Lightweight Monadic Regions

Safely using scarce resources

Bas van Dijk
v.dijk.bas@gmail.com

Dutch HUG Day, 24 April 2010

Background

- Invented by: **Oleg Kiselyov & Chung-chieh Shan**
- `cabal install regions`

Background

- Invented by: **Oleg Kiselyov & Chung-chieh Shan**
- `cabal install` **`regions`**

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example
Specification

1. open two files for reading, one of them a configuration file;
2. read the name of an output file (such as the log file) from the configuration file;
3. open the output file and zip the contents of both input files into the output file;
4. close the configuration file;
5. copy the rest, if any, of the other input file to the output file;
6. close both the output and the input file.

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example
Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example

Example run

input.txt

```
a
b
c
d
e
f
g
h
i
j
k
```

config.txt

```
out.txt
1
2
3
4
```

out.txt

```
1
a
2
b
3
c
4
d
e
f
g
h
i
j
k
```

Guiding Example
Implementation

```
test :: IO ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← test_internal hIn
  till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: Handle → IO Handle
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn  ≫= hPutStrLn hOut
  return hOut
```

```
till :: Monad m ⇒ m Bool → m () → m ()
till condition iteration = loop
  where
    loop = do b ← condition
              if b
                then return ()
                else iteration ≫ loop
```

Guiding Example
Implementation (with close)

```
test :: IO ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← test_internal hIn
  till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut
  hClose hOut
  hClose hIn

test_internal :: Handle → IO Handle
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn  ≫= hPutStrLn hOut
  hClose hCfg
  return hOut
```

Guiding Example

Implementation (with exception handling)

```
test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: Handle → IO Handle
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

Guiding Example
Implementation

```
test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: Handle → IO Handle
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

IOModes

Only read from read-only handles!

```
test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: Handle → IO Handle
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

IOModes

Only write to write-only handles!

```
test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: Handle → IO Handle
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

IOModes

Encode IOModes in types

- `cabal install` **`explicit-iomodes`**
- **newtype** *Handle ioMode* = *Handle* (*System.IO.Handle*)
  - **data** *ReadMode*
    data *WriteMode*
    data *AppendMode*
    data *ReadWriteMode*
- *openFile* :: *FilePath* → *IOMode ioMode* → *IO* (*Handle ioMode*)
  - **data** *IOMode ioMode* **where**
    ReadMode :: *IOMode ReadMode*
    WriteMode :: *IOMode WriteMode*
    AppendMode :: *IOMode AppendMode*
    ReadWriteMode :: *IOMode ReadWriteMode*

IOModes

Encode IOModes in types

- `cabal install` **`explicit-iomodes`**
- **newtype** *Handle ioMode* = *Handle* (*System.IO.Handle*)
  - **data** *ReadMode*
    data *WriteMode*
    data *AppendMode*
    data *ReadWriteMode*
- *openFile* :: *FilePath* → *IOMode ioMode* → *IO* (*Handle ioMode*)
  - **data** *IOMode ioMode* **where**
    ReadMode :: *IOMode ReadMode*
    WriteMode :: *IOMode WriteMode*
    AppendMode :: *IOMode AppendMode*
    ReadWriteMode :: *IOMode ReadWriteMode*

IOModes

Encode IOModes in types

- `cabal install` **`explicit-iomodes`**
- **newtype** *Handle ioMode* = *Handle* (*System.IO.Handle*)
  - **data** *ReadMode*
    data *WriteMode*
    data *AppendMode*
    data *ReadWriteMode*
- *openFile* :: *FilePath* → *IOMode ioMode* → *IO* (*Handle ioMode*)
  - **data** *IOMode ioMode* **where**
    ReadMode :: *IOMode ReadMode*
    WriteMode :: *IOMode WriteMode*
    AppendMode :: *IOMode AppendMode*
    ReadWriteMode :: *IOMode ReadWriteMode*

IOModes

Encode IOModes in types

- `cabal install` **`explicit-iomodes`**
- **newtype** *Handle ioMode* = *Handle* (*System.IO.Handle*)
  - **data** *ReadMode*
    data *WriteMode*
    data *AppendMode*
    data *ReadWriteMode*
- *openFile* :: *FilePath* → *IOMode ioMode* → *IO* (*Handle ioMode*)
  - **data** *IOMode ioMode* **where**
    ReadMode :: *IOMode ReadMode*
    WriteMode :: *IOMode WriteMode*
    AppendMode :: *IOMode AppendMode*
    ReadWriteMode :: *IOMode ReadWriteMode*

IOModes

Encode IOModes in types

- `cabal install` **`explicit-iomodes`**
- **newtype** *Handle ioMode* = *Handle* (*System.IO.Handle*)
  - **data** *ReadMode*
    data *WriteMode*
    data *AppendMode*
    data *ReadWriteMode*
- *openFile* :: *FilePath* → *IOMode ioMode* → *IO* (*Handle ioMode*)
  - **data** *IOMode ioMode* **where**
    ReadMode :: *IOMode ReadMode*
    WriteMode :: *IOMode WriteMode*
    AppendMode :: *IOMode AppendMode*
    ReadWriteMode :: *IOMode ReadWriteMode*

IOModes
Constrain operations

- $$\begin{aligned} &\mathit{hGetLine} :: \mathit{ReadModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{IO}\ \mathit{String} \\ &\mathit{hIsEOF} :: \mathit{ReadModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{IO}\ \mathit{Bool} \\ &\mathit{hPutStrLn} :: \mathit{WriteModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{String} \rightarrow \mathit{IO}\ () \end{aligned}$$

- **class** *ReadModes ioMode*
  class *WriteModes ioMode*

  instance *ReadModes ReadMode*
  instance *ReadModes ReadWriteMode*

  instance *WriteModes WriteMode*
  instance *WriteModes AppendMode*
  instance *WriteModes ReadWriteMode*

IOModes
Constrain operations

- $$\begin{aligned} &\mathit{hGetLine} :: \mathit{ReadModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{IO}\ \mathit{String} \\ &\mathit{hIsEOF} :: \mathit{ReadModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{IO}\ \mathit{Bool} \\ &\mathit{hPutStrLn} :: \mathit{WriteModes}\ \mathit{ioMode} \Rightarrow \mathit{Handle}\ \mathit{ioMode} \rightarrow \mathit{String} \rightarrow \mathit{IO}\ () \end{aligned}$$

- **class** *ReadModes ioMode*
  class *WriteModes ioMode*

  instance *ReadModes ReadMode*
  instance *ReadModes ReadWriteMode*

  instance *WriteModes WriteMode*
  instance *WriteModes AppendMode*
  instance *WriteModes ReadWriteMode*

IOModes
Only one import!

```
import System.IO.ExplicitIOModes

test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn >>= hPutStrLn hOut

test_internal :: ReadModes ioMode ⇒
                 Handle ioMode → IO (Handle WriteMode)
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg >>= hPutStrLn hOut
        hGetLine hIn  >>= hPutStrLn hOut
      return hOut
```

IOModes

Inferred types change (*hIn*)

```
import System.IO.ExplicitIOModes

test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn >>= hPutStrLn hOut

test_internal :: ReadModes ioMode ⇒
                 Handle ioMode → IO (Handle WriteMode)
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg >>= hPutStrLn hOut
        hGetLine hIn  >>= hPutStrLn hOut
      return hOut
```

IOModes

Inferred types change (*hOut*)

```
import System.IO.ExplicitIOModes

test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn >>= hPutStrLn hOut

test_internal :: ReadModes ioMode ⇒
                 Handle ioMode → IO (Handle WriteMode)
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg >>= hPutStrLn hOut
        hGetLine hIn  >>= hPutStrLn hOut
      return hOut
```

Regions

Don't forget to close!

```
test :: IO ()
test =
  bracket (openFile "input.txt" ReadMode) hClose $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: ReadModes ioMode ⇒
                 Handle ioMode → IO (Handle WriteMode)
test_internal hIn = do
  bracket (openFile "config.txt" ReadMode) hClose $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

Regions
withFile

$$\mathit{withFile} :: \mathit{FilePath} \to \mathit{IOMode} \to (\mathit{Handle} \to \mathit{IO}\ a) \to \mathit{IO}\ a$$
$$\mathit{withFile}\ \mathit{fp}\ \mathit{ioMode} = \mathit{bracket}\ (\mathit{openFile}\ \mathit{fp}\ \mathit{ioMode})\ \mathit{hClose}$$

Regions

Use *withFile* to abstract close

```
test :: IO ()
test =
  withFile "input.txt" ReadMode $ λhIn →
    bracket (test_internal hIn) hClose $ λhOut →
      till (hIsEOF hIn) $ hGetLine hIn ≫= hPutStrLn hOut

test_internal :: ReadModes ioMode ⇒
                 Handle ioMode → IO (Handle WriteMode)
test_internal hIn = do
  withFile "config.txt" ReadMode $ λhCfg → do
    fname ← hGetLine hCfg
    bracketOnError (openFile fname WriteMode) hClose $ λhOut → do
      till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
        hGetLine hCfg ≫= hPutStrLn hOut
        hGetLine hIn  ≫= hPutStrLn hOut
      return hOut
```

Regions

Extend *openFile*

$$openFile :: FilePath \rightarrow IOMode\ ioMode \rightarrow IO\ (E.Handle\ ioMode)$$

Regions

- ```
  openFile :: MonadCatchIO pr ⇒
              FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
  ```
- ```
  newtype RegionT pr a = RegionT
    { unRegionT :: ReaderT (IORef [AnyHandle]) pr a}
  ```
- ```
  deriving (Functor
            , Applicative
            , Alternative
            , Monad
            , MonadPlus
            , MonadFix
            , MonadTrans
            , MonadIO
            , MonadCatchIO
            )
  ```
- ```
  data AnyHandle = ∀ ioMode. Any (Handle ioMode)
  ```

Regions

- $openFile :: MonadCatchIO\ pr \Rightarrow FilePath \rightarrow IOMode\ ioMode \rightarrow RegionT\ pr\ (E.Handle\ ioMode)$
- **newtype** $RegionT\ pr\ a = RegionT\ \{unRegionT :: ReaderT\ (IORef\ [AnyHandle])\ pr\ a\}$
- **deriving** ($Functor$, $Applicative$, $Alternative$, $Monad$, $MonadPlus$, $MonadFix$, $MonadTrans$, $MonadIO$, $MonadCatchIO$)
- **data** $AnyHandle = \forall\ ioMode.\ Any\ (Handle\ ioMode)$

Regions

- $$openFile :: MonadCatchIO\ pr \Rightarrow FilePath \rightarrow IOMode\ ioMode \rightarrow RegionT\ pr\ (E.Handle\ ioMode)$$
- **newtype** $RegionT\ pr\ a = RegionT\ \{unRegionT :: ReaderT\ (IORef\ [AnyHandle])\ pr\ a\}$
- **deriving** (*Functor*
  , *Applicative*
  , *Alternative*
  , *Monad*
  , *MonadPlus*
  , *MonadFix*
  , *MonadTrans*
  , *MonadIO*
  , *MonadCatchIO*
)
- **data** $AnyHandle = \forall\ ioMode.\ Any\ (Handle\ ioMode)$

Regions

- $$openFile :: MonadCatchIO\ pr \Rightarrow FilePath \rightarrow IOMode\ ioMode \rightarrow RegionT\ pr\ (E.Handle\ ioMode)$$
- **newtype** $RegionT\ pr\ a = RegionT\ \{unRegionT :: ReaderT\ (IORef\ [AnyHandle])\ pr\ a\}$
- **deriving** (*Functor*
  , *Applicative*
  , *Alternative*
  , *Monad*
  , *MonadPlus*
  , *MonadFix*
  , *MonadTrans*
  , *MonadIO*
  , *MonadCatchIO*
)
- **data** $AnyHandle = \forall\ ioMode.\ Any\ (Handle\ ioMode)$

Regions

Implementation of *openFile*

```
openFile :: MonadCatchIO pr ⇒
            FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
openFile fp ioMode = block $ do
  h ← liftIO $ E.openFile fp ioMode
  register h
  return h

register :: MonadCatchIO pr ⇒ Handle ioMode → RegionT pr ()
register h = RegionT $ ask >>= liftIO ∘ flip modifyIORef (Any h:)
```

Regions

Call native *openFile*

```
openFile :: MonadCatchIO pr ⇒
            FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
openFile fp ioMode = block $ do
  h ← liftIO $ E.openFile fp ioMode
  register h
  return h

register :: MonadCatchIO pr ⇒ Handle ioMode → RegionT pr ()
register h = RegionT $ ask >>= liftIO ∘ flip modifyIORef (Any h:)
```

Regions

register handle

```
openFile :: MonadCatchIO pr ⇒
            FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
openFile fp ioMode = block $ do
  h ← liftIO $ E.openFile fp ioMode
  register h
  return h

register :: MonadCatchIO pr ⇒ Handle ioMode → RegionT pr ()
register h = RegionT $ ask >>= liftIO ∘ flip modifyIORef (Any h:)
```

Regions
register handle

```
openFile :: MonadCatchIO pr ⇒
            FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
openFile fp ioMode = block $ do
  h ← liftIO $ E.openFile fp ioMode
  register h
  return h

register :: MonadCatchIO pr ⇒ Handle ioMode → RegionT pr ()
register h = RegionT $ ask >>= liftIO ∘ flip modifyIORef (Any h:)
```

Regions

block asynchronous exceptions

```
openFile :: MonadCatchIO pr ⇒
            FilePath → IOMode ioMode → RegionT pr (E.Handle ioMode)
openFile fp ioMode = block $ do
  h ← liftIO $ E.openFile fp ioMode
  register h
  return h

register :: MonadCatchIO pr ⇒ Handle ioMode → RegionT pr ()
register h = RegionT $ ask >>= liftIO ∘ flip modifyIORef (Any h:)
```

Regions
Running regions

```
runRegionT :: MonadCatchIO pr ⇒ RegionT pr a → pr a
runRegionT r = bracket (liftIO $ newIORef [])
                       (liftIO ∘ after)
                       (runReaderT $ unRegionT r)
  where
    after ioref = do hs ← readIORef ioref
                     forM_ hs $ λ(Any h) → hClose h
```

Regions

Create initial empty list of handles

```
runRegionT :: MonadCatchIO pr ⇒ RegionT pr a → pr a
runRegionT r = bracket (liftIO $ newIORef [])
                       (liftIO ∘ after)
                       (runReaderT $ unRegionT r)
  where
    after ioref = do hs ← readIORef ioref
                     forM_ hs $ λ(Any h) → hClose h
```

Regions

Run given region

```
runRegionT :: MonadCatchIO pr ⇒ RegionT pr a → pr a
runRegionT r = bracket (liftIO $ newIORef [])
                       (liftIO ∘ after)
                       (runReaderT $ unRegionT r)
  where
    after ioref = do hs ← readIORef ioref
                     forM_ hs $ λ(Any h) → hClose h
```

Regions
Call finalizer

```
runRegionT :: MonadCatchIO pr ⇒ RegionT pr a → pr a
runRegionT r = bracket (liftIO $ newIORef [])
                       (liftIO ∘ after)
                       (runReaderT $ unRegionT r)
  where
    after ioref = do hs ← readIORef ioref
                     forM_ hs $ λ(Any h) → hClose h
```

Regions

Close all opened handles

```
runRegionT :: MonadCatchIO pr ⇒ RegionT pr a → pr a
runRegionT r = bracket (liftIO $ newIORef [])
                       (liftIO ∘ after)
                       (runReaderT $ unRegionT r)
  where
    after ioref = do  hs ← readIORef ioref
                      forM_ hs $ λ(Any h) → hClose h
```

Regions
Operations

```
hGetLine :: (ReadModes ioMode, MonadIO r) ⇒
            Handle ioMode → r String
hGetLine h = liftIO $ E.hGetLine h

hIsEOF :: (ReadModes ioMode, MonadIO r) ⇒
          Handle ioMode → r Bool
hIsEOF h = liftIO $ E.hIsEOF h

hPutStrLn :: (WriteModes ioMode, MonadIO r) ⇒
             Handle ioMode → String → r ()
hPutStrLn h s = liftIO $ E.hPutStrLn h s
```

Regions

Operations - Just lift them

```
hGetLine :: (ReadModes ioMode, MonadIO r) ⇒
            Handle ioMode → r String
hGetLine h = liftIO $ E.hGetLine h

hIsEOF :: (ReadModes ioMode, MonadIO r) ⇒
          Handle ioMode → r Bool
hIsEOF h = liftIO $ E.hIsEOF h

hPutStrLn :: (WriteModes ioMode, MonadIO r) ⇒
             Handle ioMode → String → r ()
hPutStrLn h s = liftIO $ E.hPutStrLn h s
```

Regions

```
main :: IO ()
main = runRegionT test

test :: MonadCatchIO pr ⇒ RegionT pr ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← runRegionT $ test_internal hIn
  till (hIsEOF hIn) $
    hGetLine hIn ≫= hPutStrLn hOut

test_internal :: (MonadCatchIO pr, ReadModes ioMode)
                 ⇒ Handle ioMode → RegionT (RegionT pr) (Handle WriteMode)
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← lift $ openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn ≫= hPutStrLn hOut
  return hOut
```

Regions

```
main :: IO ()
main = runRegionT test

test :: MonadCatchIO pr ⇒ RegionT pr ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← runRegionT $ test_internal hIn
  till (hIsEOF hIn) $
    hGetLine hIn ≫= hPutStrLn hOut

test_internal :: (MonadCatchIO pr, ReadModes ioMode)
              ⇒ Handle ioMode → RegionT (RegionT pr) (Handle WriteMode)
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← lift $ openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn ≫= hPutStrLn hOut
  return hOut
```

Regions

```
main :: IO ()
main = runRegionT test

test :: MonadCatchIO pr ⇒ RegionT pr ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← runRegionT $ test_internal hIn
  till (hIsEOF hIn) $
    hGetLine hIn ≫= hPutStrLn hOut

test_internal :: (MonadCatchIO pr, ReadModes ioMode)
                 ⇒ Handle ioMode → RegionT (RegionT pr) (Handle WriteMode)
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← lift $ openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn ≫= hPutStrLn hOut
  return hOut
```

Regions

```
main :: IO ()
main = runRegionT test

test :: MonadCatchIO pr ⇒ RegionT pr ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← runRegionT $ test_internal hIn
  till (hIsEOF hIn) $
    hGetLine hIn ≫= hPutStrLn hOut

test_internal :: (MonadCatchIO pr, ReadModes ioMode)
                 ⇒ Handle ioMode → RegionT (RegionT pr) (Handle WriteMode)
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← lift $ openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn ≫= hPutStrLn hOut
  return hOut
```

Regions

```
main :: IO ()
main = runRegionT test

test :: MonadCatchIO pr ⇒ RegionT pr ()
test = do
  hIn ← openFile "input.txt" ReadMode
  hOut ← runRegionT $ test_internal hIn
  till (hIsEOF hIn) $
    hGetLine hIn ≫= hPutStrLn hOut

test_internal :: (MonadCatchIO pr, ReadModes ioMode)
                 ⇒ Handle ioMode → RegionT (RegionT pr) (Handle WriteMode)
test_internal hIn = do
  hCfg ← openFile "config.txt" ReadMode
  fname ← hGetLine hCfg
  hOut ← lift $ openFile fname WriteMode
  till (liftM2 (∨) (hIsEOF hCfg) (hIsEOF hIn)) $ do
    hGetLine hCfg ≫= hPutStrLn hOut
    hGetLine hIn ≫= hPutStrLn hOut
  return hOut
```

Regions

Returning handles from regions

```
hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
hack = do h ← runRegionT region
          hPutStrLn h "Don't do this at home!"
  where
    region :: RegionT pr (Handle WriteMode)
    region = do h ← openFile "output.txt" WriteMode
                return h
```

Regions

Associate handle with region

- **newtype** *RegionalHandle ioMode* $(r :: * \to *) =$ *RegionalHandle* $\{$*internalHandle* :: *Handle ioMode*$\}$

- *openFile* :: *MonadCatchIO pr* $\Rightarrow$
  FilePath $\to$ *IOMode ioMode* $\to$
  RegionT s pr (*RegionalHandle ioMode* (*RegionT s pr*))

Regions

Associate handle with region

- **newtype** $RegionalHandle\ ioMode\ (r :: * \to *) = RegionalHandle$
  $\{ internalHandle :: Handle\ ioMode \}$

- $openFile :: MonadCatchIO\ pr \Rightarrow$
  $FilePath \to IOMode\ ioMode \to$
  $RegionT\ s\ pr\ (RegionalHandle\ ioMode\ (RegionT\ s\ pr))$

Regions

Restrict handles to their region

- **newtype** *RegionT s pr a* = ...
- *runRegionT* :: *MonadCatchIO pr* ⇒ (∀ *s*. *RegionT s pr a*) → *pr a*
- ```
  hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
  hack = do h ← runRegionT region
            hPutStrLn h "Don't do this at home!"
    where
      region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
      region = do h ← openFile "output.txt" WriteMode
                  return h
  ```
- ```
  Inferred type is less polymorphic than expected.
  Quantified type variable `s' escapes
  ```

Regions

Restrict handles to their region

- **newtype** *RegionT s pr a* = ...
- *runRegionT* :: *MonadCatchIO pr* ⇒ (∀ *s*. *RegionT s pr a*) → *pr a*
- ```
  hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
  hack = do h ← runRegionT region
            hPutStrLn h "Don't do this at home!"
    where
      region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
      region = do h ← openFile "output.txt" WriteMode
                  return h
  ```
- ```
  Inferred type is less polymorphic than expected.
  Quantified type variable `s' escapes
  ```

Regions

Restrict handles to their region

- **newtype** *RegionT s pr a* = ...
- *runRegionT* :: *MonadCatchIO pr* ⇒ (∀ *s*. *RegionT s pr a*) → *pr a*
- ```
  hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
  hack = do h ← runRegionT region
            hPutStrLn h "Don't do this at home!"
    where
      region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
      region = do h ← openFile "output.txt" WriteMode
                  return h
  ```
- ```
  Inferred type is less polymorphic than expected.
  Quantified type variable `s' escapes
  ```

Regions

Restrict handles to their region

- **newtype** *RegionT s pr a* = ...
- *runRegionT* :: *MonadCatchIO pr* ⇒ (∀ *s*. *RegionT s pr a*) → *pr a*
- ```
  hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
  hack = do h ← runRegionT region
            hPutStrLn h "Don't do this at home!"
    where
      region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
      region = do h ← openFile "output.txt" WriteMode
                  return h
  ```
- `Inferred type is less polymorphic than expected.`
  `Quantified type variable 's' escapes`
  ```

Regions
Operations

```
hGetLine :: (ReadModes ioMode, MonadIO cr) ⇒
            RegionalHandle ioMode pr → cr String
hGetLine h = liftIO $ E.hGetLine (internalHandle h)

hIsEOF :: (ReadModes ioMode, MonadIO cr) ⇒
          RegionalHandle ioMode pr → cr Bool
hIsEOF h = liftIO $ E.hIsEOF (internalHandle h)

hPutStrLn :: (WriteModes ioMode, MonadIO cr) ⇒
             RegionalHandle ioMode pr → String → cr ()
hPutStrLn h s = liftIO $ E.hPutStrLn (internalHandle h) s
```

Regions
Return computation

```
hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
hack = do h ← runRegionT region
          hPutStrLn h "Don't do this at home!"
  where
    region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
    region = do h ← openFile "output.txt" WriteMode
                return h
```

```
hack2 :: ∀ pr. MonadCatchIO pr ⇒ pr ()
hack2 = do m ← runRegionT region
           m
  where
    region :: RegionT s pr (pr ())
    region = do h ← openFile "output.txt" WriteMode
                return $ hPutStrLn h "Don't do this at home!"
```

Regions
Return computation

```
hack :: ∀ pr. MonadCatchIO pr ⇒ pr ()
hack = do h ← runRegionT region
          hPutStrLn h "Don't do this at home!"
  where
    region :: RegionT s pr (RegionalHandle WriteMode (RegionT s pr))
    region = do h ← openFile "output.txt" WriteMode
                return h
```

```
hack2 :: ∀ pr. MonadCatchIO pr ⇒ pr ()
hack2 = do m ← runRegionT region
           m
  where
    region :: RegionT s pr (pr ())
    region = do h ← openFile "output.txt" WriteMode
                return $ hPutStrLn h "Don't do this at home!"
```

Regions

Operations are too general!

$$hGetLine :: (ReadModes\ ioMode, MonadIO\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow cr\ String$$

$$hIsEOF :: (ReadModes\ ioMode, MonadIO\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow cr\ Bool$$

$$hPutStrLn :: (WriteModes\ ioMode, MonadIO\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow String \rightarrow cr\ ()$$

Regions
Restrict

$$hGetLine :: (ReadModes\ ioMode, MonadIO\ cr, pr\ `ParentOf`\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow cr\ String$$

$$hIsEOF :: (ReadModes\ ioMode, MonadIO\ cr, pr\ `ParentOf`\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow cr\ Bool$$

$$hPutStrLn :: (WriteModes\ ioMode, MonadIO\ cr, pr\ `ParentOf`\ cr) \Rightarrow RegionalHandle\ ioMode\ pr \rightarrow String \rightarrow cr\ ()$$

Regions
ParentOf

- **class** $(Monad\ pr, Monad\ cr) \Rightarrow pr\ `ParentOf`\ cr$
- **instance** $Monad\ r \Rightarrow r\ `ParentOf`\ r$
- **instance** $(Monad\ cr, cr\ `TypeCast2`\ RegionT\ s\ pcr, pr\ `ParentOf`\ pcr) \Rightarrow pr\ `ParentOf`\ cr$
- $RegionT\ ps\ ppr\ `ParentOf`\ RegionT\ cs\ (RegionT\ pcs\ (RegionT\ ppcs\ \ldots\ (RegionT\ ps\ ppr)))$

Regions
ParentOf

- **class** $(Monad\ pr, Monad\ cr) \Rightarrow pr\ `ParentOf`\ cr$
- **instance** $Monad\ r \Rightarrow r\ `ParentOf`\ r$
- **instance** $(Monad\ cr$
  $, cr\ `TypeCast2`\ RegionT\ s\ pcr$
  $, pr\ `ParentOf`\ pcr$
  $)$
  $\Rightarrow pr\ `ParentOf`\ cr$
- $RegionT\ ps\ ppr\ `ParentOf`\ RegionT\ cs$
  $(RegionT\ pcs$
  $(RegionT\ ppcs$
  $\ldots$
  $(RegionT\ ps\ ppr)))$

Regions
ParentOf

- **class** $(Monad\ pr, Monad\ cr) \Rightarrow pr\ `ParentOf`\ cr$
- **instance** $Monad\ r \Rightarrow r\ `ParentOf`\ r$
- **instance** $(Monad\ cr$
  $, cr\ `TypeCast2`\ RegionT\ s\ pcr$
  $, pr\ `ParentOf`\ pcr$
  $)$
  $\Rightarrow pr\ `ParentOf`\ cr$
- $RegionT\ ps\ ppr\ `ParentOf`\ RegionT\ cs$
  $(RegionT\ pcs$
  $(RegionT\ ppcs$
  $\ldots$
  $(RegionT\ ps\ ppr)))$

Regions
ParentOf

- **class** $(Monad\ pr, Monad\ cr) \Rightarrow pr\ `ParentOf`\ cr$
- **instance** $Monad\ r \Rightarrow r\ `ParentOf`\ r$
- **instance** $(Monad\ cr$
  $, cr\ `TypeCast2`\ RegionT\ s\ pcr$
  $, pr\ `ParentOf`\ pcr$
  $)$
  $\Rightarrow pr\ `ParentOf`\ cr$
- $RegionT\ ps\ ppr\ `ParentOf`\ RegionT\ cs$
  $(RegionT\ pcs$
  $(RegionT\ ppcs$
  $\ldots$
  $(RegionT\ ps\ ppr)))$

Regions
Type casting

```
instance (Monad cr
         , cr `TypeCast2` RegionT s pcr
         , pr `ParentOf` pcr
         )
          ⇒ pr `ParentOf` cr

class TypeCast2    (a :: ∗ → ∗) (b :: ∗ → ∗) |  a → b
                                             ,  b → a
class TypeCast2′ t (a :: ∗ → ∗) (b :: ∗ → ∗) | t a → b
                                             , t b → a
class TypeCast2″ t (a :: ∗ → ∗) (b :: ∗ → ∗) | t a → b
                                             , t b → a
instance TypeCast2′ () a b ⇒ TypeCast2    a b
instance TypeCast2″ t  a b ⇒ TypeCast2′ t a b
instance TypeCast2″ () a a
```

Regions

hack2 now fails

- ```
  hack2 :: ∀ sp pr. MonadCatchIO pr ⇒ RegionT sp pr ()
  hack2 = do m ← runRegionT region
             m
    where
      region :: ∀ s. RegionT s (RegionT sp pr) (RegionT sp pr ())
      region = do h ← openFile "output.txt" WriteMode
                  return $ hPutStrLn h "Don't do this at home!"
  ```

- ```
  Inferred type is less polymorphic than expected
  Quantified type variable `s' is mentioned in the
  environment:...
  ```

- *RegionT s (RegionT sp pr) `ParentOf` (RegionT sp pr)*

Regions

hack2 now fails

- ```
  hack2 :: ∀ sp pr. MonadCatchIO pr ⇒ RegionT sp pr ()
  hack2 = do m ← runRegionT region
             m
    where
      region :: ∀ s. RegionT s (RegionT sp pr) (RegionT sp pr ())
      region = do h ← openFile "output.txt" WriteMode
                  return $ hPutStrLn h "Don't do this at home!"
  ```

- ```
  Inferred type is less polymorphic than expected
  Quantified type variable `s' is mentioned in the
  environment:...
  ```

- *RegionT s (RegionT sp pr) `ParentOf` (RegionT sp pr)*

Regions

hack2 now fails

- ```
  hack2 :: ∀ sp pr. MonadCatchIO pr ⇒ RegionT sp pr ()
  hack2 = do m ← runRegionT region
             m
    where
      region :: ∀ s. RegionT s (RegionT sp pr) (RegionT sp pr ())
      region = do h ← openFile "output.txt" WriteMode
                  return $ hPutStrLn h "Don't do this at home!"
  ```

- `Inferred type is less polymorphic than expected`
  `Quantified type variable 's' is mentioned in the environment:...`
- *RegionT s (RegionT sp pr) `ParentOf` (RegionT sp pr)*
  ```

Regions

Leaking handles via *IOErrors*

- *throwsException* = *runRegionT* \$ **do**
  h ← *openFile* `"foo.txt"` *ReadMode*
  hSeek h AbsoluteSeek (−1)

- *leak* = *throwsException* ‘*catch*‘ λ(*e* :: *IOError*) →
  case *ioeGetHandle e* **of**
  Nothing → *return* ()
  Just h → *System.IO.hGetLine h*

- *ioeGetHandle* :: *IOError* → *Maybe Handle*

Regions

Leaking handles via *IOErrors*

- *throwsException* = *runRegionT* \$ **do**
  h ← *openFile* `"foo.txt"` *ReadMode*
  hSeek h AbsoluteSeek (−1)

- *leak* = *throwsException* ‘*catch*‘ λ(*e* :: *IOError*) →
  case *ioeGetHandle e* **of**
  Nothing → *return* ()
  Just h → *System.IO.hGetLine h*

- *ioeGetHandle* :: *IOError* → *Maybe Handle*

Regions

Leaking handles via *IOErrors*

- *throwsException* = *runRegionT* \$ **do**
  h ← *openFile* `"foo.txt"` *ReadMode*
  hSeek h AbsoluteSeek (−1)

- *leak* = *throwsException* ‘*catch*‘ λ(*e* :: *IOError*) →
  case *ioeGetHandle e* **of**
  Nothing → *return* ()
  Just h → *System.IO.hGetLine h*

- *ioeGetHandle* :: *IOError* → *Maybe Handle*

Regions

Remove handles from *IOErrors*

- $sanitize :: IO\ a \rightarrow IO\ a$
  $sanitize = modifyIOError\ \$\ \lambda e \rightarrow e\ \{ioe\_handle = Nothing\}$

- ```
  hIsEOF    h      = liftIO $ sanitize $ E.hIsEOF    (internalHandle h)
  hPutStrLn h s    = liftIO $ sanitize $ E.hPutStrLn (internalHandle h) s
  hSeek     h sm n = liftIO $ sanitize $ E.hSeek     (internalHandle h) sm n
  ```
  ```

Regions

Remove handles from *IOErrors*

- $sanitize :: IO\ a \to IO\ a$
  $sanitize = modifyIOError\ \$\ \lambda e \to e\ \{ioe\_handle = Nothing\}$

- $$\begin{aligned} hIsEOF\ \ h &= liftIO\ \$\ sanitize\ \$\ E.hIsEOF\ \ (internalHandle\ h) \\ hPutStrLn\ h\ s &= liftIO\ \$\ sanitize\ \$\ E.hPutStrLn\ (internalHandle\ h)\ s \\ hSeek\ \ h\ sm\ n &= liftIO\ \$\ sanitize\ \$\ E.hSeek\ \ (internalHandle\ h)\ sm\ n \end{aligned}$$

Regions

Extension: dynamically extending the lifetime of handles

```
test2 = runRegionT $ do
  h ← runRegionT $ test5_internal "conf2.txt"
  l ← hGetLine h

test2_internal conf_fname = do
  hc ← openFile conf_fname ReadMode
  fname1 ← hGetLine hc
  fname2 ← hGetLine hc
  h1 ← openFile fname1 ReadMode
  h2 ← openFile fname2 ReadMode
  l1 ← hGetLine h1
  l2 ← hGetLine h2
  let hOld = if l1 < l2 then h2 else h1
  return hOld
```

Regions

Extension: dynamically extending the lifetime of handles

```
test2 = runRegionT $ do
  h ← runRegionT $ test5_internal "conf2.txt"
  l ← hGetLine h

test2_internal conf_fname = do
  hc ← openFile conf_fname ReadMode
  fname1 ← hGetLine hc
  fname2 ← hGetLine hc
  h1 ← openFile fname1 ReadMode
  h2 ← openFile fname2 ReadMode
  l1 ← hGetLine h1
  l2 ← hGetLine h2
  let hOld = if l1 < l2 then h2 else h1
  return hOld
```

Regions

Extension: dynamically extending the lifetime of handles

```
test2 = runRegionT $ do
  h ← runRegionT $ test5_internal "conf2.txt"
  l ← hGetLine h

test2_internal conf_fname = do
  hc ← openFile conf_fname ReadMode
  fname1 ← hGetLine hc
  fname2 ← hGetLine hc
  h1 ← openFile fname1 ReadMode
  h2 ← openFile fname2 ReadMode
  l1 ← hGetLine h1
  l2 ← hGetLine h2
  let hOld = if l1 < l2 then h2 else h1
  dup hOld
```

Regions

Extension: dynamically extending the lifetime of handles

$$
\begin{aligned}
&\mathit{dup} :: \mathit{MonadCatchIO}\ \mathit{ppr} \Rightarrow \\
&\quad \mathit{RegionalHandle}\ \mathit{ioMode}\ (\mathit{RegionT}\ \mathit{cs})\ (\mathit{RegionT}\ \mathit{ps}\ \mathit{ppr}) \rightarrow \\
&\quad\quad \mathit{RegionT}\ \mathit{cs}\ (\mathit{RegionT}\ \mathit{ps}\ \mathit{ppr}) \\
&\quad\quad\quad (\mathit{RegionalHandle}\ \mathit{ioMode}\ (\mathit{RegionT}\ \mathit{ps}\ \mathit{ppr}))
\end{aligned}
$$

Generalization
Not just files

```
class Resource resource where
  data Handle resource :: *
  openResource :: resource → IO (Handle resource)
  closeResource :: Handle resource → IO ()

data RegionalHandle resource (r :: * → *) = RegionalHandle
  {internalHandle :: Handle resource}

open :: (Resource resource, MonadCatchIO m) ⇒
        resource → RegionT s m
                   (RegionalHandle resource (RegionT s m))
```

Generalization
Not just files

```
class Resource resource where
  data Handle resource :: *
  openResource :: resource → IO (Handle resource)
  closeResource :: Handle resource → IO ()

data RegionalHandle resource (r :: * → *) = RegionalHandle
  {internalHandle :: Handle resource}

open :: (Resource resource, MonadCatchIO m) ⇒
        resource → RegionT s m
                   (RegionalHandle resource (RegionT s m))
```

Generalization
Not just files

```
class Resource resource where
  data Handle resource :: *
  openResource :: resource → IO (Handle resource)
  closeResource :: Handle resource → IO ()

data RegionalHandle resource (r :: * → *) = RegionalHandle
  {internalHandle :: Handle resource}

open :: (Resource resource, MonadCatchIO m) ⇒
        resource → RegionT s m
                   (RegionalHandle resource (RegionT s m))
```

Generalization
Not just files

```
newtype RegionT s m a = RegionT
  { unRegionT :: ReaderT (IORef [AnyHandle]) m a }

data AnyHandle = forall resource r ∘ Resource resource ⇒
                 Any (RegionalHandle resource r)
```

Generalization
Not just files

$$\textbf{newtype}\ \mathit{RegionT}\ s\ m\ a = \mathit{RegionT}\ \{\mathit{unRegionT} :: \mathit{ReaderT}\ (\mathit{IORef}\ [\mathit{AnyHandle}])\ m\ a\}$$

$$\textbf{data}\ \mathit{AnyHandle} = \mathit{forall}\ \mathit{resource}\ r \circ \mathit{Resource}\ \mathit{resource} \Rightarrow \mathit{Any}\ (\mathit{RegionalHandle}\ \mathit{resource}\ r)$$

Generalization
Files as scarce resources

```
cabal install safer-file-handles
```

```
data File ioMode where
  File     :: Binary → FilePath → IOMode ioMode → File ioMode
  TempFile :: Binary → FilePath → Template → DefaultPermissions →
              File ReadWriteMode
```

Generalization
Files as scarce resources

```
cabal install safer-file-handles
```

```
data File ioMode where
  File     :: Binary → FilePath → IOMode ioMode → File ioMode
  TempFile :: Binary → FilePath → Template → DefaultPermissions →
                File ReadWriteMode
```

```
instance Resource (File ioMode) where
  data Handle (File ioMode) =
      FileHandle { mbFilePath :: Maybe FilePath
                 , handle :: E.Handle ioMode }
  openResource (File isBinary filePath ioMode) =
    FileHandle Nothing <$>
      (if isBinary then E.openBinaryFile else E.openFile)
      filePath ioMode
  openResource (TempFile isBinary filePath template defaultPerms) =
    uncurry (FileHandle ∘ Just) <$>
      (case (isBinary, defaultPerms) of
        (False, False) → E.openTempFile
        (True,  False) → E.openBinaryTempFile
        (False, True)  → E.openTempFileWithDefaultPermissions
        (True,  True)  → E.openBinaryTempFileWithDefaultPermissions)
        filePath template
  closeResource = sanitizeIOError ∘ E.hClose ∘ handle
```

Generalization
Memory as a scarce resource

```
cabal install regional-pointers
```

```
newtype Memory a = Memory {size :: Int}

instance Resource (Memory a) where
  newtype Handle (Memory a) = Pointer {ptr :: Ptr a}
  openResource = fmap Pointer ∘ mallocBytes ∘ size
  closeResource = free ∘ ptr

type RegionalPtr a r = RegionalHandle (Memory a) r

peek :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → cr a
poke :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → a → cr ()
```

Generalization
Memory as a scarce resource

```
cabal install regional-pointers
```

```
newtype Memory a = Memory {size :: Int}

instance Resource (Memory a) where
  newtype Handle (Memory a) = Pointer {ptr :: Ptr a}
  openResource = fmap Pointer ∘ mallocBytes ∘ size
  closeResource = free ∘ ptr

type RegionalPtr a r = RegionalHandle (Memory a) r

peek :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → cr a
poke :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → a → cr ()
```

Generalization
Memory as a scarce resource

```
cabal install regional-pointers
```

```
newtype Memory a = Memory {size :: Int}

instance Resource (Memory a) where
  newtype Handle (Memory a) = Pointer {ptr :: Ptr a}
  openResource = fmap Pointer ∘ mallocBytes ∘ size
  closeResource = free ∘ ptr

type RegionalPtr a r = RegionalHandle (Memory a) r

peek :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → cr a
poke :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → a → cr ()
```

Generalization
Memory as a scarce resource

```
cabal install regional-pointers
```

```
newtype Memory a = Memory {size :: Int}

instance Resource (Memory a) where
  newtype Handle (Memory a) = Pointer {ptr :: Ptr a}
  openResource = fmap Pointer ∘ mallocBytes ∘ size
  closeResource = free ∘ ptr

type RegionalPtr a r = RegionalHandle (Memory a) r

peek :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → cr a
poke :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → a → cr ()
```

Generalization
Memory as a scarce resource

```
cabal install regional-pointers
```

```
newtype Memory a = Memory {size :: Int}

instance Resource (Memory a) where
  newtype Handle (Memory a) = Pointer {ptr :: Ptr a}
  openResource = fmap Pointer ∘ mallocBytes ∘ size
  closeResource = free ∘ ptr

type RegionalPtr a r = RegionalHandle (Memory a) r

peek :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → cr a
poke :: (pr `ParentOf` cr, Storable a, MonadIO cr) ⇒
        RegionalPtr a pr → a → cr ()
```

Generalization
USB devices as scarce resources

```
cabal install usb
```

```
data Device = Device { getDevFrgnPtr :: ForeignPtr C'libusb_device
                     , ...
                     }
```

```
cabal install usb-safe
```

```
instance Resource USB.Device where
  data Handle USB.Device = DeviceHandle
    { internalDevHndl :: USB.DeviceHandle, ... }
  openResource = fmap DeviceHandle ∘ USB.openDevice
  closeResource = USB.closeDevice ∘ internalDevHndl

type RegionalDeviceHandle r = RegionalHandle USB.Device r
```

Generalization
USB devices as scarce resources

```
cabal install usb
```

```
data Device = Device { getDevFrgnPtr :: ForeignPtr C'libusb_device
                     , ...
                     }
```

```
cabal install usb-safe
```

```
instance Resource USB.Device where
  data Handle USB.Device = DeviceHandle
    { internalDevHndl :: USB.DeviceHandle, ... }
  openResource = fmap DeviceHandle ∘ USB.openDevice
  closeResource = USB.closeDevice ∘ internalDevHndl

type RegionalDeviceHandle r = RegionalHandle USB.Device r
```

Generalization
USB devices as scarce resources

```
cabal install usb
```

```
data Device = Device { getDevFrgnPtr :: ForeignPtr C'libusb_device
                     , ...
                     }
```

```
cabal install usb-safe
```

```
instance Resource USB.Device where
  data Handle USB.Device = DeviceHandle
    { internalDevHndl :: USB.DeviceHandle, ... }
  openResource = fmap DeviceHandle ∘ USB.openDevice
  closeResource = USB.closeDevice ∘ internalDevHndl

type RegionalDeviceHandle r = RegionalHandle USB.Device r
```

Generalization
USB devices as scarce resources

```
cabal install usb
```

```
data Device = Device {getDevFrgnPtr :: ForeignPtr C'libusb_device
                     , ...
                     }
```

```
cabal install usb-safe
```

```
instance Resource USB.Device where
  data Handle USB.Device = DeviceHandle
    {internalDevHndl :: USB.DeviceHandle, ...}
  openResource = fmap DeviceHandle ∘ USB.openDevice
  closeResource = USB.closeDevice ∘ internalDevHndl

type RegionalDeviceHandle r = RegionalHandle USB.Device r
```

Generalization
USB devices as scarce resources

```
cabal install usb
```

```
data Device = Device {getDevFrgnPtr :: ForeignPtr C'libusb_device
                     , ...
                     }
```

```
cabal install usb-safe
```

```
instance Resource USB.Device where
  data Handle USB.Device = DeviceHandle
    {internalDevHndl :: USB.DeviceHandle, ...}
  openResource = fmap DeviceHandle ∘ USB.openDevice
  closeResource = USB.closeDevice ∘ internalDevHndl

type RegionalDeviceHandle r = RegionalHandle USB.Device r
```

Questions

Questions ?