ST monáda

Existuje monáda `Control.Monad.ST` s typem `data ST s a`. Také existuje funkce
`runST :: (forall s. ST s a) -> a`, která provede výpočet. Všimněte si toho `forall`...
`stToIO :: ST RealWorld a -> IO a`
Protože ve skutečnosti je `IO` monáda jenom `ST RealWord`, popsané odkazy a pole i v monádě `IO` jsou speciální případ nyní popisovaných polí a odkazů v `ST` monádě.

```
module Data.STRef
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
modifySTRef ref f = writeSTRef ref . f =<< readSTRef ref
swap::STRef s a->STRef s a->ST s ()
swap a b = do x<-readSTRef a; y<-readSTRef b; writeSTRef a y; writeSTRef b x
```

A v modulu `Data.Array.ST` jsou pole uvnitř `ST` monády:

```
data STArray s i e
data STUArray s i e
instance MArray (STArray s) e (ST s)
instance MArray (STUArray s) boxed (ST s)

runSTArray::Ix i=>(forall s. ST s (STArray s i e)) -> Array i e
runSTUArray :: Ix i => (forall s. ST s (STUArray s i e)) -> UArray i e

newArray ::...=>(i, i)->e->m (a i e) newListArray::...=> (i, i)->[e]->m (a i e)
readArray::...=>a i e->i->m e        writeArray  ::...=> a i e ->i->e-> m ()
getBounds::...=>a i e->m (i, i); getElems::...->m [e]; getAssocs::...->m [(i,e)]
```

`count::[Int]->Array Int Int` Řekněme, že čísla jsou 0..9

```
count n = runSTArray $ do a <- newArray (0,9) 0
                          mapM_ (\i->readArray a i >>= writeArray a i . (+1)) n
                          return a

freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)

unsafeFreeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
unsafeThaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

Tyto funkce fungují efektivně pro konverze

```
  Data.Array.IO.IOUArray <-> Data.Array.Unboxed.UArray
  Data.Array.ST.STUArray <-> Data.Array.Unboxed.UArray
  Data.Array.IO.IOArray <-> Data.Array.Array
  Data.Array.ST.STArray <-> Data.Array.Array

array bnds assocs = runSTArray $ do a <- newArray_ bnds
                                    mapM_ (uncurry $ writeArray a) assocs
                                    return a
runSTArray starr = runST (starr >>= unsafeFreeze)

import Control.Monad
import Data.Array.IArray
import Data.Array.ST

performSwaps a swaps = runSTArray $
  do b <- newArray_ (bounds a)
     forM_ (indices a) $ \i -> writeArray b i (a!i)
     forM_ swaps $ \(i,j) -> swapArray b i j
     return b

  where swapArray b i j = do x <- readArray b i
                             y <- readArray b j
                             writeArray b i y
                             writeArray b j x
```

```
                          Monadické funkce v Prelude
                          --------------------------
class Monad where ...
sequence      :: Monad m => [m a] -> m [a]
sequence_     :: Monad m => [m a] -> m ()
mapM          :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_         :: Monad m => (a -> m b) -> [a] -> m ()

                                     Monad
                                     -----
when          :: Monad m => Bool -> m () -> m ()
unless        :: Monad m => Bool -> m () -> m ()
ap            :: Monad m => m (a -> b) -> m a -> m b

guard         :: MonadPlus m => Bool -> m ()
msum          :: MonadPlus m => [m a] -> m a

mapAndUnzipM  :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
zipWithM      :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_     :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
foldM         :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
filterM       :: Monad m => (a -> m Bool) -> [a] -> m [a]

liftM         :: Monad m => (a -> b) -> (m a -> m b)
liftM2        :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3        :: Monad m => (a -> b -> c -> d) ->
                            (m a -> m b -> m c -> m d)
liftM4        :: Monad m => (a -> b -> c -> d -> e) ->
                            (m a -> m b -> m c -> m d -> m e)
liftM5        :: Monad m => (a -> b -> c -> d -> e -> f) ->
                            (m a -> m b -> m c -> m d -> m e -> m f)

                                 Control.Monad
                                 -------------
(>=>)         :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
forever       :: Monad m => m a -> m b
forM          :: Monad m => [a] -> (a -> m b) -> m [b]
forM_         :: Monad m => [a] -> (a -> m b) -> m ()
replicateM    :: Monad m => Int -> m a -> m [a]
replicateM_   :: Monad m => Int -> m a -> m ()
```