym str
```

```
    if sym ≡ noSymbol then fail $ "Invalid key symbol name \"" ++ str ++ "\""
                      else return sym
```

Parser that tries to read a symbolic name, or ASCII character if that fails.

```
keysym' :: CharParser st KeySym
keysym' = try keysymName <|> asciiKeysym
```

Same as *keysym'* except the names are surrounded with angled brackets.
TODO: this *isSpecial* pattern is ugly.

```
keysym :: CharParser st KeySym
keysym = do
  isSpecial ← testAhead (char '<')
  if isSpecial then char '<' *> keysymName <* char '>'
             else  asciiKeysym
```

2.3 Parsing modifier keys

Parser that reads an abbreviation for a modifier key and returns its mask.

```
keymask' :: CharParser st KeyMask
keymask' = choice xs
  where xs = zipWith (λs m → m <$ try (string s))
             ["S","C","M1","M2","M3","M4","M5"]
             [shiftMask, controlMask, mod1Mask, mod2Mask, mod3Mask, mod4Mask, mod5Mask]
```

Same as *keymask'* but also accepts "M" for *modMask*, represented as *Nothing*.

```
keymask :: CharParser st (Maybe KeyMask)
keymask = Just <$> keymask' <|> Nothing <$ char 'M'
```

2.4 Parsing key combinations

Parser that either reads a single *asciiKeysym*, or many (or no) hypen-seperated *keymask*s and a *keysym'* between angled brackets.

```
keyCombo :: CharParser st (Set (Maybe KeyMask), KeySym)
keyCombo = do
  isSpecial ← testAhead (char '<')
  if isSpecial then char '<' *> pure withMasks <*> tryMany (keymask <* char '-') <*> keysym' <* ch
             else  noMasks <$> asciiKeysym
             where noMasks   = λs   → (mempty,       s)
                   withMasks = λm s → (S.fromList m, s)
```

2.5 Parsing actions

Parser that reads a *spawn* action.

```
keyActionSpawn :: CharParser st KeyAction
keyActionSpawn = char '$' *>
  (ActionSpawn <$> ((:) <$> (noneOf "|" <?> "a command")
                        <*> (manyTill anyChar $ lookAhead $ void (char '|') <|> eof)))
```

Parser that reads any action.

```
keyAction :: CharParser st KeyAction
keyAction = keyActionSpawn
```

2.6 Parsing key bindings

Parser that reads a sequence space-sperated of key combinations followed by an action.

```
keybind :: CharParser st (SM (Set (Maybe KeyMask), KeySym) KeyAction)
keybind = toSM <$> trySome (keyCombo <* some space) <*> keyAction
```

Parser that reads a list of *keybind*s, seperated by pipes (i.e., `|`).

```
keybinds :: CharParser st (SM (Set (Maybe KeyMask), KeySym) KeyAction)
keybinds = spaces *> pure mconcat <*> keybind `sepBy1` (char '|' *> spaces) <* spaces
```

2.7 Putting it all together

Wraps *keybinds* with *parse*. Intended to the be main export to other modules (so they do not also need to import Parsec).

```
parseKeymap :: String -> Either ParseError (SM (Set (Maybe KeyMask), KeySym) KeyAction)
parseKeymap = parse keybinds ""
```

3 Lifting

This module defines functions for lifting values into Template Haskell typed expressions.

```
{-# LANGUAGE TemplateHaskell #-}
module Lift where
```

The "goal" function is one that works on the output of the parser and returns an expression representing a lambda taking an *XConfig* and returning the key map; with a type signature

$$
\begin{aligned}
\mathit{goal} :: {} & \mathit{SM} \quad (\mathit{Set}\ (\mathit{Maybe}\ \mathit{KeyMask}), \mathit{KeySym})\ \mathit{KeyAction} \\
& \to \mathit{TExpQ}\ (\mathit{XConfig}\ \mathit{Layout} \to \mathit{Map}\ (\mathit{KeyMask}, \mathit{KeySym})\ (X\ ()))
\end{aligned}
$$

This function will be broken up into smaller "steps":

1. Lift the key actions to expressions representing X actions.
2. Lift the key combinations to expressions representing pairs of masks and symbols.
3. Lift the *ESM*s to expressions representing single X actions, taking right values directly and merging left ones with *submap*.
4. Lift the *SM*s to expressions representing plain *Map*s.
5. Finally, splice resulting expression into one with the *XConfig* lambda.

3.1 Imports

For the *Map* and *Set* data structures:

```
import           Data.Map    (Map)
import qualified Data.Map as M
import           Data.Set    (Set)
import qualified Data.Set as  S
```

For Template Haskell types and functions:

```
import Language.Haskell.TH.Lib
import Language.Haskell.TH.Syntax
```

For working with *KeyMask*s and *KeySym*s:

```
import XMonad           (KeyMask, KeySym, (.|.))
import Foreign.C.Types (CUInt)
```

For working with *XConfig*:

```
import XMonad (Layout, XConfig (XConfig, modMask))
```

For working with *KeyAction*s:

```
import XMonad                 (X, spawn)
import XMonad.Actions.Submap (submap)
```

Finally, custom types:

```
import Types
```

3.2 Lifting key actions

```
liftKeyAction :: KeyAction → TExpQ (X ())
liftKeyAction (ActionSpawn x) = [∨ spawn x ∨]
```

3.3 Lifting key combinations

Before key masks can be lifted, a *Lift* instance for its “root” type, *CUInt*, must be declared:

```
instance Lift CUInt where lift = litE ∘ integerL ∘ toInteger
```

Compared to the rest, this function takes an extra argument, *varName* :: *Name*. This represents the value of *modMask*.

```
liftKeyCombo :: Name → (Set (Maybe KeyMask), KeySym) → TExpQ (KeyMask, KeySym)
liftKeyCombo varName (set, sym) =
  let knownMasks    = foldr (λmask rest → maybe rest (.|.rest) mask) 0 set
      varExpression = TExp <$> varE varName
  in  if Nothing ∈ set
        then [∨ (knownMasks .|. $$(varExpression), sym) ∨]
        else [∨ (knownMasks, sym)                       ∨]
```

3.4 Lifting *ESM*s

Right values can be returned as-is, but left values require more work:

1. Recurse on child maps.
2. Convert the map to a (sorted) list of pairs.
3. Lift each pair in the list.
4. Lift the whole list.
5. Splice the list's expression into one turning it back into a map.
6. Splice the map's expression into one calling *submap* on it.

```
liftESM :: ESM (TExpQ (KeyMask, KeySym)) (TExpQ (X ())) → TExpQ (X ())
liftESM (RSM x)              = x
liftESM (LSM (SM rawMap)) =
  let rightMap     =     fmap liftESM rawMap
      rightList    =     M.toAscList rightMap
      rightList'   =     fmap   tupToExp rightList
                         where tupToExp (comboExp, actionExp) =
```

$$[\vee\ (\$\$(comboExp), \$\$(actionExp))\ \vee]$$

$$\begin{aligned}
&\quad rightExpression = && fmap\ TExp\ \$\ listE\ \$\ map\ (fmap\ unType)\ rightList' \\
&\quad mapExpression\ = && [\vee\ M.fromDistinctAscList\ \$\$\ (rightExpression)\ \vee] \\
&\mathbf{in}\ \ [\vee\ submap\ \$\$\ (mapExpression)\ \vee]
\end{aligned}$$

3.5 Lifting *SM*s

1. Call *liftESM* on child maps.
2. Convert the map to a (sorted) list of pairs.
3. Lift each pair in the list.
4. Lift the whole list.
5. Splice the list's expression into one turning it back into a map.

TODO: switch name with *liftSM*? feels like I've been using "submap" to refer more general functions, unlike this.

$$\begin{aligned}
&liftSubmaps :: SM\ \ (TExpQ\ (KeyMask, KeySym))\ (TExpQ\ (X\ ()))\\
&\qquad\qquad \to TExpQ\ (Map\ \ (KeyMask, KeySym)\ \ (X\ ()))\\
&liftSubmaps\ (SM\ rawMap) =\\
&\ \ \mathbf{let}\ flatMap \qquad\ = fmap\ liftESM\ rawMap\\
&\qquad flatList \qquad\ = M.toAscList\ flatMap\\
&\qquad flatList' \qquad\ = fmap\ \ \ tupToExp\ flatList\\
&\qquad\qquad\qquad\qquad \mathbf{where}\ tupToExp\ (comboExp, actionExp) =\\
&\qquad\qquad\qquad\qquad\qquad [\vee\ (\$\$(comboExp), \$\$(actionExp))\ \vee]\\
&\qquad listExpression\ \ = fmap\ TExp\ \$\ listE\ \$\ map\ (fmap\ unType)\ flatList'\\
&\qquad mapExpression = [\vee\ M.fromDistinctAscList\ \$\$\ (listExpression)\ \vee]\\
&\ \ \mathbf{in}\ \ mapExpression
\end{aligned}$$

3.6 Putting it all together

$$\begin{aligned}
&liftSM :: SM\ (Set\ (Maybe\ KeyMask), KeySym)\ KeyAction\\
&\qquad \to TExpQ\ (XConfig\ Layout \to Map\ (KeyMask, KeySym)\ (X\ ()))\\
&liftSM\ sm = \mathbf{do}\\
&\ \ mmVarName \leftarrow newName\ \texttt{"mm"}\\
&\ \ \mathbf{let}\ expressionSM\ \ = smMapKeysMonotonic\ (liftKeyCombo\ mmVarName)\ \$\ fmap\ liftKeyAction\ sm\\
&\qquad mmVarPattern = [p\ |\ XConfig\ \{modMask = \$(varP\ mmVarName)\}\ |]\\
&\qquad mapExpression = liftSubmaps\ expressionSM\\
&\ \ fmap\ TExp\ \$\ lamE\ [mmVarPattern]\ \$\ fmap\ unType\ mapExpression
\end{aligned}$$

Primary export of this module.

$$\begin{aligned}
&liftKeymap :: SM\ (Set\ (Maybe\ KeyMask), KeySym)\ KeyAction \to ExpQ\\
&liftKeymap = fmap\ unType \circ liftSM
\end{aligned}$$

Part II
Exports

4 QuasiQuoters

Module that defines the QuasiQuoter.

$$\textbf{module}\ \mathit{QQ}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Language.Haskell.TH.Quote}$$

$$\textbf{import}\ \mathit{Parsers}$$
$$\textbf{import}\ \mathit{Lift}$$

Quotes a keymap from a string or throws an error if it can't.

$$\mathit{quoteKeymap} = \mathit{either}\ (\mathit{fail} \circ \mathit{show})\ \mathit{liftKeymap} \circ \mathit{parseKeymap}$$

The QuasiQuoter. It does not support quoting anything other than expressions.

```
keymap :: QuasiQuoter
keymap = QuasiQuoter
  { quoteExp  = quoteKeymap
  , quotePat  = fail "keymap can only quote expressions."
  , quoteDec  = fail "keymap can only quote expressions."
  , quoteType = fail "keymap can only quote expressions."
  }
```

5 Utilities

Utility functions to be exported by the main module alongside the quasiquoter.

$$\textbf{module}\ \mathit{Utilities}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Data.Map}\ (\mathit{Map})$$
$$\textbf{import}\ \mathit{XMonad}$$

Roughly equivalent to *XMonad.Util.EZConfig.addtionalKeys* except that the argument here is the same type as *keys* from *XConfig*.

```
additionalKeys ::    XConfig l
                  -> (XConfig Layout -> Map (KeyMask, KeySym) (X ()))
                  ->   XConfig l
addtionalKeys xc newKeys = xc {keys = newKeys <+> keys xc}
```

The same function, with the arguments flipped. Fits better with functions like *withUrgencyHook*, etc.

$$\mathit{withAdditionalKeys} = \mathit{flip}\ \mathit{addtionalKeys}$$