Lightweight Monadic Regions

Oleg Kiselyov
FNMOC
oleg@pobox.com

Abstract

We present Haskell libraries that statically ensure the safe use of resources such as file handles. We statically prevent accessing an already closed handle or forgetting to close it. The libraries can be trivially extended to other resources such as database connections and graphic contexts.

Because file handles and similar resources are scarce, we want to not just assure their safe use but further deallocate them *soon* after they are no longer needed. Relying on Fluet and Morrisett's [4] calculus of *nested* regions, we contribute a novel, improved, and extended implementation of the calculus in Haskell, with file handles as resources.

Our library supports region polymorphism and *implicit* region subtyping, along with higher-order functions, mutable state, recursion, and run-time exceptions. A program may allocate arbitrarily many resources and dispose of them in any order, not necessarily

LIFO. Region annotations are part of an expression's *inferred* type. Our new Haskell encoding of monadic regions as monad transformers needs no witness terms. It assures timely deallocation even when resources have markedly different lifetimes and the identity of the longest-living resource is determined only dynamically.

For contrast, we also implement a Haskell library for manual resource management, where deallocation is explicit and safety is assured by a form of linear types. We implement the linear typing in Haskell with the help of phantom types and a parameterized monad to statically track the type-state of resources.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.4.2 [*Operating Systems*]: Storage Management—Allocation/deallocation strategies

General Terms Design, Languages

Keywords monads, parametric polymorphism, regions, resource management, subtyping, type classes, type systems, effect systems

1. Introduction

The typical program uses various kinds of resources: memory, file handles, database connections, locks, graphic contexts, device reservations, and so on. Two goals recur in the management of these resources: safe access and timely disposal. First, a resource can be

Haskell'08, September 25, 2008, Victoria, BC, Canada.

Chung-chieh Shan

Rutgers University

ccshan@cs.rutgers.edu

used only after it is allocated and before it is deallocated. Second, each resource should be deallocated exactly once, soon after it is no longer needed. This paper presents new ways for a Haskell program to *statically* assure these goals.

Heap memory is one important resource, which Haskell systems manage automatically and admirably. Garbage collection allows a sound type system to statically assure the safety of memory access, but makes it hard to predict when a particular piece of memory that is no longer needed will be deallocated. Memory is generally a plentiful resource, so timely deallocation is not vital.

The other kinds of resources named above are scarce, so timely deallocation is integral to their proper management. Such resources are the topic of this paper. Alas, Haskell itself offers little support

for managing these resources. Finalizers are not a solution, as there are few guarantees on when they are run, if they are run at all.[1]

The bulk of this paper demonstrates the use of *regions* to manage these resources. Regions were originally introduced by Tofte and Talpin [18] for automatic memory management, but can be used for other kinds of resources. A lexically scoped construct 'letregion ρ in e' creates a new region labeled ρ and evaluates the expression e, which can use the region to hold heap-allocated data. When e finishes, the region along with all its data is disposed of. The lifetimes of regions and their data properly nest and reflect the block structure of the program. As a resource management technique, regions stand out by not only assuring safe access statically but also offering predictable deallocation that is timely in many cases. Region memory management is implemented in MLKit [17] and Cyclone [5]. Fluet and Morrisett [4] mention a Haskell implementation of regions that uses subtyping witness terms.

Five main difficulties with regions are

1. to statically ensure that resources allocated in a region cannot be used outside the region: Tofte and Talpin [18] introduced a non-trivial effect system for this purpose.
2. to maintain region polymorphism and subtyping: The user of some resources should not care exactly which regions they belong to, so long as they can be accessed safely.
3. to keep the notation convenient: The programmer should not have to provide numerous type and region annotations and coercions.
4. to allow resource lifetimes that do not nest properly: It should

be possible to deallocate resources not in the reverse order of their allocation.

[1] Simon Marlow said that GHC provides "no guarantee that a finalizer will be run before your program exits. The only thing you can be sure of is that it will *not* run while the `ForeignPtr` is still reachable by the garbage collector." `http://www.haskell.org/pipermail/haskell-cafe/2007-May/025455.html`

5. to allow resource lifetimes that are not known statically: From several allocated resources, one might be dynamically chosen to be used for the rest of the computation, and the rest deallocated.

This paper solves all these problems in Haskell, extending Fluet and Morrisett's work [4].

For concreteness, we use file handles as the resources in this paper. An open file handle is created by opening a file, then used to read and write the file. Closing the handle dissociates it from the file; once a handle is closed, it must not be used. All opened handles must be closed, preferably soon after they are no longer needed.

1.1 Motivating example

We use the following motivating example inspired by real life:

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.

The example demonstrates creating and using several file handles whose lifetimes are not properly nested. It captures the frequent situation of reading a configuration file, opening files named there, closing the configuration file, then working with the other files. Some of those other files, like log files, stay open until the program ends. We aim to implement this example in a way that statically ensures that only open file handles are used and all file handles are closed in a timely manner, even in the face of potential IO failures.

1.2 Our contributions

We develop two libraries of *safe handles* for file IO. Whereas the low-level handles provided by standard Haskell can be used unsafely, our libraries statically guarantee that all accessible safe handles are open. By an *accessible* handle, we mean a value (perhaps a variable in scope) that can be passed to an IO operation (such as reading or writing) without causing a type error. The libraries further ensure that all created safe file handles are closed predictably. We sometimes call safe handles just handles.

Our first library, the library of lightweight monadic regions, is a novel encoding of the region calculus $\mathrm{F}^{\mathrm{RGN}}$ [4] in Haskell. Like other encodings of regions in Haskell, our encoding is inspired by the `ST` monad, so it uses phantom types and rank-2 polymorphism to label and encapsulate a region's computations and its open handles. Unlike other encodings, our encoding maintains region polymorphism and subtyping without passing and applying witness

terms (coercions). Region subtyping is decided entirely at the type level and incurs no run-time overhead.

The gist of our solution is to build a family of monads by applying an `ST`-like monad transformer repeatedly to the `IO` monad. This family represents the nesting of regions. To make region subtyping implicit, we implement a type-class constraint that checks whether one monad in the family is an ancestor of (and thus is safe to coerce to) another. This implementation only compares types for equality, never inequality, so it is lightweight compared to a previous encoding of implicit region subtyping [7]. That encoding describes region nesting using a type-level list of quantified type variables; it requires the controversial extension of incoherent instances to check for membership in the list.

This new library solves all five problems with regions identified above, including the last problem, that of statically unknown lifetimes. The library ensures resource disposal even in the presence of run-time exception handling. The run-time overhead of the library is lighter than tracking handle state in standard Haskell IO.

For comparison, we describe another novel approach: safe manual resource management. In this approach, the programmer closes each handle explicitly, after it is no longer needed. The type system prevents access to closed handles, using a parameterized monad to track type-state. This tracking is purely at the type level, so this solution has no run-time overhead. Alas, the number of handles open at any time must be statically known, and it is unwieldy to recover from errors given that most IO operations can fail.

1.3 Structure of the paper

Section 2 develops a version of Haskell's `ST` monad for file han-

dles rather than memory references. This section presents the bulk of our library; the subsequent sections add remarkably small extensions to ensure timely disposal. Section 3 introduces regions for resource management and implements Fluet and Morrisett's F^{RGN} calculus [4] using explicit witness terms of region subtyping, again for file handles rather than memory references. These introductory sections motivate the use of *implicit* region subtyping, resolved at the type level. This usability improvement gives rise to our final library of lightweight monadic regions in Section 4.

Section 5 describes an extension to our library that lets us prolong the lifetime of a dynamically chosen handle by promoting it to an ancestor region. Section 6 presents an alternative approach: manual resource management with safety ensured statically by a type system that tracks type-state. We describe several drawbacks of that approach, in particular, its apparent incompatibility with the fact that most IO operations can fail. We then review related work and conclude.

Our complete code is available at `http://okmij.org/ftp/Computation/resource-aware-prog/`.

2. Safe file IO in a single region

We start by drawing inspiration from an analogy between the safety of file handles and the safety of memory references. First, we want to access only open file handles, just as we want to access only references to allocated memory. Second, all open file handles must be closed, just as all allocated memory must be freed.

Haskell's `ST` monad and its `STRef` values guarantees such memory safety [11, 14] by using the *same* type variable `s` to tag both the type of memory references (`STRef s a`) and the type of computa-

tions using these references (`ST s b`). Not only are `ST` and `STRef` both abstract type constructors, but `runST`, the function for running `ST` computations, has a rank-2 type that universally quantifies over the type variable `s` and prevents it from 'leaking'. (This quantification makes `s` an *eigenvariable* or a fresh name [13, 15].) Hence, in a well-typed program, all accessible memory references are to allocated memory and can be disposed of once the `ST` computation ends. As is usual for automatic memory management, this memory safety depends on there being no way to deallocate memory explicitly. The `ST` monad thus satisfies all of our requirements except timeliness: an allocated `STRef` persists until the end of the whole computation.

The untimeliness of deallocation may be tolerable for memory, but not for file handles because they are scarcer. Nevertheless, the `ST` monad offers a good starting point. In the later sections we extend it to dispose of allocated resources sooner.

2.1 Interface

We provide the monad `SIO s`, which is analogous to the monad `ST s`, and *safe handles* of type `SHandle m` labeled by a monad `m` (which in this section is always `SIO s`), which are analogous to memory references of type `STRef s a`. For a user of our library, these types are abstract. The values of these types can only be manipulated by the following functions:

```
runSIO     :: (forall s. SIO s v) -> IO v
newSHandle :: FilePath -> IOMode ->
              SIO s (SHandle (SIO s))

shGetLine  :: SHandle (SIO s) -> SIO s String
```

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

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

The function `newSHandle` replaces standard Haskell's `openFile`: it opens the file identified by `FilePath` for reading, writing, etc., as specified by `IOMode`. The function, if successful, returns a safe handle, labeled by the monad of the computation that created the handle. The result type of `newSHandle` is quite like that of `newSTRef`; in particular, the type of the returned `SHandle` contains the type parameter `s` of the `SIO` monad. Any safe handle that `newSHandle` yields is open and associated with the successfully opened file, but of course `newSHandle` may fail—perhaps the file does not exist or the process does not have permission to manipulate the file. In that case, the function crucially does not yield a safe handle, but instead throws an `Exception` describing the problem. The exceptions may be caught and handled in the `SIO s` monad using the functions `shCatch` and `shThrow`, which are direct analogues of the Haskell functions `catch` and `throwIO`.

The convenience function `shReport` prints a progress or debugging message on `stderr`. The function `shGetLine` reads a newline-terminated string from the file associated with a handle; `shPutStrLn` writes a newline-terminated string to the file, and `shIsEOF` tests if the file has any more data to read. These functions are again direct analogues of the corresponding functions in Haskell's `System.IO` library. They all can throw exceptions.

The function `runSIO` executes the `SIO s` computation and returns its result. The rank-2 type of `runSIO`, along with the type `s` threaded throughout the computation and labeling the handles, ensures that the result of `runIO` may not contain safe handles created in the `SIO s` computation, nor computations that include such handles. After the computation finishes, since it is no longer possible to use the handles created in the computation, `runSIO` may safely close them all. If the computation raises an uncaught exception, `runSIO` will re-raise the exception, again after closing all handles created in the computation. As with the `ST` monad, our library does not export any analogue of `hClose`. In this section, all handles stay open until the computation ends, when `runSIO` closes them.

The functions of our library let us write our motivating example, §1.1, without the timely disposal of resources. To show that we do not have to put all IO code in one big function, we divide the computation into two functions, `test3` and `test3_internal`, which share the useful combinator `till`.

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

test3 = runSIO (do
  h1 <- newSHandle "/tmp/SafeHandles.hs" ReadMode
  h3 <- test3_internal h1
  till (shIsEOF h1)
       (shGetLine h1 >>= shPutStrLn h3)
  shReport "test3 done")
test3_internal h1 = do
```

```
  h2 <- newSHandle "/tmp/ex-file.conf" 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)
  shReport "Finished zipping h1 and h2"
  return h3
```

The function `test3_internal` opens the configuration file, reads the name of the output file, and opens it to create the handle `h3`. It then zips the two input handles `h1` and `h2` into `h3`. Finally it returns the handle `h3` it created. Back in `test3` we copy the rest of `h1` to `h3`. The type of `test3_internal` is inferred to be

```
SHandle (SIO s) -> SIO s (SHandle (SIO s))
```

as expected: a computation that receives and returns a safe handle.

Executing `test3` produces the new file and writes the following on `stderr`:

```
Finished zipping h1 and h2
test3 done
Closing {handle: /tmp/t1}
Closing {handle: /tmp/ex-file.conf}
Closing {handle: /tmp/SafeHandles.hs}
```

All opened handles are indeed closed, after the end of `test3`, the whole computation.

We can ensure even more safety properties of handles. For example, by annotating each `SHandle` with an additional phantom

type label RO, WO, or RW, we can prevent writing to a read-only (RO) handle. This safety property is so easy to guarantee that one would think it would have been specified in Haskell 98.

2.2 Implementation

We implement this interface in the file SafeHandles1.hs in the accompanying source code. The file has limited export and hence constitutes the 'security kernel'. Only the functions in this file know and can manipulate the representation of the otherwise opaque SIO and SHandle. Our implementation internally relies on standard Haskell IO.

The IO monad with safe handles SIO is implemented as a monad transformer IORT applied to IO. The monad transformer seems overkill here, yet it prepares us for the final solution.

```
newtype IORT s m v =
  IORT{ unIORT:: ReaderT (IORef [Handle]) m v }
  deriving (Monad)

type SIO s = IORT s IO
```

To close all handles created in the computation at the end, we have to maintain the list of all created handles and update it whenever a new handle is created. Since we use the IO monad internally, we implement this mutable list as IORef [Handle]. We store a list of Handles rather than SHandles because the list is never exposed to the user. Although the contents of this mutable cell changes as handles are opened, the location of the cell itself does not change. We therefore make this location available throughout the computation using a reader monad transformer (we could have used implicit parameters or implicit configurations).

Since our library does file IO, we may be tempted to make the `IORT` transformer an instance of `MonadIO`. That would however let the programmer inject any `IO` action into `SIO`, in particular actions that open and close low-level handles, and defeat IO safety. Instead, we define our own version of `MonadIO`, called `RMonadIO`, and keep it unexported. Whereas `MonadIO` only supports lifting `IO` actions, our `RMonadIO` also supports 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
  brace = ...
  snag  = ...
  lIO   = id
```

The `lIO` method is same as the `liftIO` method in Haskell's `MonadIO` class, but the `brace` and `snag` methods are new: they let us clean up after exceptions, generalizing `bracket` and `catch` in the `Control.Exception` module.

The IO exception raised by various `System.IO` functions may include a low-level `System.IO.Handle`. To make sure the low-level handle does not escape the `SIO` computation in an exception, we define `brace` and `snag` to remove the handle from the IO exception before re-raising it or passing it to the exception handler.

It is easy to lift this functionality through the `ReaderT` and thus `IORT` transformers, so that `IORT s IO` is an instance of `RMonadIO`.

```
instance RMonadIO m => RMonadIO (ReaderT r m) where
  brace before after during = ReaderT (\r ->
    let rr m = runReaderT m r
    in brace (rr before) (rr.after) (rr.during))
  snag m f = ReaderT (\r ->
    runReaderT m r `snag` \e -> runReaderT (f e) r)
  lIO = lift . lIO

instance RMonadIO m => RMonadIO (IORT s m) where
  brace before after during = IORT
    (brace (unIORT before) (unIORT.after)
           (unIORT.during))
  snag m f = IORT (unIORT m `snag` (unIORT . f))
  lIO = IORT . lIO
```

We can now define `runSIO` to run an `SIO` action that tracks open handles dynamically. At the beginning of the action, `runSIO` initializes the list of created handles to empty. At the end, even if an exception was thrown, it tries to close every handle in the list.

```
runSIO :: (forall s. SIO s v) -> IO v
runSIO m = brace (lIO (newIORef [])) after
                 (runReaderT (unIORT m))
  where after handles =
          lIO (readIORef handles >>= mapM_ close)
        close h = do
          hPutStrLn stderr ("Closing " ++ show h)
          catch (hClose h) (\e -> return ())
```

A safe handle `SHandle` is a simple wrapper around a low-level

`Handle`, with the same run-time representation. The sole argument to the `SHandle` type constructor is the monad `m` where the handle belongs. Like the data constructor `IORT`, the data constructor `SHandle` is of course private to the security kernel.

```
newtype SHandle (m :: * -> *) = SHandle Handle
```

The implementation of `newSHandle` shows that it is like `openFile` of the Haskell IO library. However, it does extra bookkeeping at run time: adding a new handle to the list of open handles.

```
newSHandle :: FilePath -> IOMode ->
              SIO s (SHandle (SIO s))
newSHandle fname fmode = IORT r'
 where r' = do h <- lIO $ openFile fname fmode
               handles <- ask
               lIO $ modifyIORef handles (h:)
               return (SHandle h)
```

The other functions, which manipulate existing handles, are trivial to implement. Only their types (shown above) are interesting.

```
shGetLine  (SHandle h) = lIO (hGetLine h)
shPutStrLn (SHandle h) = lIO . hPutStrLn h
shIsEOF    (SHandle h) = lIO (hIsEOF h)

shThrow  = lIO . throwIO
shCatch  = snag
shReport = lIO . hPutStrLn stderr
```

2.3 Assessment

Our `SIO` library in this section statically ensures that all accessible

safe handles are open and that all opened handles are closed exactly once. As with the `ST` monad, we achieve this safety even in the presence of general recursion and run-time exceptions.

Although `newSHandle` and `runSIO` incur some run-time overhead to maintain the list of open handles and to close them, the other, more frequently used handle-manipulation functions such as `shPutStrLn` do not even have to check that the handle is open or look for it in the list of open handles, because all safe handles are assured open statically. Thus, safe handles would be faster to use overall if we implement them not in standard Haskell but using an optimized low-level IO library that does not store or check whether a handle is open at run time.

GHC provides a function `withFile` that combines opening and closing a file. Its type

```
FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

suggests that the newly created handle cannot be accessed once closed. To the contrary, nothing prevents the user from including the handle in the result of `withFile`.

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

The handle above will be closed by the time `hGetLine` gets it. Hence `withFile` does not satisfy our most important desideratum, that all accessible handles be open.

On the downside, safe handles are deallocated hardly in a timely manner. Rather, they are closed only at the very end of the computation. We turn to nested regions to solve this problem.

3. Nested regions using explicit witness terms

In the rest of this paper, we organize an IO action into regions whose run-time lifetimes are properly nested, unlike those of handles. Each handle belongs to a region and can be used in that region or any descendant region. During a region's lifetime, a handle can be created in that region even if a younger (descendant) region is also live. When a region exits, all handles in that region are closed.

For example, we may create a handle `h1` in the current region, then another handle `h2` in the parent region, which lasts longer. When the current region exits, `h1` will be closed but not `h2`. The handle `h2` can be returned as the result of the current region and used until the parent region exits. Thus, although regions are properly nested, we can still open and close handles without properly nested lifetimes by assigning a handle to an ancestor region.

In short, we apply Fluet and Morrisett's region calculus $\mathrm{F}^{\mathrm{RGN}}$ [4] to the case of IO handles. In this section, independently of their implementation we implement their encoding of region subtyping in explicit witness terms by slightly changing our code in the previous section.

3.1 A first attempt with not enough sharing

To dispose of different handles at different times, we may try to *nest* `SIO` computations—that is, run one `SIO` computation within another. Just like an `SIO` computation run using `runSIO`, the child `SIO` computation closes all safe handles it creates when it finishes, because these handles are encapsulated in the child computation and never needed by the parent computation. Thus we approach our ideal that the resources are deallocated soon after they are no longer needed. To introduce child `SIO` computations, we add to our library the following function with a trivial implementation:

```
newNaiveReg :: (forall s. SIO s v) -> SIO r v
newNaiveReg m = lIO (runSIO m)
```

As with `runSIO`, the rank-2 type of `newNaiveReg` statically prevents handles created in the child computation (whose type contains `s`) from being used outside of the child computation, so they can safely be closed once the child computation is finished.

Here is an example using `newNaiveReg`.

```
test2' = runSIO (do
  h1 <- newSHandle "fname1" ReadMode
  l1 <- shGetLine h1
  let op = newSHandle "fname3" ReadMode

  res <- newNaiveReg (do
    h2 <- newSHandle "fname2" ReadMode
    h3 <- op
    -- l1 <- shGetLine h1
    l2 <- shGetLine h2
    l3 <- shGetLine h3
    return (l1 ++ l2 ++ l3))

  h3 <- op
  l3 <- shGetLine h3
  return l1)
```

The main `SIO` computation opens the safe handle `h1` and reads the line `l1` from it. The child computation opens two other handles `h2` and `h3`, reads from them, and combines the result with `l1`. This child computation accesses `l1` by lexical scope, but it could be as well defined in a separate function. Either way, the safe handles `h2`

and `h3` of the child computation are freed once `res` is computed.

Two computations can share values such as `l1` and `res`, even unexecuted actions such as `op` to access the same file. However, two computations may *not* share any value that includes a safe handle created within a region or a computation involving that handle. For example, the child computation may not return `h3`. Also, although the handle `h1` takes lexical scope over the child computation, it cannot be used there: uncommenting the line `l1 <- shGetLine h1` gives a type error, because the two computations have different phantom types `s` and `r`.

Although we have achieved both handle safety and timely disposal, `newNaiveReg` fails to express our main example, §1.1. To show this failure, we consider the three files involved one by one.

1. The configuration file should be closed before the two other files in the example. Hence, we want to open the configuration file in a child region.
2. The other input file is used after the configuration file is closed. Hence, that input file has to be opened in the parent region. However, the child computation also needs to read from it, which `newNaiveReg` prohibits.
3. The output file needs to be opened in the lifetime of the configuration file handle—that is, in the child computation. However, the parent computation also needs to write to it, which `newNaiveReg` again prohibits.

In sum, `newNaiveReg` is too restrictive because it does not let a child computation create and access file handles that belong to parent regions. We proceed to relax this restriction while preserving the safety invariant that all accessible handles are always open.

3.2 Using a parent region from a child computation

One approach to fixing `newNaiveReg` is to follow how Launchbury and Sabry [12] add nesting to `runST`. In our notation, they redefine `newNaiveReg` as follows.

```
newLSRgn :: (forall s. SIO (r,s) v) -> SIO r v
```

The tuple of labels `(r,s)` thus encodes the nesting of the child region `s` in the parent region `r`. A new operation

```
importSHandle :: SHandle (SIO r) ->
                 SHandle (SIO (r,s))
```

lets the child computation access handles in the parent region. Alas, as extensively discussed by Fluet and Morrisett [4, Section 2], this solution lacks *region polymorphism*: within a child computation, we would like to use (even create) safe handles in *any* ancestor region, without nailing down the exact lineage of each handle.

Fluet and Morrisett [4] introduce a solution with region polymorphism. The essential idea is to coerce a parent computation, which may allocate and use handles in the parent region, to a child computation, which can be composed with actions that allocate and use handles in the child region. The coercion function is the constructive proof (or *witness*) of *region subtyping*: every parent-region computation can serve as a child-region computation. As Fluet and Morrisett [4, Section 4] put it, "we consider a region to be a subtype of all the regions that it outlives."

We implement Fluet and Morrisett's solution by adding one type and one function to the interface of our `SIO` library.

```
newtype SubRegion r s =
  SubRegion (forall v. SIO r v -> SIO s v)

newRgn :: (forall s. SubRegion r s -> SIO s v) ->
          SIO r v
```

A value of the type `SubRegion r s` witnesses that the region labeled `r` is an ancestor of the region labeled `s`. The witness is constructive in that it lets us coerce any computation of type `SIO r v` into a computation of type `SIO s v`. These witnesses can easily be composed as functions, so `r` may not be the immediate parent of `s`, just an ancestor or `s`. The witness is polymorphic in the type `v` of the value produced by the computation. Although GHC's "Liberalised type synonyms" extension lets us define `SubRegion` not as a `newtype` but as a type synonym, we keep `newtype` to emphasize the polymorphism and to avoid having to specify the types of these witnesses explicitly.

As the types of `newRgn` and `newNaiveReg` show, the new function `newRgn` is like `newNaiveReg` in that it lets a parent computation include a child computation, but `newRgn` provides a subtyping witness to the child computation so that it can use and create handles in the parent region. (Our `newRgn` is Fluet and Morrisett's `letRGN`, and our `runSIO` is their `runRGN` [4, Figure 1].)

Using `newRgn`, we can finally express our main example.

```
test3 = runSIO (do
  h1 <- newSHandle "/tmp/SafeHandles.hs" ReadMode
  h3 <- newRgn (test3_internal h1)
  till (shIsEOF h1)
```

```
      (shGetLine h1 >>= shPutStrLn h3)
  shReport "test3 done")
test3_internal h1 (SubRegion liftSIO) = do
  h2 <- newSHandle "/tmp/ex-file.conf" 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))
  shReport "Finished zipping h1 and h2"
  return h3
```

The code is almost same as in §2, except `test3_internal` runs in a child region and receives as its second argument a subtyping witness `liftSIO` from the parent region. This crucial difference can be seen in the `stderr` transcript of running this code:

```
Finished zipping h1 and h2
Closing {handle: /tmp/ex-file.conf}
test3 done
Closing {handle: /tmp/t1}
Closing {handle: /tmp/SafeHandles.hs}
```

Unlike in §2, the configuration file `h2` is closed as soon as it is no longer needed (that is, when its user `test3_internal` ends). When the whole `test3` ends, only the two other safe handles have to be closed.

The child computation `test3_internal` uses the coercion

function `liftSIO` extensively to access the handles `h1` and `h3` in the parent region. It also uses `liftSIO` to coerce the action of opening the output file, `newSHandle fname WriteMode`, so as to create the safe handle in the parent region and label its type with the parent monad. Hence, the output file handle is not closed when `test3_internal` finishes and can be returned as the result of `test3_internal`.

The inferred type for `test3_internal` is

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

Compared to the type of `test3_internal` in §2, this type shows region polymorphism: the input and the output handles need only belong to *a* parent region.

3.3 Implementation

We implement this extended interface in the accompanying source file `SafeHandlesFM.hs`. Little differs from §2.2: the definitions of `IORT`, `SIO`, `SHandle`, and `RMonadIO` all remain the same, along with all operations on safe handles such as `shPutStrLn`. Even `newSHandle` and `runSIO` do not need to change. The key lies in the implementation of the only new function, `newRgn`:

```
newRgn :: (forall s. SubRegion r s -> SIO s v) ->
          SIO r v
newRgn body =
  IORT (do env_outer <- ask
           let witness (IORT m) =
                 lIO (runReaderT m env_outer)
```

```
lIO (runSIO (body (SubRegion witness))))
```

The last line above is `newNaiveReg`. The rest of the code builds a subtyping witness so that the child computation may use the parent region: At the type level, `SubRegion r s` transfers the capabilities of the ancestor region `r` to the descendant region `s`. At the term level, the witness passes the mutable cell containing the list of low-level handles from the parent's environment to the child, so the child computation may access the parent region's run-time state.

3.4 Assessment

We have achieved our goals: we can now close a safe handle well before the overall computation ends, yet all accessible handles are still always open.

Just as in §2, our region discipline is compatible with exception handling, in particular, exceptions when opening files. The following code illustrates handling an exception raised in a nested region.

```
test_copy fname_in fname_out = do
  hout <- newSHandle fname_out WriteMode
  (do newRgn (\(SubRegion liftSIO) -> do
        hin <- newSHandle fname_in ReadMode
        till (shIsEOF hin)
             (shGetLine hin
              >>= liftSIO . shPutStrLn hout))
      shReport "Finished copying")
   `shCatch` \e -> do
     shReport ("Exception caught: " ++ show e)
     shPutStrLn hout ("Copying failed: " ++ show e)
```

This code copies the file at `fname_in` to `fname_out`.

```
> runSIO (test_copy "/etc/motd" "/tmp/t1")
Closing {handle: /etc/motd}
Finished copying
Closing {handle: /tmp/t1}
```

If the file at `fname_in` cannot be opened, then the input handle `hin` is not created. Instead, an error message is written to `fname_out`.

```
> runSIO (test_copy "/non-existent" "/tmp/t1")
Exception caught: /non-existent: openFile:
   does not exist (No such file or directory)
Closing {handle: /tmp/t1}
```

As these transcripts confirm, all open handles (and nothing else) are closed at the end, whether an exception was raised or not.

One may remark that our `SIO` library still lacks an explicit close operation. Indeed, for a resource-management technique to be automatic and safe, the user cannot be allowed to deallocate resources explicitly. With regions, the lack of an explicit close operation is *not* a principled drawback! The user may create arbitrarily many regions—even one region per handle—and can raise an exception to forcefully and predictably end a region and close its handles.

On the downside, we see from `test3_internal` that using nested regions requires applying many coercions. It is easy to forget to apply `liftSIO`, which would identify `t` and `s` in the inferred type of `test3_internal` and make the function region-monomorphic. The error will be reported at a different place, when we try to use a region-monomorphic `test3_internal` in `test3`.

A related drawback is that witnesses must be passed explicitly, which becomes annoying when many handles are in use and we want to maintain region polymorphism. For example, suppose we

write a function that copies a line from one handle to another.

```
test4 h1 h2 = do line <- shGetLine h1
                 shPutStrLn h2 line
```

The inferred type

```
SHandle (SIO s) -> SHandle (SIO s) -> IORT s IO ()
```

is region-monomorphic: both input handles must belong to the region of the computation. To maintain full region polymorphism, we have to add *two* witness arguments to the function and apply them without mixing them up.

```
test43 h1 h2 (SubRegion liftSIO1)
             (SubRegion liftSIO2) = do
  line <- liftSIO1 $ shGetLine h1
  liftSIO2 $ shPutStrLn h2 line
```

The new inferred type below allows the two input handles to belong to any parent region of the current computation.

```
SHandle (SIO t1) -> SHandle (SIO t2) ->
SubRegion t1 s -> SubRegion t2 s -> SIO s ()
```

To remove the burden of dealing with witnesses explicitly, we want the type system to generate and pass witnesses automatically. Fluet and Morrisett [4] already anticipated this desire and how we fulfill it, by treating `SubRegion` as an abstract type constructor.

4. Nested regions as monad transformers

The desired implicit subtyping among regions is reminiscent of the implicit subtyping among effects that denotes monad morphisms [2]. Indeed, we can treat a `SubRegion` witness as a monad mor-

phism, and a call to `newRgn` as reifying an effectful computation. That is, for one region to spawn another is for a monad transformer to apply to the parent region's computation monad.

Monads and monad transformers are poster-child applications of Haskell's type-class overloading mechanism. By treating, in this section, each new region as an application of a monad transformer, we use type classes to resolve region subtyping as well.

4.1 Interface

We generalize the `SIO` monad to a *family* of `SIO` monads. Each monad in the family is the result of applying zero or more monad transformers of the form `IORT s`, where `s` is a phantom type, to the `IO` monad. Each instance of the transformer has its own label `s`. For example, a monadic action of type `IORT s (IORT r IO) Int` computes an `Int` using a current region represented by the type `s` and a parent region represented by the type `r`. The monad `SIO s` is still a synonym for `IORT s IO` and belongs to the family.

We thus arrive at our library of lightweight monadic regions, the `SIO` library. Fig. 1 gives the full interface. It includes operations on `IORef` cells, used in §4.3 to demonstrate handle safety in the presence of mutable state, and `shDup`, to be explained in §5.

Since we wish to execute only safe `IO` computations in the `SIO` family, `IORT s` is not an instance of the `MonadTrans` or `MonadIO` classes, to prevent lifting of arbitrary `IO` computations. We use our own class `RMonadIO` (defined in §2.2) as the kind predicate of the `SIO` family. The class `RMonadIO` is not exported from the library and hence closed to instances added by the user.[2]

Another important type-level predicate is `MonadRaise`, also a closed type-class. The constraint `MonadRaise m1 m2` holds when

m2 is the result of applying zero or more monad transformers IORT s to m1. In other words, MonadRaise m1 m2 holds if m1 is an ancestor of m2 in the SIO family. Thus, the multiparameter type-class MonadRaise m1 m2 controls implicit region subtyping.

To create a new region, the library exports a function newRgn. The rank-2 type of newRgn shows that we prepend a new label s to the monad of the parent region m to make the monad of the child region. No explicit subtyping witness is passed.

The type of newSHandle allows creating a handle in any monad of the SIO family *except* IO, that is, in a 'safe' monad only. As before, the handle is labeled with the monad in which it is created.

The remaining operations are the same as before, but with more general types. Operations such as shThrow work not just in the SIO monad but in the whole family of SIO monads. Operations on handles such as shGetLine also work in any member m2 of the SIO family, as long as m2 descends from the monad m1 where the handle was created. So, the computation shGetLine h can be executed in any descendant of the monad that created the handle h. In particular,

[2] To let the user write signatures of functions using the SIO library, we should export a synonym of RMonadIO. The original RMonadIO remains closed. We introduced this method of defining closed classes before [10].

```
type IORT s m v  -- opaque
type SIO s v     = IORT s IO
type SHandle m   -- opaque

runSIO      :: (forall s. SIO s v) -> IO v
newRgn      :: RMonadIO m =>
               (forall s. IORT s m v) -> m v
liftSIO     :: Monad m => IORT r m a ->
```

```
                IORT s (IORT r m) a

newSHandle  :: RMonadIO m => FilePath -> IOMode ->
               IORT s m (SHandle (IORT s m))

shGetLine   :: (MonadRaise m1 m2, RMonadIO m2) =>
               SHandle m1 -> m2 String
shPutStrLn  :: (MonadRaise m1 m2, RMonadIO m2) =>
               SHandle m1 -> String -> m2 ()
shIsEOF     :: (MonadRaise m1 m2, RMonadIO m2) =>
               SHandle m1 -> m2 Bool

shThrow     :: RMonadIO m => Exception -> m a
shCatch     :: RMonadIO m => m a ->
               (Exception -> m a) -> m a
shReport    :: RMonadIO m => String -> m ()

sNewIORef   :: RMonadIO m => a -> m (IORef a)
sReadIORef  :: RMonadIO m => IORef a -> m a
sWriteIORef :: RMonadIO m => IORef a -> a -> m ()

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

Figure 1. The interface of the final SIO library

because the `IO` monad cannot create any safe handle and is not a descendant of any other monad, it cannot execute `shGetLine h`.

To perform an action in the parent region such as creating a

safe handle, the library exports the function `liftSIO`. The type of `liftSIO` is like `SubRegion r s` in §3.2. Whereas `liftSIO` only witnesses immediate parenthood among regions, a `SubRegion` value can witness any ancestry. This difference is not a big deal because `liftSIO` can be iterated to reach any ancestor. A more substantial difference between the `liftSIO` function and `SubRegion` values is that we need not pass `liftSIO` around to maintain region polymorphism, because all operations on existing handles, such as `shGetLine`, are already region-polymorphic. Rather, `liftSIO` is typically used only when *creating* a handle, to specify the ancestor region to which the new handle should belong.[3]

To show the concision of implicit subtyping that this interface affords, we rewrite our running example `test3`. We show only `test3_internal` below, as the code for the main function `test3` remains unchanged from §3.2.

```
test3_internal h1 = do
  h2 <- newSHandle "/tmp/ex-file.conf" ReadMode
  fname <- shGetLine h2
  h3 <- liftSIO (newSHandle fname WriteMode)
  shPutStrLn h3 fname
  till (liftM2 (||) (shIsEOF h2) (shIsEOF h1))
       (shGetLine h2 >>= shPutStrLn h3 >>
        shGetLine h1 >>= shPutStrLn h3)
  shReport "Finished zipping h1 and h2"
  return h3
```

[3] In rare cases, we have to apply `liftSIO` to maintain region polymorphism even when we are only operating on existing handles from parent regions. See the end of `SafeHandlesTest.hs`.

We use `liftSIO` only once, to create a new handle in a region other than the current one. All other uses of handles (be they from the same or ancestor regions) require no `liftSIO`. The handles of ancestor regions can be used just as they are. The extra argument to `test3_internal` is gone: witnesses for region subtyping are managed by the type checker.

The inferred type of `test3_internal` is region-polymorphic:

```
(RMonadIO m, MonadRaise m1 (IORT s (IORT r m))) =>
SHandle m1 ->
IORT s (IORT r m) (SHandle (IORT r m))
```

The type says that `test3_internal` must be executed in a region with a parent. The function receives a handle from some ancestor region and returns a handle in the parent region. That is indeed the most general type for `test3_internal`.

We achieve region polymorphism more directly than in §3.4. Now the function to copy a line from one safe handle to another

```
test4 h1 h2 = do
  line <- shGetLine h1
  shPutStrLn h2 line
```

has the *inferred* type

```
(MonadRaise m1 t, RMonadIO t, MonadRaise m11 t) =>
SHandle m1 -> SHandle m11 -> t ()
```

The type literally says that `test4` is an `SIO` computation whose arguments are two handles that must be open throughout the computation. The handles may belong to the same region or different ones. The regions may be the current region or its ancestors. The in-

ferred type is thus *region-polymorphic* and the most general. Unlike `test43` in §3.4, we no longer need to pass extra witness arguments or apply `liftSIO`.

4.2 Implementation

We implement the final version of our `SIO` library in the accompanying source file `SafeHandles.hs`. This version is only slightly different from the two previous versions. In particular, the representation of `IORT` and `SIO` remains the same. The main change from §3 at the term level is to replace `SubRegion` with `liftSIO`.

```
liftSIO = IORT . lift
```

All the handle operations, from `newSHandle` to `shReport` including `shGetLine` and `shCatch`, can be implemented exactly as in §2 and §3, just with more general types. Even `runSIO` remains the same. Better, `newRgn` is now simply `runSIO` with a more general type! Conversely, `runSIO` is just the specialization of `newRgn` to the case where the monad `m` is `IO`.

The only major addition of `SafeHandles.hs` is at the type level: implementing `MonadRaise`. A simple but approximate way to implement `MonadRaise` is to add the following instances.

```
instance Monad m =>
  MonadRaise m m
instance Monad m =>
  MonadRaise m (IORT s1 m)
instance Monad m =>
  MonadRaise m (IORT s2 (IORT s1 m))
instance Monad m =>
  MonadRaise m (IORT s3 (IORT s2 (IORT s1 m)))
```

This incomplete approximation does not require functional dependencies or more controversial extensions. We can still nest an arbitrary number of regions using `newRgn`, as well as create and use handles in an arbitrary ancestor region using `liftSIO`. The incompleteness only limits *implicit* region subtyping to three levels deep, so it does not matter much in practice, yet it is quite unsatisfying.

In full generality, the constraint `MonadRaise m1 m2` should hold whenever `m2` is the result of applying `IORT s` to `m1` any number of times (informally, whenever `m1` is a 'suffix' of `m2`). This general specification can be implemented directly and inductively—if we admit functional dependencies, overlapping instances, and undecidable instances.

```
instance Monad m => MonadRaise m m
instance (Monad m2, TypeCast2 m2 (IORT s m2'),
          MonadRaise m1 m2')
    => MonadRaise m1 m2
```

While short, the implementation is not trivial at all. It relies on the type-improvement constraint `TypeCast2` to delay unifying `m2` with `IORT s m2'`. The `TypeCast2` class is a variant for kind `*->*` of the `TypeCast` class. These powerful constraints are discussed in more detail elsewhere [8]; in particular, `TypeCast` and its cousin `TypeEq` are used to implement heterogeneous collections [9].

```
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
```

One may worry that the type-class trickery like `TypeCast2` may make inferred types and error messages hard to understand. That has not been the case. The inferred types clearly represent region subtyping by `MonadRaise` constraints, as illustrated by `test3_internal` and `test4` in the previous section. Some type error messages may seem unhelpful: leaking a handle from its region produces an error message about an inferred type being less polymorphic than expected (see §4.3 for discussion). Such error messages are also emitted when leaking an `STRef` in the ordinary `ST` monad, so they should be familiar to Haskell programmers.

4.3 Is handle safety truly guaranteed?

Our overall safety property is that all accessible handles are open, which means that the type system must never let a handle or any computation involving it be accessible ('leak') beyond its region's lifetime. For example, `test3_internal` should be allowed to return to `test3` the handle `h3`, but not `h2` because `h2` belongs to the child region. To this end, the type system assigns to `h2` the type `SHandle (IORT s (IORT r m))`. This type includes the eigenvariable `s`, which cannot escape the scope of `forall` in the type of `newRgn`. Thus, if we replace `return h3` in `test3_internal` by `return h2`, we get the type error

```
Inferred type is less polymorphic than expected
  Quantified type variable ‘s’ escapes
```

```
In the first argument of ‘newRgn’,
In a ’do’ expression:
  h3 <- newRgn (test3_internal h1)
```

This message adequately describes the error and pinpoints its location. In contrast, although the handle `h3` is also created in `test3_internal`, that `newSHandle` operation actually occurs in the parent region (thanks to `liftSIO`), so the type of `h3` only mentions the monad `IORT r m`. The type `r` is quantified in the type of `runSIO`, called by `test3`. Thus, `h3` may escape from `test3_internal` but not `test3`.

Instead of leaking a handle, we may try to leak a computation involving the handle:

```
do ac <- newRgn (do
           h2 <- newSHandle "fname2" ReadMode
           return (shGetLine h2))
   ac
```

This code is unsafe because `newRgn` closes the handle `h2` when the child computation exits. Executing the action `ac` outside of `newRgn` would attempt to read from an already closed handle. Fortunately, this code raises a type error. The handle `h2` has the type `SHandle (IORT s m)`, where `s` is quantified in the type of `newRgn`. The computation `shGetLine h2` therefore has the type

```
(MonadRaise (IORT s m) m2, RMonadIO m2) =>
m2 String
```

Since `do`-bindings are monomorphic, the type checker must resolve the two constraints above to infer a type for `ac`. Since `m2` is unknown, constraint resolution fails and yields a type error. In-

deed, every way to instantiate `m2` that satisfies the suffix constraint `MonadRaise (IORT s m) m2` mentions `s`, but the type of `ac`, which contains `m2`, cannot mention `s` because `ac` takes scope beyond `s`. It does not help to existentially quantify over `s` in the type of `ac`, because the type error would just shift to where `ac` is used.

We can try to defeat safety in many other ways. We may try to store the handle `h2` or a computation involving `h2` in a `IORef` mutable cell allocated outside `newRgn` (see `testref` in `SafeHandlesTest.hs`). But the type of `h2` includes the eigenvariable `s`, and the type of a computation involving `h2` contains either an unresolved `MonadRaise` constraint or the same `s`. Storing the handle or computation in the cell raises one of the type errors already described.

This variety of attempts, albeit unsuccessful, makes one wish for formal assurances. That is the subject of current work. Most promising is relating our library to Fluet and Morrisett's calculus $\mathbf{F}^{\mathrm{RGN}}$ [4]. The calculus seems a good fit since it supports first-class polymorphism, monads, and assignments—and has been proven type-sound [4, Theorem 1]. We need to add basic IO operations and monad transformers, then relate our `MonadRaise` to their `witnessRGN`. A more direct approach may be to follow Moggi and Sabry [14, Section 1, "Methodology and techniques"]. They instrument a structural operational semantics for a higher-order λ-calculus to check for region safety and dangling references, then establish type safety for the instrumented semantics.

5. Extension: prolonging the life of a handle

Our running example demonstrated deallocating resources in an arbitrary order, not necessarily the reverse order of their allocation.

Nevertheless, the order of deallocation is known when the resources are allocated. In fact, we know to close the output file last and the configuration file first, before even running the example. In practice, the lifetimes of handles may be known only after they are created. This uncertainty is illustrated by the following example due to Matthew Fluet:

1. open a configuration file;
2. read the names of two log files from the configuration file;
3. open the two log files and read a dated entry from each;
4. close the configuration file and the *newer* log file;
5. continue processing the *older* log file;
6. close the *older* log file.

Here, one of the two log files is used longer than the other, but we do not know which until we open both. Because the lifetimes of the handles are not statically known, it may seem that we cannot use regions to timely close the configuration file and the newer log file.

We can solve this problem while maintaining the static guarantees of the regions, by introducing the function `shDup` in Fig. 1. The function is akin to POSIX's `dup2` function: it copies a handle to produce an alias that, as the type indicates, belongs to the parent region of the handle. We end up with two handles that access the same file (and share the same offset). The file stays open until (and only until) all of its handles are disposed of.

Using `shDup`, we write our new example as follows:

```
test5 = runSIO (do
  h <- newRgn (test5_internal "/tmp/ex-file2.conf")
  l <- shGetLine h
  shReport ("Continue with the older file: " ++ l)
```

```
  shReport "test5 done")

test5_internal conf_fname = do
  hc <- newSHandle conf_fname ReadMode
  fname1 <- shGetLine hc
  fname2 <- shGetLine hc
  h1 <- newSHandle fname1 ReadMode
  h2 <- newSHandle fname2 ReadMode
  l1 <- shGetLine h1
  l2 <- shGetLine h2
  let (fname_old,h_old) | l1 < l2   = (fname2,h2)
                        | otherwise = (fname1,h1)
  shReport ("Older log file: " ++ fname_old)
  shDup h_old -- prolong the life of that handle
```

The guard `l1 < l2` chooses at run time which of the two handles needs further processing. We duplicate that handle and return the copy to the main function `test5`. Since the copy is assigned to the parent region, returning it is safe and well-typed (whereas we may return neither `h1` nor `h2`, as they are assigned to the child region). We cannot tell statically which of the two handles, `h1` or `h2` will be duplicated and returned. When `test5_internal` finishes with its region, all three handles opened there will be closed. However, since one of `h1` or `h2` has been duplicated, the closing will only decrement a reference count but not close the file. The main function `test5` receives a handle in its region for the unclosed file. When `test5` finishes, the file will be closed for real. The transcript of running `test5` confirms our description.

The implementation of `shDup` does not use `dup2` or other system calls. We merely need to add a reference count to our tracking

of handles assigned to regions. We introduce a data type

```
data HandleR = HandleR Handle (IORef Integer)
```

to associate a reference count with a low-level handle. It is easy to write the internal functions

```
new_hr   :: Handle -> IO HandleR
close_hr :: HandleR -> IO ()
eq_hr    :: Handle -> HandleR -> Bool
```

to create a `HandleR` with the initial reference count 1; to decrement the reference count and close the handle when the count goes to 0; and to compare a `Handle` against a `HandleR` for equality. We modify the representation of `IORT` so it maintains a mutable list of `HandleR`s rather than `Handle`s. Likewise, we modify the function `newRgn` to invoke `close_hr` instead of `hClose`, and `newSHandle` to invoke `new_hr`. The new function `shDup` is most interesting:

```
shDup (SHandle h) = IORT (do
   handles <- ask >>= lIO . readIORef
   let Just hr@(HandleR _ refcount) =
          find (eq_hr h) handles
   lIO  (modifyIORef refcount succ)
   lift (IORT (do        -- in the parent monad
                 handles <- ask
                 lIO (modifyIORef handles (hr:)))
   return (SHandle h))
```

To duplicate a handle, `shDup` locates the handle among those assigned to the current region, increments the reference count, and adds the handle to the list of handles of the parent region.

It incurs some dynamic overhead to allow duplicating handles:

`newSHandle` and `newRgn` have to maintain reference counts, and `shDup` also does work at run-time. All these extra computations are total and raise no errors. Operations on existing handles, such as `shGetLine`, are not affected at all! Thus, the overall run-time impact of adding handle duplication to the library is negligible.

6. Alternative: tracking type-state in a parameterized monad

A different way to ensure that resources are handled safely is to use an advanced type system to statically guarantee that *manual* resource management is safe. The programmer both allocates and deallocates resources, and so can deallocate a resource as soon as it is no longer needed. The type system ensures that a deallocated resource is not used, and that all resources are eventually deallocated. The type system thus tracks *type-state*, an "extension to the notion of type" to account for the fact that "the operations that can be performed on a variable depend not only on the type of the variable, but also upon the *state* of the variable" [16].

Tracking type-state seems to require quite advanced type (and effect) systems, in particular, modal or substructural type systems [3, 6, 19]. None of these advanced type systems are available in mainstream Haskell. Nevertheless, we can use a *parameterized monad* [1, 10] as a poor programmer's substructural type system to track type-state in Haskell. All such tracking is done at the type level and has no run-time overhead. Previously [10], we used a parameterized monad to enforce protocol constraints by tracking the number of times an operation is executed such as reading from a device register. We can treat the use of handles as a protocol too.

6.1 Interface

A parameterized monad is not a monad, and is not predefined in Haskell (although the upcoming version of GHC will support the `do`-notation for it). Its interface can be described as a type class.

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

A parameterized monad `m` has three type parameters `p q a`, compared to only `a` for a real monad. The parameter `a` is the type of the value produced by the computation. The types `p` and `q` describe the state when the computation begins and ends. The operations `gret` and `gbind` generalize `return` (unit) and `>>=` (bind) for an ordinary monad. The type of `gbind` threads the type state from one part of a computation to the next. For convenience, we define the left-associative low-precedence infix operator `>==` as a synonym for `gbind`. We also define the infix operator `+>>`, generalizing `>>` for ordinary monads, as `vm1 +>> vm2 = gbind vm1 (const vm2)`.

To track safe handles, we define a parameterized monad `TSIO s`, analogous to the single-region safe-IO monad `SIO s` of §2. We also introduce a type of safe handles `TSHandle s l`, labeled with a phantom type parameter `l` not present in `SHandle m` or `STRef s`. This label `l` is a Peano numeral at the type level that uniquely identifies the safe handle. The type state (tracked with `p` and `q`) contains a list of the labels of the currently open handles. To be more precise, the type state can be described in pseudo-Haskell by the following declarations of 'algebraic data kinds' and 'kind synonyms':

```
kind     TypeState    = (LabelCounter, OpenLabels)
```

```
kind     LabelCounter = Nat0
datakind OpenLabels   = N | C Label OpenLabels
kind     Label        = Nat0
datakind Nat0         = Z | S Nat0
```

That is, the type state is the counter to generate fresh labels, paired with the list of open labels. Our interface is as follows.

```
type TSIO s p q a  -- opaque
type TSHandle s l  -- opaque
runTSIO    :: (forall s. TSIO s (Z,N) (qc,N) a) ->
              IO a
tshOpen    :: (NewLabel p l p1, AddLabel l p1 q) =>
              FilePath -> IOMode ->
              TSIO s p q (TSHandle s l)
tshClose   :: RemLabel l p q =>
              TSHandle s l -> TSIO s p q ()
tshGetLine :: IsMember l p => TSHandle s l ->
              TSIO s p p String
tshPutStrLn:: IsMember l p => TSHandle s l ->
              String -> TSIO s p p ()
tshIsEOF   :: IsMember l p => TSHandle s l ->
              TSIO s p p Bool
tshReport  :: StateOK p => String -> TSIO s p p ()
```

The constraint `StateOK p` above checks that `p` is a well-formed type-state. The constraint `NewLabel p l p1` is a type-level function to generate a fresh label `l`. The other type-level functions `AddLabel l p q`, `RemLabel l p q`, and `IsMember l p` respectively add, remove, and search for the label in the type state. Opening a safe handle generates a fresh label and adds it to the open

label list of the type state. The label of a closed handle is removed from the type state, so `tshGetLine` cannot be invoked on a closed handle. To run a `TSIO` computation, the list of active labels in `p` and `q` must be empty (i.e., `N`), which means that no handle is open at the beginning and no handle can remain open at the end. We thus achieve all our goals. All this tracking takes place *entirely* in types: unlike Atkey's typed state [1], our type parameters `p` and `q` are phantom. At run-time, `TSIO p q a` is identical to `IO a` and `TSHandle l` is just the low-level `Handle`.

We have successfully implemented this approach (see the code and tests in `multi-handle-io0.hs`). In particular, we write our motivating example, §1.1, as follows.

```
test3 = runTSIO (
  tshOpen "/tmp/SafeHandles.hs" ReadMode >== \h1 ->
  test3_internal h1 >== \h3 ->
  till (tshIsEOF h1)
       (tshGetLine h1 >>= tshPutStrLn h3) >>
  tshReport "test3 done" +>>
  tshClose h1 +>>
  tshClose h3)

test3_internal h1 =
  tshOpen "/tmp/ex-file.conf" 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) >>
```

```
tshReport "Finished zipping h1 and h2" +>>
tshClose h2 +>>
gret h3
```

For any parameterized monad `m` and any type state `p`, the type constructor `m p p` is a monad. In particular, `TSIO s p p` is an instance of `Monad`, so we can use ordinary monad notation, including `do`-notation, for `TSIO` computations that do not affect the type state. In particular, the combinator `till` remains as in §2 above. Copying from `h2` and `h1` to `h3` neither opens nor closes handles and hence does not affect the type state. Therefore we write these computations using ordinary monad functions such as `>>=`, `>>`, and `liftM2`.

6.2 Assessment

The main example implemented with the `TSIO` library looks quite similar to the code written with the single-region `SIO` library of §2, especially if we overlook the slight difference in monad notations. Both implementations ensure that only open handles can be manipulated and all handles are eventually closed. The single-region `SIO` library provides these assurances at the cost of dynamically tracking open handles and denying the programmer the ability to explicitly close handles. All handles are closed only at the end of the whole computation. The nested-region `SIO` libraries of §3 and §4 improve the timeliness of deallocation. The `TSIO` library, in contrast, stresses manual resource management and does provide the operation to explicitly close a handle as soon as it is no longer needed. The library statically tracks the status of safe handles; a closed handle, although nominally available, cannot be used. For example, in the `test3` code above, omitting any of the `tshClose` operations or adding new ones leads to a type error.

Static tracking obviates dynamic tracking: whereas `runSIO` in §2–§4 performs non-trivial computations, `runTSIO` is operationally the identity function. However, the type-state approach has several drawbacks, compared with the region approach in §4.

First, the extensive type-class programming required makes the inferred types large and the error messages poor. The operation to copy a line from one handle to another

```
test4 h1 h2 = do line <- tshGetLine h1
                 tshPutStrLn h2 line
```

now has the inferred type

```
(Apply RemL (t1, t3) r, Nat0 t2,
 Apply RemL (t4, t3) r1) =>
TSHandle s t1 -> TSHandle s t4 ->
TSIO s (t2, t3) (t2, t3) ()
```

which says that `test4` takes two open handles. Compared to the type inferred for `test4` in §4, the `TSIO` type is less clear and betrays the implementation details of the `TSIO` library. The type inferred for `test3_internal` is yet larger, containing 8 type-class constraints.

Second, because each handle is tracked statically and individually, it is impossible to express (recursive) programs that open a statically unknown number of handles (for example, zipping together several input files, whose names and number are given by the user). The region approach does not have this problem, because it does not track the number of handles in each region.

Third, it is very hard to handle failures of IO operations. Whether these failures are managed as exceptions, what looks like a sequence of IO actions that moves from an initial state to a final state may in fact stop at any intermediate step. The resulting

proliferation of control-flow possibilities and the path-insensitivity of our type-state tracking makes it hopelessly unwieldy to recover after errors in a `TSIO` computation, especially when the lifetimes of resources are not nested. For example, in order to close handles properly in the following typical code, we need to write four different exception handlers or recovery routines explicitly—one for each subset of the two files involved.

```
do h1 <- tshOpen "file1" ReadMode
   l1 <- tshGetLine h1
   h2 <- tshOpen "file2" WriteMode
   tshPutStrLn h2 l1
   tshClose h1
   tshPutStrLn h2 l1
   tshClose h2
```

7. Related work

Launchbury and Peyton Jones [11] invented the `ST` monad and pioneered the use of rank-2 polymorphism to encapsulate memory references and operations on them to a single region. Launchbury and Sabry [12] generalized this idea to nested regions, but their representation of monads indexed by lineage provides limited region polymorphism. Our solution is cast in terms of file handles rather than memory references, and provides full region polymorphism. We also introduce handle duplication as a way to dynamically prolong the life of a handle.

Tofte and Talpin [18] invented regions for memory management and introduced a type-and-effect system to ensure safe access, that is, never dereferencing a dangling pointer. Like them, we support

creating arbitrarily many regions and allocating resources in any live region, be it the current region or one of its ancestors. Like them, we label the type of an allocated resource with the region holding it. A computation that allocates or accesses resources bears *effect annotations* in its type, which specify the labels of regions needed to execute the computation safely. Our annotations are typeclass constraints, which require no extension to Haskell and can be inferred. Also like Tofte and Talpin, we support region polymorphism: a computation may be polymorphic over the regions it uses.

Cyclone [5] and Fluet and Morrisett's monadic regions [4] extend the region calculus with region subtyping: a resource allocated in an older region is available as if it were located in any younger region. Our library in §4 can be regarded as another implementation of monadic regions, taking advantage of the type system to manage implicit evidence for region subtyping automatically. Handle duplication, described in §5, is a new extension of monadic regions.

One difference between regions of heap data and regions of file handles is that some heap data, such as pairs, may contain components that are themselves on a heap. Thus we need to track not only the lifetimes of containers but also of their components. Our region library can easily be extended for resources that contain other resources. Unlike implementations of monadic regions such as Fluet and Morrisett's and ours, Tofte and Talpin [18]'s effect system allows manipulating a value that contains dangling pointers, as long as the pointers are never dereferenced.

The safety guarantees of our approach rely on type eigenvariables, which are fresh names. Reasoning with fresh names is notoriously complex [13, 15]. We avoid many difficulties because we only ever assert to the type checker that two types are the same (must unify), never that two types are different (must not unify).

We review work related to type-state in §6 above.

8. Conclusions

We describe and implement two new ways to manage resources such as file handles in Haskell: monadic regions (with implicit subtyping) and type-state tracking. All of our libraries provide previously unavailable static guarantees, namely that a resource is used only when it is still allocated, and that all allocated resources are deallocated. Neither approach limits the number of resources or the order of their deallocation. We support general recursion (limited for type-state tracking), higher-order computations, and mutable state. Our work applies as is to other kinds of resources, such as database connections and device reservations. It seems compatible with Haskell′ as well. Our ongoing work is to try our approaches in larger projects (e.g., the portable SQL multi-database access library Takusen or web application servers) to test scalability.

Type-state tracking and monadic regions offer different tradeoffs between static and dynamic resource management.

- On one hand, type-state tracking is most precise and imposes no run-time overhead, because it is fully static and manual. We use a parameterized monad to make this form of linear typing available in Haskell today.
- On the other hand, monadic regions are much more concise, can express more computations, and impose negligible run-time overhead (less than that of the existing Haskell IO library). Our implementations of monadic regions provide region polymorphism and region subtyping, with clear inferred types and no

need for any type annotations. Our final implementation, based on monad transformers with implicit subtyping, is especially convenient to use because it frees the programmer from producing or passing any subtyping evidence and minimizes the notational overhead of regions.

In both approaches, deallocation is predictable, as well as timely to an extent depending on what the program does and how the programmer structures it. In particular, the new ability to duplicate handles across monadic regions allows the lifetime of handles to be controlled more dynamically. The type-state tracking approach, due to its intricate implementation, large inferred types, and poor error messages, remains merely a tantalizing proof of concept. We hope that type-level computations in GHC or other Haskell-like languages will make type-state tracking practical.

Throughout this work, we use monadic style to convert control-flow problems to data-flow ones, in particular to assign expressive types to effectful computations and reason with them. Writing programs thus in A-normal form is ungainly compared to the pure functional style of clausal definitions and pattern matching, so 'lazy IO' is deplorably popular despite its unsoundness. It would be more gainly to transfer our approach to an impure language like OCaml, but there we need an effect system in lieu of the monadic types. We thus need a language with an effect system.

Acknowledgments

We thank Greg Morrisett and Matthew Fluet for helpful discussions. Matthew Fluet suggested the example implemented in §5.

References

[1] Atkey, Robert. 2006. Parameterised notions of computation. In *MSFP 2006: Workshop on mathematically structured functional programming*, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society.

[2] Filinski, Andrzej. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

[3] Fluet, Matthew, Greg Morrisett, and Amal J. Ahmed. 2006. Linear regions are all you need. In *ESOP*, 7–21.

[4] Fluet, Matthew, and J. Gregory Morrisett. 2004. Monadic regions. In *ICFP '04: Proceedings of the ACM international conference on functional programming*. New York: ACM Press.

[5] Grossman, Dan, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 282–293.

[6] Igarashi, Atsushi, and Naoki Kobayashi. 2002. Resource usage analysis. In *POPL*, 331–342. New York: ACM Press.

[7] Kiselyov, Oleg. 2004. Heavy-weight implementation of region calculus. `http://okmij.org/ftp/Haskell/regions.html#heavy-weight`.

[8] ———. 2007. Type improvement constraint, local functional dependencies, and a type-level typecase. `http://okmij.org/ftp/Haskell/typecast.html`.

[9] Kiselyov, Oleg, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proc. ACM SIGPLAN workshop on Haskell*, 96–107.

[10] Kiselyov, Oleg, and Chung-chieh Shan. 2007. Lightweight static resources: Sexy types for embedded and systems programming. In *Draft proceedings of TFP 2007: 6th symposium on trends in functional programming*, ed. Marco T. Morazán and Henrik Nilsson. Tech. Rep. TR-SHU-CS-2007-04-1, Department of Mathematics and Computer Science, Seton Hall University.

[11] Launchbury, John, and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

[12] Launchbury, John, and Amr Sabry. 1997. Monadic State: Axiomatization and Type Safety. In *ICFP*, 227–238.

[13] Miller, Dale, and Alwen Tiu. 2005. A proof theory for generic judgments. *ACM Trans. Comput. Log.* 6(4):749–783.

[14] Moggi, Eugenio, and Amr Sabry. 2001. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming* 11(6):591–627.

[15] Pitts, Andrew M. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186(2):165–193.

[16] Strom, Robert E., and Daniel M. Yellin. 1993. Extending typestate checking using conditional liveness analysis. *IEEE*

Transactions on Software Engineering 19(5):478–485.

[17] Tofte, Mads, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2006. Programming with regions in the MLKit (revised for version 4.3.0). Tech. Rep., IT University of Copenhagen, Denmark.

[18] Tofte, Mads, and Jean-Pierre Talpin. 1997. Region-based memory management. *Inf. Comput.* 132(2):109–176.

[19] Walker, David, Karl Crary, and J. Gregory Morrisett. 2000. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.* 22(4):701–771.