Lightweight monadic regions

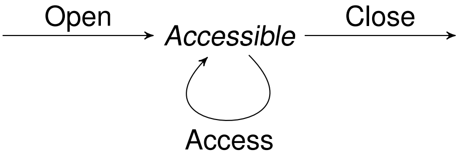
Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers ⇄ Aarhus)

Haskell Symposium
25 September 2008

What?

Goal: Resource management

- No access after close (down with run-time checking)
- Timely disposal (especially for scarce resources)
- Error handling



Motivating example: File handles

input
2
4
6
7
8
9

config
log
1
3
5

1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

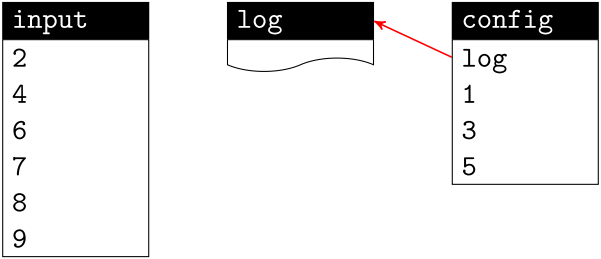
Motivating example: File handles

input
2
4
6
7
8
9

config
log
1
3
5

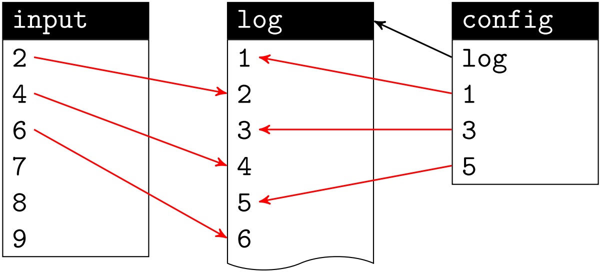
1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

Motivating example: File handles



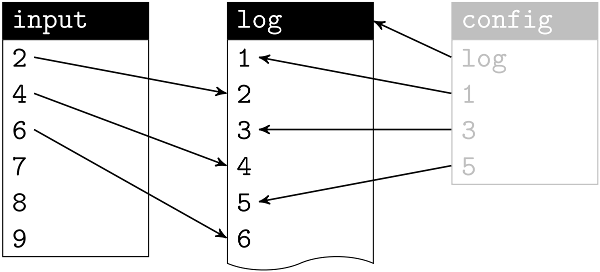
1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

Motivating example: File handles



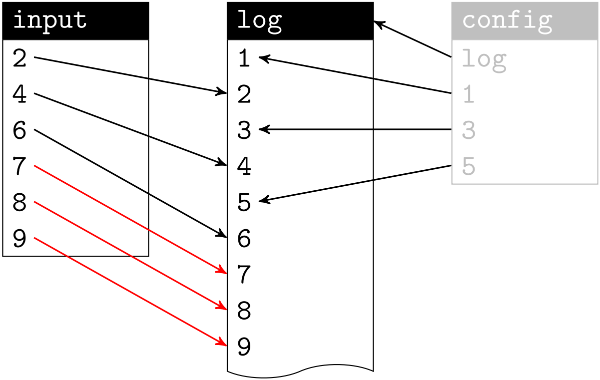
1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

Motivating example: File handles



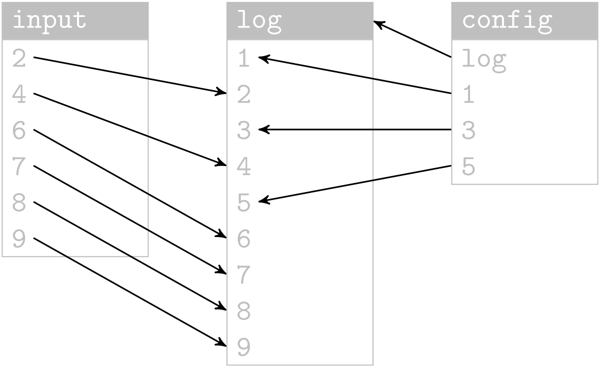
1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

Motivating example: File handles



1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

Motivating example: File handles



1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

How?

Goal: Resource management

- No access after close
- Timely disposal
- Error handling

Solution: Nested regions

- Phantom types a la ST
- Monad transformer
- *Implicit* region subtyping

Impose a syntactic discipline on *native* capabilities.

Further applications

- Database connections
- OpenGL contexts
- Automatic differentiation

Another approach

Safe *manual* resource management, using a *parameterized monad*

How?

Goal: Resource management

- No access after close
- Timely disposal
- Error handling

Solution: Nested regions

- Phantom types a la ST
- Monad transformer
- *Implicit* region subtyping

Impose a syntactic discipline on *native* capabilities.

Further applications

- Database connections
- OpenGL contexts
- Automatic differentiation

Another approach

Safe *manual* resource management, using a *parameterized monad*

Outline

- **Safe file handles in a single region**
    - Interface
    - Implementation
- Nested regions using explicit witness terms
    - Interface
    - Implementation
- Nested regions as monad transformers
    - Interface
    - Implementation
- Manual resource management

Leaking handles is dangerous

Encapsulate a file handle for safety?

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode = bracket (openFile name mode) hClose
```

> `withFile name mode act` *opens a file using* `openFile` *and passes the resulting handle to the computation* `act`. *The handle will be closed on exit from* `withFile`, *whether by normal termination or by raising an exception.*

But the type `a` could be `Handle`!

```
withFile "FilePath" ReadMode return >>= hGetLine
```

Prevent leaking statically, by analogy to state threads.
Then, no need to check dynamically for reading from a closed file.

Leaking handles is dangerous

Encapsulate a file handle for safety?

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode = bracket (openFile name mode) hClose
```

> `withFile name mode act` *opens a file using* `openFile`
> *and passes the resulting handle to the computation* `act`*.*
> *The handle will be closed on exit from* `withFile`*,*
> *whether by normal termination or by raising an exception.*

But the type `a` could be `Handle`!

```
withFile "FilePath" ReadMode return >>= hGetLine
```

Prevent leaking statically, by analogy to state threads.
Then, no need to check dynamically for reading from a closed file.

State threads

```
ST          :: * -> * -> *
STRef       :: * -> * -> *
instance Monad (ST s)
```

Allocate

```
newSTRef    :: a -> ST s (STRef s a)
```

Access

```
readSTRef   :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Encapsulate

```
runST       :: (∀s. ST s a) -> a
```

Every cell is implicitly deallocated exactly once, after all access.

State threads

```
ST          :: * -> * -> *
STRef       :: * -> * -> *
instance Monad (ST s)
```

Allocate

```
newSTRef    :: a -> ST s (STRef s a)
```

Access

```
readSTRef   :: STRef s a -> ST s a
writeSTRef  :: STRef s a -> a -> ST s ()
```

Encapsulate

```
runST       :: (∀s. ST s a) -> a
```

Every cell is implicitly deallocated exactly once, after all access.

State threads

```
ST           :: * -> * -> *
STRef        :: * -> * -> *
instance Monad (ST s)
```

Allocate

```
newSTRef    :: a -> ST s (STRef s a)
```

Access

```
readSTRef   :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Encapsulate

```
runST        :: (∀s. ST s a) -> a
```

Every cell is implicitly deallocated exactly once, after all access.

Handle threads

```
SIO          :: * -> * -> *
SHandle      :: (* -> *) -> *
instance Monad (SIO s)
```

Allocate

```
newSHandle :: FilePath -> IOMode -> SIO s (SHandle (SIO s))
```

Access

```
shGetLine  :: SHandle (SIO s) -> SIO s String
shPutStrLn :: SHandle (SIO s) -> String -> SIO s ()
shIsEOF    :: SHandle (SIO s) -> SIO s Bool
```

Encapsulate

```
runSIO     :: (∀s. SIO s a) -> IO a
```

Every handle is implicitly closed exactly once, after all access.

Handle threads

```
SIO         :: * -> * -> *
SHandle     :: (* -> *) -> *
instance Monad (SIO s)
```

Allocate

```
newSHandle :: FilePath -> IOMode -> SIO s (SHandle (SIO s))
```

Access

```
shGetLine  :: SHandle (SIO s) -> SIO s String
shPutStrLn :: SHandle (SIO s) -> String -> SIO s ()
shIsEOF    :: SHandle (SIO s) -> SIO s Bool
```

Encapsulate

```
runSIO     :: (∀s. SIO s a) -> IO a
```

Every handle is implicitly closed exactly once, after all access.

Handle threads

```
SIO          :: * -> * -> *
SHandle      :: (* -> *) -> *
instance Monad (SIO s)
```

Allocate

```
newSHandle :: FilePath -> IOMode -> SIO s (SHandle (SIO s))
```

Access

```
shGetLine  :: SHandle (SIO s) -> SIO s String
shPutStrLn :: SHandle (SIO s) -> String -> SIO s ()
shIsEOF    :: SHandle (SIO s) -> SIO s Bool
```

Encapsulate

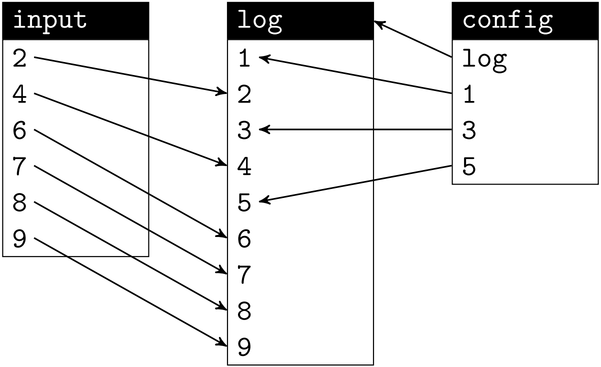
```
runSIO     :: (∀s. SIO s a) -> IO a
```

Every handle is implicitly closed exactly once, after all access.

Usage

Simple monadic programming.

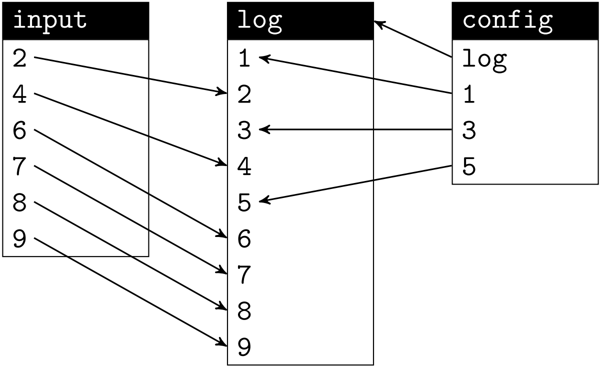
```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))
```



Usage

Simple monadic programming.

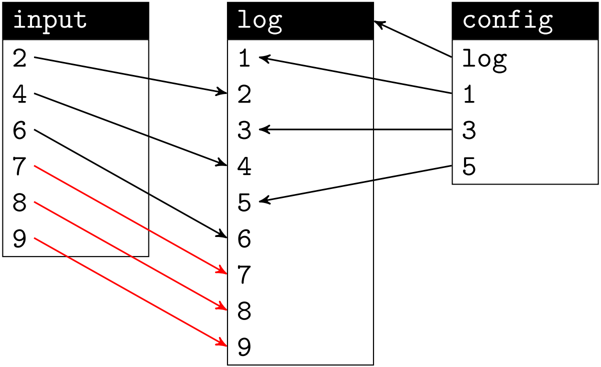
```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))
```



Usage

Simple monadic programming.

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))
```



Usage

Simple monadic programming.

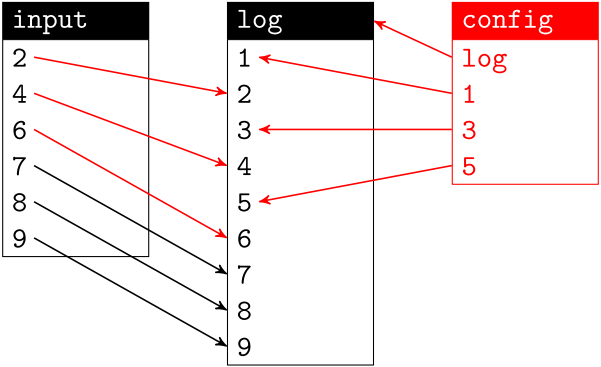
```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))

till condition iteration = loop where
  loop = do b <- condition
            if b then return ()
                 else iteration >> loop
```

Usage

Simple monadic programming.

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))
```



Usage

Simple monadic programming.

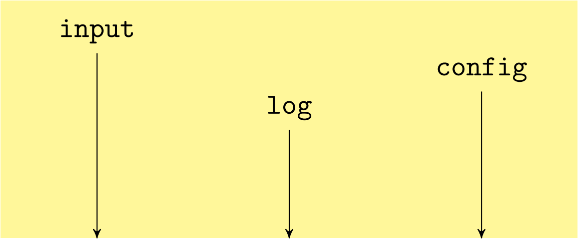
```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))

test3_internal h1 = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- newSHandle fname WriteMode
  shPutStrLn h3 fname
  till (liftM2 (||) (shIsEOF h2) (shIsEOF h1))
       (shGetLine h2 >>= shPutStrLn h3 >>
        shGetLine h1 >>= shPutStrLn h3)
  return h3
```

Usage

Simple monadic programming.

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))
```



Error handling

Every operation can throw an exception, especially `newSHandle`.

```
shThrow :: Exception -> SIO s a
shCatch :: SIO s a -> (Exception -> SIO s a) -> SIO s a
```

Sanitize `Exception` to remove any unsafe (low-level) `Handle`.
Re-throw `Exception` if uncaught in `runSIO`.

Implementation

Apply the reader monad transformer to `IO`, for `runSIO` and `newSHandle` to keep a list of open handles in an `IORef` cell.

```
newtype SHandle     = SHandle Handle
newtype IORT s m a = IORT (IORef [Handle] -> m a)
type    SIO s       = IORT s IO
```

Implementation

Apply the reader monad transformer to `IO`, for `runSIO` and `newSHandle` to keep a list of open handles in an `IORef` cell.

```
newtype SHandle     = SHandle Handle
newtype IORT s m a = IORT (IORef [Handle] -> m a)
type    SIO s       = IORT s IO
```

Run-time overhead when opening files, not accessing them.

Implementation

Apply the reader monad transformer to `IO`, for `runSIO` and `newSHandle` to keep a list of open handles in an `IORef` cell.

```
newtype SHandle     = SHandle Handle
newtype IORT s m a = IORT (IORef [Handle] -> m a)
type    SIO s       = IORT s IO
```

Plumbing: a monad class for IO and exception handling

```
class Monad m => RMonadIO m where
  brace :: m a -> (a -> m b) -> (a -> m c) -> m c
  snag  :: m a -> (Exception -> m a) -> m a
  lIO   :: IO a -> m a
```

```
instance RMonadIO IO where ... -- Sanitize exceptions
instance RMonadIO m => RMonadIO (IORT s m) where ...
```

Unexported names constitute the security kernel

Implementation

Apply the reader monad transformer to `IO`, for `runSIO` and `newSHandle` to keep a list of open handles in an `IORef` cell.

```
newtype SHandle     = SHandle Handle
newtype IORT s m a = IORT (IORef [Handle] -> m a)
type    SIO s       = IORT s IO
```

Plumbing: a monad class for IO and exception handling

```
class Monad m => RMonadIO m where
  brace :: m a -> (a -> m b) -> (a -> m c) -> m c
  snag  :: m a -> (Exception -> m a) -> m a
  lIO   :: IO a -> m a
```

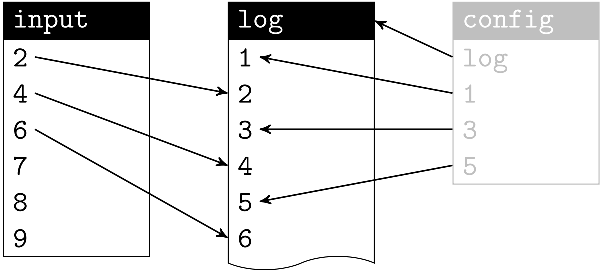
```
instance RMonadIO IO where ... -- Sanitize exceptions
instance RMonadIO m => RMonadIO (IORT s m) where ...
```

Unexported names constitute the security kernel

Outline

- Safe file handles in a single region
  - Interface
  - Implementation
- ▶ **Nested regions using explicit witness terms**
  - Interface
  - Implementation
- Nested regions as monad transformers
  - Interface
  - Implementation
- Manual resource management

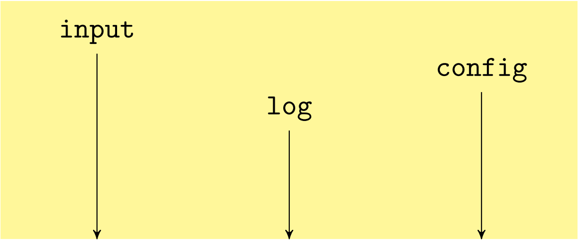
Motivating example: File handles



1. Open `input` and `config` for reading.
2. From `config`, read the file name `log` to open for writing.
3. Zip `input` and `config` into `log`.
4. Close `config`.
5. Copy the rest of `input` to `log`.

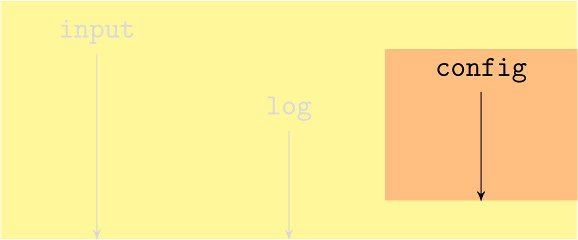
Nested regions

- To close `config` early, open it in a child region.
- To use `input` and `log` while `config` is open, let a child computation use parent regions (Launchbury and Sabry).
- To make a child computation polymorphic in its parent regions, pass witnesses for region subtyping (Fluet and Morrisett).



Nested regions

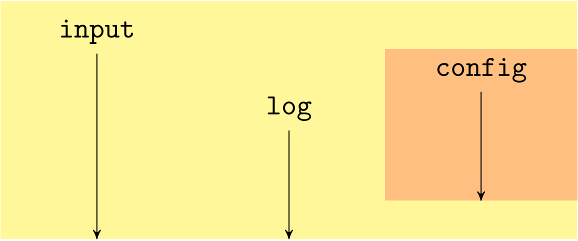
- To close `config` early, open it in a child region.
- To use `input` and `log` while `config` is open, let a child computation use parent regions (Launchbury and Sabry).
- To make a child computation polymorphic in its parent regions, pass witnesses for region subtyping (Fluet and Morrisett).




```
newRgn :: (∀s. SIO s a) -> SIO r a
newRgn m = lIO (runSIO m)
```

Nested regions

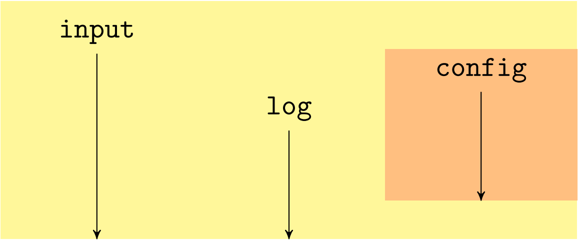
- To close `config` early, open it in a child region.
- To use `input` and `log` while `config` is open, let a child computation use parent regions (Launchbury and Sabry).
- To make a child computation polymorphic in its parent regions, pass witnesses for region subtyping (Fluet and Morrisett).




```
newRgn :: (∀s. SIO (r,s) a) -> SIO r a
importSHandle :: SHandle (SIO r) -> SHandle (SIO (r,s))
```

Nested regions

- To close `config` early, open it in a child region.
- To use `input` and `log` while `config` is open, let a child computation use parent regions (Launchbury and Sabry).
- To make a child computation polymorphic in its parent regions, pass witnesses for region subtyping (Fluet and Morrisett).




```
newRgn :: (∀s. SubRegion r s -> SIO s a) -> SIO r a
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Usage

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- newRgn (test3_internal h1)
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))

test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Usage

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- newRgn (test3_internal h1)
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))

test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Usage

```
test3 = runSIO (do
  h1 <- newSHandle "input" ReadMode
  h3 <- newRgn (test3_internal h1)
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3))

test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Usage

Haskell infers region polymorphism for `test3_internal`:

```
test3_internal :: SHandle (SIO t) -> SubRegion t s ->
                  SIO s (SHandle (SIO t))
```

Still, explicit witnesses are annoying and error-prone to juggle.

```
test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Implementation

Unchanged from before:

```
newtype SHandle     = SHandle Handle
newtype IORT s m a = IORT (IORef [Handle] -> m a)
type    SIO s       = IORT s IO
```

The only new function:

```
newRgn :: (∀s. SubRegion r s -> SIO s a) -> SIO r a
newRgn body = IORT (\open ->
              let witness (IORT m) = lIO (m open)
              in runSIO (body witness))

type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Outline

Implicit region subtyping

```
test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Implicit region subtyping

```
test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO . shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Nested regions as monad transformers

A witness for region subtyping is a monad morphism!

```
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Create a child region by applying a monad transformer.
Get a *family* of `SIO` monads:

```
class Monad m => RMonadIO m
instance RMonadIO IO
instance RMonadIO (IORT r IO)
instance RMonadIO (IORT s (IORT r IO))
...
```

Nested regions as monad transformers

A witness for region subtyping is a monad morphism!

```
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Create a child region by applying a monad transformer.
Get a *family* of `SIO` monads:

```
class Monad m => RMonadIO m
instance RMonadIO IO
instance RMonadIO m => RMonadIO (IORT s m)
```

Nested regions as monad transformers

A witness for region subtyping is a monad morphism!

```
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Create a child region by applying a monad transformer.
Get a *family* of `SIO` monads:

```
class Monad m => RMonadIO m

liftSIO :: Monad m => IORT r m a -> IORT s (IORT r m) a
```

Nested regions as monad transformers

A witness for region subtyping is a monad morphism!

```
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Create a child region by applying a monad transformer.
Get a *family* of `SIO` monads:

```
class Monad m => RMonadIO m

liftSIO :: Monad m => IORT r m a -> IORT s (IORT r m) a
```

Express region ancestry by a type predicate:

```
class (RMonadIO m, RMonadIO n) => MonadRaise m n
instance RMonadIO m => MonadRaise m m
instance RMonadIO m => MonadRaise m (IORT s1 m)
instance RMonadIO m => MonadRaise m (IORT s2 (IORT s1 m))
...
```

Nested regions as monad transformers

A witness for region subtyping is a monad morphism!

```
type SubRegion r s = ∀a. SIO r a -> SIO s a
```

Create a child region by applying a monad transformer.
Get a *family* of `SIO` monads:

```
class Monad m => RMonadIO m

liftSIO :: Monad m => IORT r m a -> IORT s (IORT r m) a
```

Express region ancestry by a type predicate:

```
class (RMonadIO m, RMonadIO n) => MonadRaise m n

shGetLine  :: MonadRaise m n => SHandle m -> n String
shPutStrLn :: MonadRaise m n => SHandle m -> String -> n ()
shIsEOF    :: MonadRaise m n => SHandle m -> n Bool
```

Region polymorphism

```
copy h1 h2 = do line <- shGetLine h1
                shPutStrLn h2 line
```

Express region ancestry by a type predicate:

```
class (RMonadIO m, RMonadIO n) => MonadRaise m n
```

```
shGetLine  :: MonadRaise m n => SHandle m -> n String
shPutStrLn :: MonadRaise m n => SHandle m -> String -> n ()
shIsEOF    :: MonadRaise m n => SHandle m -> n Bool
```

Region polymorphism

```
copy :: (MonadRaise m1 n, MonadRaise m2 n)
     => SHandle m1 -> SHandle m2 -> n ()

copy h1 h2 = do line <- shGetLine h1
                shPutStrLn h2 line
```

Express region ancestry by a type predicate:

```
class (RMonadIO m, RMonadIO n) => MonadRaise m n

shGetLine  :: MonadRaise m n => SHandle m -> n String
shPutStrLn :: MonadRaise m n => SHandle m -> String -> n ()
shIsEOF    :: MonadRaise m n => SHandle m -> n Bool
```

Implicit region subtyping

```
test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Implicit region subtyping

```
test3_internal :: MonadRaise m (IORT s (IORT r n)) =>
  SHandle m -> IORT s (IORT r n) (SHandle (IORT r n))

test3_internal h1 liftSIO = do
  h2 <- newSHandle "config" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  liftSIO (shPutStrLn h3 fname)
  till (liftM2 (||) (shIsEOF h2)
                    (liftSIO (shIsEOF h1)))
       (shGetLine h2 >>= liftSIO shPutStrLn h3 >>
        liftSIO (shGetLine h1 >>= shPutStrLn h3))
  return h3
```

Use `liftSIO` to *create* a handle in an ancestor region.

Implementation

Only changes:

1. `newRgn` is just `runSIO` with a more general type.
2. `liftSIO = IORT . const`

Express region ancestry by a type predicate

```
class (RMonadIO m, RMonadIO n) => MonadRaise m n

instance RMonadIO m => MonadRaise m m
instance RMonadIO m => MonadRaise m (IORT s1 m)
instance RMonadIO m => MonadRaise m (IORT s2 (IORT s1 m))
...
```

Express region ancestry by a type predicate

```
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE UndecidableInstances   #-}
{-# LANGUAGE OverlappingInstances   #-}

class (RMonadIO m, RMonadIO n) => MonadRaise m n

instance RMonadIO m => MonadRaise m m
instance (RMonadIO n, TypeCast2 n (IORT s n'),
          MonadRaise m n')
    => MonadRaise m n

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
```

Recap

- Encapsulate resource access in regions
- Nest computation by monad transformers (Filinski)
- Practical tradeoff between implicit subtyping and inference

The struggle for timely disposal continues:
When opening a file, we may not yet know when to close it.

```
shDup :: RMonadIO m =>
         SHandle (IORT s (IORT r m)) ->
         IORT s (IORT r m) (SHandle (IORT r m))
```

Recap

- Encapsulate resource access in regions
- Nest computation by monad transformers (Filinski)
- Practical tradeoff between implicit subtyping and inference

The struggle for timely disposal continues:
When opening a file, we may not yet know when to close it.

```
shDup :: RMonadIO m =>
         SHandle (IORT s (IORT r m)) ->
         IORT s (IORT r m) (SHandle (IORT r m))
```

Outline

Type-state

Explicit close eases timely disposal, but how to ensure safety?

Track open files exactly and statically in a *parameterized monad*.

```
class Monadish m where
  gret  :: a -> m p p a
  gbind :: m p q a -> (a -> m q r b) -> m p r b
```

Type-state

Explicit close eases timely disposal, but how to ensure safety?

Track open files exactly and statically in a *parameterized monad*.

```
class Monadish m where
  gret  :: a -> m p p a
  gbind :: m p q a -> (a -> m q r b) -> m p r b
```

Type-state

Explicit close eases timely disposal, but how to ensure safety?

Track open files exactly and statically in a *parameterized monad*.

```
class Monadish m where
  gret  :: a -> m p p a
  gbind :: m p q a -> (a -> m q r b) -> m p r b

test3_internal h1 =
  tshOpen "config" ReadMode >== \h2 ->
  tshGetLine h2 >== \fname ->
  tshOpen fname WriteMode >== \h3 ->
  tshPutStrLn h3 fname >>
  till (liftM2 (||) (tshIsEOF h2) (tshIsEOF h1))
       (tshGetLine h2 >>= tshPutStrLn h3 >>
        tshGetLine h1 >>= tshPutStrLn h3) >>
  tshClose h2 +>>
  gret h3
```

Type-state

Explicit close eases timely disposal, but how to ensure safety?

Track open files exactly and statically in a *parameterized monad*.

```
class Monadish m where
  gret  :: a -> m p p a
  gbind :: m p q a -> (a -> m q r b) -> m p r b

test3_internal h1 =
  tshOpen "config" ReadMode >== \h2 ->
  tshGetLine h2 >== \fname ->
  tshOpen fname WriteMode >== \h3 ->
  tshPutStrLn h3 fname >>
  till (liftM2 (||) (tshIsEOF h2) (tshIsEOF h1))
       (tshGetLine h2 >>= tshPutStrLn h3 >>
        tshGetLine h1 >>= tshPutStrLn h3) >>
  tshClose h2 +>>
  gret h3
```

Type-state

```
test3_internal
 :: TSHandle s Z -> TSIO s
                         (S Z, C Z N)
                         (S (S (S Z)), C (S (S Z)) (C Z N))
                         (TSHandle s (S (S Z)))


test3_internal h1 =
  tshOpen "config" ReadMode >== \h2 ->
  tshGetLine h2 >== \fname ->
  tshOpen fname WriteMode >== \h3 ->
  tshPutStrLn h3 fname >>
  till (liftM2 (||) (tshIsEOF h2) (tshIsEOF h1))
       (tshGetLine h2 >>= tshPutStrLn h3 >>
        tshGetLine h1 >>= tshPutStrLn h3) >>
  tshClose h2 +>>
  gret h3
```

Type-state

```
test3_internal
 :: TSHandle s 0 -> TSIO s
                         (1, [0])
                         (3, [2,0])
                         (TSHandle s 2)


test3_internal h1 =
  tshOpen "config" ReadMode >== \h2 ->
  tshGetLine h2 >== \fname ->
  tshOpen fname WriteMode >== \h3 ->
  tshPutStrLn h3 fname >>
  till (liftM2 (||) (tshIsEOF h2) (tshIsEOF h1))
       (tshGetLine h2 >>= tshPutStrLn h3 >>
        tshGetLine h1 >>= tshPutStrLn h3) >>
  tshClose h2 +>>
  gret h3
```

Type-state

```
test3_internal
 :: (Apply (Closure RemL bf2) (t, C (S t) (C t u)) r3,
     EQN t (S t) bf2,
     Apply (Closure RemL bf1) (S t, C (S t) (C t u)) r2,
     EQN t t bf1,
     Apply (Closure RemL bf) (t2, C (S t) (C t u)) r1,
     EQN t2 (S t) bf,
     Apply (Closure RemL bf1) (t, C t u) r,
     Nat0 t) =>
    TSHandle a t2 -> TSIO a
                      (t, u)
                      (S (S t), r3)
                      (TSHandle a (S t))
```

Assessment

Pros:

- Explicit, timely disposal
- No list of open handles at run time

Cons:

- Type-class tomfoolery
- Handle count must be statically known
- Unwieldy error handling

Conclusion

Two ways to manage multiple scarce resources (file handles, . . .)

- Monadic regions (automatic; struggle for timeliness)
- Type-state tracking (manual; struggle for safety)

Static guarantees

- No access after closing
- Predictable, flexible, timely disposal

Compatible with

- Error handling
- General recursion
- Higher-order computations
- Mutable state