Making Monads First-class with Template Haskell [*]

Pericles S. Kariotis   Adam M. Procter   William L. Harrison

Department of Computer Science
University of Missouri
Columbia, Missouri. USA.

psk1db@mizzou.edu   amp269@mizzou.edu   harrisonwl@missouri.edu

Abstract

Monads as an organizing principle for programming and semantics are notoriously difficult to grasp, yet they are a central and powerful abstraction in Haskell. This paper introduces a domain-specific language, MonadLab, that simplifies the construction of monads, and describes its implementation in Template Haskell. MonadLab makes monad construction truly first class, meaning that arcane theoretical issues with respect to monad transformers are completely hidden from the programmer. The motivation behind the design of MonadLab is to make monadic programming in Haskell simpler while providing a tool for non-Haskell experts that will assist them in understanding this powerful abstraction.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Staged Programming, Monads, Domain-specific Languages

1. Introduction

Praising monads to the Haskell community is like "preaching to the choir." Monads have become a central programming abstraction in Haskell with benefits including modularity and maintainability, effectful programming and mathematical precision. Monad lore (i.e., the principles underlying the use and construction of monads) is also infamous for its steep learning curve. This is more than a pedagogical concern. The steep learning curve of monad lore intimidates potential new Haskell users and impedes the growth of the Haskell community.

One means of improving this situation is to encapsulate monad construction within a language design. The approach explored in this article creates a DSL for automatically defining monads ("MonadLab") embedded in Template Haskell (TH). With MonadLab, the creation of monads is first-class: the monad lore underlying the creation of layered monads is encapsulated entirely by the language definition. **The contributions of this paper are:**

- *First-class Monads.* The MonadLab DSL insulates users from the theoretical details underlying monad construction. The implementation of MonadLab in Template Haskell handles *all* details of monad construction.
- *Simplicity & Completeness.* The Haskell Monad Transformer Library gets close to full automation of monad construction, but falls short in several ways. Template Haskell's metaprogramming can overcome these problems as it does in the MonadLab implementation. The result is a very direct formulation of monad transformers.
- *A Non-trivial Case Study for Template Haskell.* Template Haskell provides useful functionality for Haskell users, but its documentation is less than extensive. We believe that this paper will be of value to those trying to learn TH. This paper provides feedback data about its strengths and weaknesses.
- *A Tool for Learning Monadic Programming and Semantics.* Monad lore is difficult to learn, and MonadLab permits new or potential Haskell users to experiment easily with monads.

Why not just use the monad transformer library? Haskell's monad transformer library (MTL) does provide tools for constructing "layered" monads, however the MTL does not insulate users from details underlying monad transformer application. It is well-known that the order of monad transformer application can affect the behavior of the whole specification. While monad transformers always produce monads, there are certain cases in which the behavior of the extra "non-proper" morphisms defined on a monad depends on the order of application (Liang 1998) and the MTL requires users to grasp this somewhat arcane material. MonadLab incorporates this domain knowledge, thereby freeing users from such considerations.

The MTL is not well-suited to multiple applications of the same monad transformer because of its implementation within Haskell's type class system. There are scenarios that are best expressed via multiple applications of a single monad transformer. Monadic separation kernels (Harrison and Hook 2008), for example, use the distinct imperative operations arising from separate applications of the state monad transformer to delimit effects and enforce information flow security. For example, to create a monad with n distinct *put* and *get* operations using the MTL, one would first have to create a class with n distinctly named *put* and *get* methods. Alternatively, one could use a single application of the state monad transformer with a single global n-tuple state and define the n distinct methods as projections. Either strategy is ungainly and effectively obscures the modularity given by modular monadic semantics.

This paper proceeds as follows. Section 2 presents a familiar example—modular interpreters—expressed in terms of MonadLab. Section 3 gives a quick review of Template Haskell. Section 4 de-

[*] This research supported by NSF CAREER Award #00017806, NRL Contract 1302-08-015S, and the Gilliom Cybersecurity Endowment.

scribes the basic API provided by MonadLab. MonadLab is a DSL embedded in Template Haskell; this embedding is described in Section 5. Sections 6 and 7 present related work and conclusions, respectively.

2. Introducing MonadLab

This section introduces the syntax of MonadLab and an extended example of its use. The abstract syntax for monad declarations in MonadLab is:

```
<Decl>  ::= monad Tag = <Monad>
<Monad> ::= <Layer> { + <Layer> }*
<Layer> ::= <MT> ( <Type> ) Tag | Id | List
<MT>    ::= StateT | EnvT | WriteT | ErrorT | ContT
```

A top-level declaration, `<Decl>`, declares a new monad, named with a `Tag`. A `<Monad>` is a collection of `<Layer>`s. Each `<Layer>` represents the application of a particular monad transformer or base monad (i.e., either the identity or list monad). A `<Type>` refers to a Haskell type expression and `Tag` is used to distinguish the non-proper morphisms of this `<Layer>` from others in a `<Monad>` specification. A `Tag` is an identifier. Supported monad transformers include `StateT`, `EnvT`, `WriterT`, `ErrorT`, and `ContT`, along with the `List` and `Id` base monads. These are equivalent to the transformers described in (Liang et al. 1995) and in the MTL. MonadLab supports other monad transformers (in particular, resumption monad transformers (Papaspyrou 2001; Harrison and Hook 2008)) that will not be discussed in this paper.

Figure 1 shows some sample MonadLab programs. A monad program consists of one or more `<Monad>` declarations and related data declarations. These data declarations are simply Haskell data declarations. The MonadLab compiler, `mlab`, takes a file "*Monads*`.mlab`" and generates a Haskell file, "*Monads*`.hs`", implementing the monads. Template Haskell does not perform type checking while generating code (Sheard and Peyton Jones 2002), so the file produced by `mlab` will be type checked when the module is imported. MonadLab performs no type checking of its own, relying instead on the Haskell type system.

2.1 A sample application: monadic interpreter

Building on an interpreter for a simple functional language, we show that MonadLab makes it very easy to add new "layers" of semantic features to the language. We start with the familiar interpreter from (Wadler 1992), and extend it with mutable state, then with output, and finally with non-deterministic choice. Each successive extension requires only a small iterative change to the monad specification, and straightforward extensions to the interpreter code. Source code for all of the interpreters described here is available online (Kariotis 2008). Each interpreter presented here is composed of two parts: the monad specification (in file `InterpMonad.mlab`), and the interpreter code (in file `Interp.hs`).

The code for the core interpreter is shown in Figure 2. It differs only slightly from (Wadler 1992) in that it uses the environment monad to manage bound variables, rather than passing the environment as a parameter to `interp`. The MonadLab specification for this monad is given in Figure 1(a) — it creates a monad with a single environment (`EnvT`) layer, named `Env`, where the type of the environment is `Environ`. Also included in the monad specification (due to scoping requirements, because it is mutually recursive with `M`) is the data type `V`, representing values in the interpreter.

To build the interpreter, we first use the command-line tool `mlab` to generate Haskell code for the monad from the MonadLab specification. We can then use GHC to compile the interpreter as

(a) An immutable environment...
```
monad M       = EnvT(Environ) Env
data  V       = Wrong | Num Int | Fun (V -> M V)
type  Environ = [(String,V)]
```

(b) ...plus a mutable store...
```
monad M       = ... + StateT(Store) Sto
data  V       = ... | Ref Loc | Unit
...
type  Store   = [(Loc,V)]
type  Loc     = Int
```

(c) ...plus output...
```
monad M       = ... + WriterT(String) Out
...
```

(d) ...plus non-determinism.
```
monad M       = ... + List
...
```

Figure 1. MonadLab declarations for the interpreter monads.

usual, or load it into an interpreter such as Hugs or `ghci`:

```
$ mlab InterpMonad.mlab
Generated InterpMonad.hs
$ ghci Interp.hs
> test (Con 42)
42
> test (App (Lam "x" (Var "x")) (Con 2112))
2112
```

The standard environment monad functions `rdEnvM :: M Environ`, `inEnvM :: Environ -> M a -> M a`, and `runM :: M a -> Environ -> a` are all automatically generated by MonadLab and exported by `InterpMonad.hs`. Note that the functions' names are derived from the name of the monad (here `M`) and the names of its layers — if we had named the environment layer `Foo` and the monad `Bar`, then these functions would have been named `rdFooBar`, `inFooBar`, and `runBar`, respectively.

2.1.1 Mutable state

We now extend our lambda language with a mutable store. First we extend the language syntax with the new constructs `LetRef`, which creates a reference to a new location and binds it to a variable; `Deref`, which retrieves the value stored at a location; `PutRef`, which updates the value stored at a location; and `Seq`, which executes two (presumably effectful) operations in sequence. The extended syntax is as follows:

```
data Term = ... | LetRef Name Term Term
          | Deref Term | PutRef Term Term
          | Seq Term Term
```

We then add to the monad specification a state layer named `Sto` over stores of type `Store`, which is simply a mapping from locations to values. We must also extend `V` to include references and a "unit" type, resulting in the new monad specification given in Figure 1(b). Finally, we modify the interpreter code by extending `interp` and `test` as shown in Figure 3. The revised interpreter makes use of the new monad operations `getStoM :: M Sto`, which retrieves the current state, and `putStoM :: Sto -> M ()`, which updates the state.

```
module Interp where

-- The interpreter monad
-- (Generated by MonadLab)
import InterpMonad

-- Language syntax
type Name = String
data Term = Var Name | Con Int | Add Term Term
          | Lam Name Term | App Term Term

mkfun :: Name -> M V -> M V
mkfun x phi =
   rdEnvM >>= \ e ->
     return (Fun $ \ arg -> inEnvM ((x,arg):e) phi)

appEnv :: Name -> M V
appEnv x = rdEnvM >>= \ e ->
           case lookup x e of
              Nothing  -> return Wrong
              (Just v) -> return v

apply :: V -> V -> M V
apply (Fun k) a = k a
apply _ _       = return Wrong

-- The interpreter
add :: V -> V -> M V
add (Num i) (Num j) = return (Num (i+j))
add _ _             = return Wrong

interp :: Term -> M V
interp (Var x)   = appEnv x
interp (Con i)   = return (Num i)
interp (Add u v) = interp u >>= \ a ->
                   interp v >>= \ b ->
                      add a b
interp (Lam x v) = mkfun x (interp v)
interp (App t u) = interp t >>= \ f ->
                   interp u >>= \ a ->
                      apply f a

test :: Term -> V
test t = runM (interp t) initEnv
         where initEnv = []
```

Figure 2. The core interpreter.

There are several points worth noting here: first, all of the newly introduced state operations are automatically generated, yet the existing environment operations can be used just as before — the programmer does not need to know how to "lift" them through the state transformer, since MonadLab takes care of this automatically. Second, there is no need to change any of the existing code for `interp` that does not deal with state — writing our interpreter in this way achieves a high degree of semantic modularity. Finally, note that even though we are beginning to build up a fairly complex monad, the `runM` function is automatically generated, and has a reasonably obvious type.

2.1.2 Output and nondeterminism

Extending the language with output is even simpler. All we have to do is add a `WriterT` layer to the monad (see Figure 1(c)), extend the language syntax and interpreter with a `Print` command, add a new value type `Unit` to `V`, and modify `test` to return the output as part of its result type:

```
getLoc :: Loc -> M V
getLoc l = getStoM >>= \s ->
           case lookup l s of
              Nothing  -> return Wrong
              (Just v) -> return v

putLoc :: Loc -> V -> M ()
putLoc l v = getStoM >>= \s -> putStoM $ (l,v):s

interp :: Term -> M V
...
interp (LetRef x v t) = interp v >>= \val ->
                          inExtEnvRef x val
                           (interp t)
interp (Deref t)      = interp t >>= \ref ->
                        case ref of
                          (Ref l) -> getLoc l
                          _       -> return Wrong
interp (PutRef r v)   = interp r >>= \ref ->
                        case ref of
                          (Ref l) -> interp v >>=
                                     putLoc l >>
                                       return Unit
                          _       -> return Wrong
interp (Seq t1 t2)    = interp t1 >> interp t2

test :: Term -> V
test t = runM (interp t) initEnv initSto
         where initEnv = []
               initSto = []
```

Figure 3. The core interpreter, extended with mutable state operations. (The definition of `inExtEnvRef` is omitted.)

```
data Term = ... | Print Term
...
interp (Print t) = interp t >>= \ v ->
                   tellOutM (show v) >>
                      return Unit
test :: Term -> (V,String)
test t =
   runM (listenOutM $ interp t) initEnv initSto
      where initEnv = []
            initSto = []
```

Once again, the writer monad operations `tellOutM :: String -> M ()` and `listenOutM :: M a -> M (a,String)` are automatically generated by MonadLab.

Finally, we can extend our language with non-determinism by changing our monad to a `List` as shown in Figure 1(d), and extending the language with the non-deterministic choice operator `Amb`. (Note, with respect to `test`, that its type changes, but the text of its body does not):

```
data Term = ... | Amb Term Term
...
interp (Amb t u) = mergeM [interp t,interp u]
test :: Term -> [(V,String)]
test t =
   runM (listenOutM $ interp t) initEnv initSto
      where initEnv = []
            initSto = []
```

3. An Overview of Template Haskell

Template Haskell is an extension to Haskell that allows parts of a program's source code to be statically computed, rather than being written entirely by hand by the programmer. There are three major parts of Template Haskell (TH): a set of data types for the representation of Haskell code, the quotation monad `Q`, and the quasi-quotation and splice operators. For a more detailed description of Template Haskell, please refer to the bibliography. In particular, there are a number of Template Haskell experience reports (Lynagh 2003a,b; Seefried et al. 2004) and a useful reference guide (Sheard and Peyton Jones 2003).

3.1 Code Representation Types

To facilitate the manipulation of code, TH provides a set of algebraic data types to represent Haskell code. The primary types are `Exp` for expressions, `Pat` for patterns, `Type` for types, and `Dec` for declarations. Every code construct has a corresponding constructor in one of these types. For example, the variable expression `x` is represented as `VarE (mkName "x")`. TH uses the `Name` data type to represent identifier names, and the constructor `VarE` creates variable expressions from variable names.

3.2 The Quotation Monad

These data types are sufficient to represent the full range of Haskell code. There are often times during code construction, however, when it becomes necessary to generate a fresh variable name for use in a pattern or expression. TH provides the quotation monad `Q` for this purpose. As a result, Haskell expressions are most often represented not as values of type `Exp`, but as values of the monadic type `Q Exp`. Patterns, types, and declarations are similarly represented. For convenience, TH provides lowercase analogs of all the code constructor functions for use in the quotation monad. For example, `varE :: Name -> Q Exp` is used to construct variable expressions. In actuality, however, `varE` is simply a convenient shorthand for `return . VarE` where `return` is the unit of the `Q` monad.

In addition to providing for fresh name generation, the `Q` monad provides access to several of the compiler's internal symbol tables. The function `reify :: Name -> Q Info` can be used to query information about a particular variable or data type. We will not elaborate further on this feature, except to note that reification can only occur at compile-time, as the symbol tables are not available at run-time.

3.3 Quasi-quote Brackets

If explicit calls to the code constructors or their monadic counterparts were the only means of creating code representations, programming in TH would be very tedious. For this reason, TH provides the quasi-quotation syntax. Quasi-quote brackets automatically convert concrete Haskell syntax to its representation in the TH data types. For example, to represent the expression `\ x y -> x + y`, we surround it with quasi-quote brackets as `[|\ x y -> x + y|]`. This is considerably more convenient than the verbose construction:

```
lamE [varP (mkName "x"), varP (mkName "y")]
(Just (infixE (varE (mkName "x")))
(varE (mkName "+"))
(Just (varE (mkName "y"))))
```

This notation also extends to patterns, types, and declarations, with only a minor adjustment in syntax: `[p|...|]` brackets are used for patterns, `[t|...|]` brackets are used for types, and `[d|...|]` brackets are used for declarations. It should be noted, however, that quasi-quotes are simply a notational convenience, and can be combined with the constructors as needed.

```
data Layer = List
           | StateT StateName (Q Type)
           | EnvT EnvName (Q Type)
           | ErrorT ErrorName (Q Type)
           | WriterT WriterName (Q Type)
           | ContT (Q Type)
```

Figure 4. The Layer data type.

3.4 The Splice Operator

The last major component of TH is the use of the `$` (splice) operator, which indicates both when code should be computed and where it should be inserted. A few examples will demonstrate its properties. Assume the following declarations:

```
computeFoo :: Int -> Q Exp
computeFoo = ...

foo = $(computeFoo 0)
```

The top-level use of `$` indicates that its operand should be evaluated at compile-time and that the resulting code (of type `Q Exp`) should be spliced into the source at the place of the call. Notice the absence of any whitespace after the `$` operator and its operand. This is required to distinguish the splice operator from the use of `$` as an ordinary Haskell operator.

The following example shows that the splice operator can also be used to insert declarations:

```
fooDecl :: Q [Dec]
fooDecl = ...
$fooDecl
```

At compile-time, `fooDecl` will be evaluated, and the resulting set of declarations will be inserted into the source in place of `$fooDecl`. It should be noted that only values of type `Q [Dec]` can be spliced into source, and not values of type `Q Dec`.

The `$` operator can also be applied inside quasi-quote brackets, as in the following example:

```
foo :: Int -> Q Exp
foo x = [| \y -> $(bar x) |]
bar :: Int -> Q Exp
bar x = ...
```

Since the `$` operator is not called at the top-level of the source, its operand will not necessarily be evaluated at compile-time. Instead, it will be evaluated when the enclosing quasi-quotation expression is evaluated, and the resulting code will be inserted inside the quasi-quotes.

4. Embedding MonadLab in Template Haskell

The MonadLab DSL is built on top of a library written in Template Haskell. The `mlab` compiler for monad specifications thus proceeds in two steps. First, the specification is translated into a call to the library function `mkMonad`. Next, the call is evaluated and the resulting Haskell declarations are written to a source file. The compiler, therefore, is simply a convenient front-end for the library.

We begin with a simple example to illustrate the direct use of the library. After importing the module `MonadLab.MonadLab`, we create the identity monad with the following meta-declaration:

```
$(mkMonad "Id" [])
```

The call to the function `mkMonad` will compute the Haskell declarations for the identity monad. The function `mkMonad` has type

```
List     : merge
StateT   : put, get
EnvT     : rd, in
ErrorT   : throw, catch
WriterT  : tell, listen, pass
ContT    : callcc
```

Figure 5. Non-proper morphisms for each layer.

`MonadName -> [Layer] -> Q [Dec]` and is the primary means of creating monads in MonadLab. The first argument is a `String` value that names the type constructor of the monad being created. In the above case, since the literal `"Id"` is passed, a data declaration like following will be generated for the monad:

```
data Id a = Id a
```

The second argument to `mkMonad` has type `[Layer]` and specifies the operations (non-proper morphisms) we would like to include in the monad in addition to the return and bind. In the case of the identity monad, no additional operations should be included, so the list is empty. A complete description of this argument and the `Layer` data type will be given shortly.

The result of the call to `mkMonad` has type `Q [Dec]`, a computation in the quotation monad of a list of Haskell declarations. These include the type constructor declaration, the instance declaration for the `Monad` constructor class, the function declarations for all non-proper morphisms, and the run function. See Section 3 for more details.

Figure 4 shows the definition of the `Layer` type. Each constructor corresponds to a well-known monad transformer or, in the case of the `List` constructor, to the list base monad. (The `List` constructor does not correspond to the `ListT` monad transformer of the MTL.) To add operations to a monad, we simply include the corresponding `Layer`s. The function `mkMonad` will then appropriately compose the corresponding monad transformers to create the new monad. For example, suppose we want to create a monad with a single mutable state of type `Int`. To do so, we write:

```
$(mkMonad "St" [ StateT "State" [t| Int |] ])
```

The resulting `St` monad declarations will include the non-proper morphisms for retrieving and updating the state, respectively:

```
getStateSt :: St Int
putStateSt :: Int -> St ()
```

This example illustrates two points. First, the `StateT` constructor is used to add mutable state components to a monad. Second, the non-proper morphisms are subject to a common naming convention. The operations associated with the state monad transformer are "get" and "put." To resolve the case in which there is more than one mutable state, each state is given a name that is appended to the operation name. Finally, the name of the monad is appended to the name of each operation to distinguish similar operations in different monads. To more clearly demonstrate this convention, let's create a monad with two mutable states:

```
$(mkMonad "St'"
     [ StateT "State1" [t| Int  |] ,
       StateT "State2" [t| Bool |] ])
```

The non-proper morphisms of `St'` are:

```
getState1St' :: St' Int
putState1St' :: Int -> St' ()
getState2St' :: St' Bool
putState2St' :: Bool -> St' ()
```

```
type Monad = ( MonadTypeCon
             , ReturnExpQ
             , BindExpQ
             , [LayerNPM]
             , LiftExpQ
             )

type MonadTransformer = Monad -> Monad

type MonadTypeCon = Q Type -> Q Type
type ReturnExpQ = Q Exp
type BindExpQ = Q Exp
type NonProperMorphismExpQ = Q Exp
type LayerNPM = (Layer
               ,[NonProperMorphismExpQ])
type LiftExpQ = Q Exp
```

Figure 6. The Monad and MonadTransformer types

The non-proper morphisms added by each `Layer` constructor are given in Figure 5. The semantics of each are well known from the MTL and from Liang et al. (Liang 1998; Liang et al. 1995). As another example, let's create the following monad that includes a `[(String, Int)]` environment, exception handling, and non-deterministic computation:

```
$(mkMonad "M" [ EnvT "Env" [t| [(String, Int)] |],
                ErrorT "Error" [t| String |] ,
                List ])
```

The non-proper morphisms generated for `M` are:

```
rdEnvM      :: M [(String, Int)]
inEnvM      :: [(String, Int)] -> M a -> M a
throwErrorM :: String -> M a
catchErrorM :: M a -> (String -> M a) -> M a
mergeM      :: [M a] -> M a
```

Notice that no name is passed to the `List` constructor, and the merge operation is appended only by the name of the monad. Because `List` corresponds to the list base monad (to which the transformers are applied), it cannot be included in the `Layer` list more than once. Hence, no confusion results from leaving this constructor nameless. The `ContT` constructor can also be included only once and, therefore, also requires no name argument.

4.1 The run Function

In addition to generating the declarations for a monad and its operations, `mkMonad` also produces a declaration for running computations in the monad. For example, the `St` monad is accompanied by the function `runSt :: St a -> Int -> a`, which takes a computation in the `St` monad and an initial value for its state and then runs the computation on the initial value. The monad `St'`, on the other hand, is accompanied by the function `runSt' :: St' a -> Int -> Bool -> a`, which runs a computation in the `St'` monad on the two passed initial state values. Last, the monad `M` is accompanied by the function:

```
runM :: M a -> [(String, Int)] -> [a]
```

which runs a computation in the `M` monad on the passed initial environment value. The `ErrorT Layer` requires no initial value, and since a `List Layer` is included, the computation produces a list of results.

Notice that there are no error, state, or environment components in the return types of these functions. If a computation throws an unhandled exception, the `run` function will simply call the Haskell

```
newtype Identity a = Identity {runIdentity :: a}

instance Monad Identity where
     return a = Identity a
     m >>= f = f (runIdentity m)
```

Figure 7. The identity monad expressed in the MTL

`error` function. It is the user's responsibility, therefore, to catch any exceptions that may occur. In this way, we avoid the need for a composite return type to reflect exceptions. Furthermore, only the computed value of the monad is returned. State, writer, and environment components must be explicitly returned in the computation if they are to be returned by the `run` function.

In general, a monad's `run` function takes a computation in the monad and a set of initial values and returns the result of running the computation on the passed initial values. The layers that require initial values are the `StateT`, `EnvT`, and `ContT` variants. These values are passed to the monad's run function in the order their layers appear in the `Layer` list, with the exception that the initial continuation is always passed last. For example, the declaration:

```
$(mkMonad "M'" [ StateT "State1" [t| Int |],
                 ErrorT "Error" [t| String |],
                 EnvT "Env" [t| Bool |],
                 StateT "State2" [t| Rational |],
                 WriterT "Writer" [t| String |]
               ])
```

will generate a `runM'` function of type:

```
   M' a -> Int -> Bool -> Rational -> a.
```

If the layer `ContT [t| String |]` is added, the `runM'` function will have type:

```
   M' a ->
       Int ->
           Bool ->
               Rational ->
                   (a -> String) -> String,
```

where an intermediate result, of type `a`, will be passed to the initial continuation to produce the final result. If a list layer were then further added, the `runM'` function would have type:

```
   M' a ->
       Int ->
           Bool ->
               Rational ->
                   (a -> String) -> [String],
```

where the initial continuation would be mapped over a list of intermediate results (of type `[a]`) to produce the final list of results.

4.2 The Semantics of `mkMonad`

As previously mentioned, `mkMonad` performs its construction by successively applying the monad transformers corresponding to the list of `Layer` values. If the `List` value is included in the layer list, the list monad will serve as the base monad for the composition. Otherwise, the base monad will be assumed to be the identity monad. With two exceptions, the transformers will be applied in order from right to left as their layers appear in the list. First, because of difficulties in lifting `inEnv` and `catchError` through the continuation transformer, if a `ContT` layer is included in the list, the continuation transformer will be the first transformer applied. This explains why the initial continuation is always the last argument of a monad's `run` function. Also, since at most one `ContT` layer is allowed, no confusion results about which continuation transformer is first applied.

```
identityMonad :: Monad
identityMonad = (identityTypeCon,
                 identityReturn,
                 identityBind,
                 [],
                 identityBaseLift)

identityTypeCon :: MonadTypeCon
identityTypeCon = \t -> t

identityReturn :: ReturnExpQ
identityReturn = [| \v -> v |]

identityBind :: BindExpQ
identityBind = [| \m -> \f -> f m |]

identityBaseLift :: LiftExpQ
identityBaseLift = [| id |]
```

Figure 8. The identity monad in MonadLab

Second, any error transformers will be applied only after all other transformers and in order from right to left as their layers appear in the list. The reason for this is that the error transformer does not commute with either the state or writer transformers. In particular, the behavior of the `catchError` non-proper morphism is dependent upon the order of transformer application. For simplicity, however, we desire that the behavior of a monad's non-proper morphisms be consistent across rearrangements of its layer list (i.e., that the + operator in the MonadLab DSL is commutative). By fixing the order of application of the error transformer with respect to the state and writer transformers, we achieve this effect. This, of course, comes at the price of the loss of control over the order of error transformer application.

A few examples will illustrate the construction. The `St`, `St'`, and `M` monads declared above have the following equivalent constructions in the MTL:

```
type St  = StateT Int Identity
type St' = StateT Int (StateT Bool Identity)
type M   = EnvT [(String, Int)] (ErrorT String [])
```

Lastly, we consider an example including the `ContT` layer:

```
   $(mkMonad "M'" [ ContT [t| Int |],
                    StateT "State" [t| Int|] ])
```

has the equivalent declaration:

```
      type M' = StateT Int (ContT Int Identity)
```

Notice that the continuation transformer is applied before the state transformer, despite the reverse ordering of the list.

5. Implementation

5.1 The `Monad` Type

The semantics of `mkMonad` given previously reflect its implementation: monads are created by successively applying monad transformers to a base monad. Once appropriate representations for monads and monad transformers have been formulated, each base monad and monad transformer can be constructed, and the semantics can be easily implemented.

Figure 6 shows the declarations of both monad and monad transformer types, along with any necessary supporting declarations. The type `Monad` is composed of a type constructor, a return, a bind, and a list of non-proper morphisms. However, because the only monads we will consider are constructed using transformers, each monad has an associated lifting for values in its base monad. We augment our `Monad` declaration to include this component and will later see its necessity.

The types of the components comprising the `Monad` tuple require only a brief explanation. The monad's type constructor is represented as a function of type `Q Type -> Q Type`. This corresponds to the notion in the MTL of a monad as a function from types to types. The components representing the return, the bind, and the base monad lifting all have type `Q Exp` and are direct representations of the code for the respective functions. Notice that the `Monad` type is not a monad in the usual sense. Instead, it is a data structure that contains abstract syntax for the declaration of a monad.

Finally, to distinguish the monad's various non-proper morphisms, the code for these operations is stored as a list of values of type `(Layer, [NonProperMorphismExpQ])`. For each `Layer` variant, code for the non-proper morphisms is stored in the order in which they are given in Figure 5. For example, the `get` and `put` operations of an arbitrary `StateT` layer would be stored as:

```
(StateT name type, [get_code, put_code])
```

5.2 The Identity `Monad`

Figure 8 lists the definition of the identity monad in MonadLab, and Figure 7 provides a reference listing of the identity monad in the MTL. For the most part, a one-to-one correspondence exists between the two. Note in the new definition, however, that the type constructor is defined simply as the identity function and there are no references to the `Identity` data constructor or `runIdentity` selector function. As we will see, this is also the case in the new defintions of the monad transformers. Therefore, no new types are introduced as a result of `Monad` construction.

After the transformers associated with each `Layer` have been applied, the `MonadTypeCon` of the resulting layered `Monad` is used to generate a type expression $\varphi(\mathtt{a})$ expressing the structure of the monad. The type declaration that is finally emitted will be of the form:

newtype M a = M $\varphi(\mathtt{a})$

where `M` is the name of the monad. We call $\varphi(\mathtt{a})$ the "inner" type of the monad. `M a`, on the other hand, is the "outer" type. Strictly speaking, this "wrapping" is only necessary so that the monad may be declared an instance of the `Monad` type class — however, it also serves to "hide" the structure of φ. The "(a)" in "$\varphi(\mathtt{a})$" signifies that type variable `a` may occur in type expression φ.

Figure 9 shows the function `createMonad` which uses the `Monad` and `MonadTransformer` types to implement the semantics of `mkMonad`. Without the need to describe the supporting functions, it can be seen that first, the `ContT` layer (if it exists) is moved to the right-most position in the layer list, and the appropriate base monad is selected. The list is then traversed from right to left, successively applying the corresponding transformers. The functions `stateT`, `errorT`, etc., have type `Layer -> MonadTransformer` and are analogous to the transformers `StateT`, `ErrorT`, etc. in the MTL. A detailed examination of their implementation is given in the following section.

5.3 The State Monad Transformer

Figure 10 shows a partial listing of the definition of the state monad transformer, along with a similar listing of the state monad transformer in the MTL. The definition in the MTL states that given a type `S` and a monad `M`, the type `StateT S M` is a monad whose operations are defined in terms of the operations of `M`. Similarly, the definition in MonadLab states that given a `StateT` constructed `Layer L` and a `Monad M`, the expression `stateT L M` is a `Monad` whose operations are defined in terms of the operations of `M`. Again, modulo the presence of data constructors and destructors, there is largely a one-to-one correspondence between the definition in the MTL and the new definition. This is naturally the case, as the declarations in the MTL provide a template for the construction of their code.

```
createMonad :: [Layer] -> Monad
createMonad ls = foldr addLayer baseMonad ls'
 where ls'       = rightAlignCont ls
       baseMonad =
           if any (hasLayerType ListLayerType) ls'
             then listMonad
             else identityMonad

addLayer :: Layer -> Monad -> Monad
addLayer l@(StateT name t)  m = stateT l m
addLayer l@(EnvT name t)    m = envT l m
addLayer l@(ErrorT name t)  m = errorT l m
addLayer l@(List)           m = m
addLayer l@(WriterT name t) m = writerT l m
addLayer l@(ContT t)        m = contT l m
```

Figure 9. Semantics implementation

The return operation provides an example of this last point. Given the code for a monad `M` and a type `S`, we would like to construct the code for the return operation of the monad `StateT S M`. The function `stateTransReturn` shows how this can be done, where the code for the inner monad `M` is passed as the `Monad` argument, and the code for type `S` is unnecessary. First, the code for the return operation of `m` is extracted. Next, the code for the return operation of `StateT S M` is constructed as:

```
\ v s -> <insert return code for m here> (s, v)
```

All operations in the inner monad are lifted in a similar manner as the return. Notice that the functions `stateTransLiftThrow` and `stateTransLiftCatch` are parameterized over the non-proper morphisms that they lift. If the additional arguments were not provided, these functions would have no way to determine which `throw` or which `catch` of the `Monad m` they should lift. Notice also that `stateTransLiftCatch` is not parameterized over an inner monad. Since only the inner `catch` operation is required for the lifting (and not the inner return or bind), the `Monad` parameter is omitted. In general, the functions that lift the non-proper morphisms are parameterized only over the code they require to perform the lifting.

The lifting of the `merge` non-proper morphism, shown at the bottom of Figure 10, requires additional explanation. The definition of the lifted merge operation is given by the expression `join . lift`, where `join` is the join operation in the `StateT s m` monad, and `lift` is the function lifting values in the list monad (the base monad) to values in the `StateT s m` monad. The lifting function is constructed as the composition of the lift of the state transformer with the base monad lifting of the inner monad. This explains why it is necessary to include the base monad lifting function in the `Monad` type: it is required to construct the merge operation.

A few scattered points remain. First, since all monad operations are defined as lambda expressions, they must be applied in prefix notation. In particular, the bind operation is applied as a prefix operator, not an infix operator. Second, note the use of `stateTransLayerNPM` and `stateTransLiftLayerNPMs` in the definition of `stateT`. The function `stateTransLayerNPM`

Monad Transformer Library Implementation

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

instance MonadTrans (StateT s) where
    lift m = StateT $ \s -> m >>= \a -> return (a, s)

instance (Monad m) => Monad (StateT s m) where
    return a = StateT $ \s -> return (a, s)
    m >>= k  = StateT $ \s -> runStateT m s >>= \ (a, s') -> runStateT (k a) s'

instance (Monad m) => MonadState s (StateT s m) where
    get   = StateT $ \s -> return (s, s)
    put s = StateT $ \_ -> return ((), s)

instance (MonadError e m) => MonadError (StateT s m) where
    throwError = lift . throwError
    m `catchError` h = StateT $ \s -> runStateT m s `catchError` \e->runStateT (h e) s
```

MonadLab Implementation

```
stateT :: Layer -> MonadTrans
stateQMonadTrans l@(StateT name t) =
    \m -> ( stateTransTypeCon t m
          , stateTransReturn m
          , stateTransBind m
          , stateTransLayerNPM l m : stateTransLiftLayerNPMs m --Wrap the npms as LayerNPM values
          , [| $(stateTransLift m) . $(getBaseLift m) |]
          )

stateTransTypeCon :: TypeQ -> Monad -> MonadTypeCon
stateTransTypeCon s m = \t -> appT (appT arrowT s) (tc (appT (appT (tupleT 2) s) t))
                where tc = getTypeCon m

stateTransLift :: Monad -> LiftExpQ
stateTransLift m = [| \m -> \s -> $bind m (\v -> $return (s, v)) |]
                where return = getReturn m
                      bind   = getBind m

stateTransReturn :: Monad -> ReturnExpQ
stateTransReturn m = [| \v -> \s -> $return (s, v) |]
                where return = getReturn m

stateTransBind :: Monad -> BindExpQ
stateTransBind m = [| \m -> \f -> \s0 -> $bind (m s0) (\(s1, v) -> f v s1) |]
                where bind = getBind m

stateTransGet :: Monad -> NonProperMorphismExpQ
stateTransGet m = [| \s -> $return (s,s) |]
                where return = getReturn m

stateTransPut :: Monad -> NonProperMorphismExpQ
stateTransPut m = [| \s -> \_ -> $return (s,()) |]
                where return = getReturn m

stateTransLiftThrow :: Monad -> NonProperMorphismExpQ -> NonProperMorphismExpQ
stateTransLiftThrow m throw = [|  $(stateTransLift m) . $throw |]

stateTransLiftCatch :: NonProperMorphismExpQ -> NonProperMorphismExpQ
stateTransLiftCatch catch = [| \x -> \h -> \s -> $catch (x s) (\e -> h e s) |]

stateTransLiftMerge :: Monad -> NonProperMorphismExpQ
stateTransLiftMerge m = [| $newJoin . $newBaseLift |]
                where newJoin     = [| \x -> $(stateTransBind m) x (\a -> a) |]
                      newBaseLift = [| $(stateTransLift m) . $(getBaseLift m) |]
```

Figure 10. The state monad transformer represented in the MTL and MonadLab.

```
createExtractExp :: [Layer] -> ExpQ
createExtractExp = foldr augmentExtractExp [| id |]

augmentExtractExp :: Layer -> ExpQ -> ExpQ
augmentExtractExp List          e  = e
augmentExtractExp (StateT _ _) e  = [| fst . $e |]
augmentExtractExp (EnvT _ _)    e  = e
augmentExtractExp (ErrorT n _) e  = [| $h . $e |]
  where h =
       [| \v ->
            case v of
                 Left  _ -> error msg
                  where msg = (n ++ " exception")
                 Right r -> r
       |]
augmentExtractExp (WriterT _ _) e  = [| fst . $e |]
augmentExtractExp (ContT _)      e  = e
```

Figure 11. Extraction function construction.

wraps the get and put operations as a LayerNPM value. The function stateTransLiftLayerNPMs sequentially lifts all the non-proper morphisms of the inner monad using functions like stateTransLiftThrow and stateTransLiftCatch in Figure 10. Last, note the verbosity of the code in the definition of the type constructor. Splicing into type brackets [t|...|] is not permitted in the current version of TH, so the type expression must be written using the code constructor functions.

5.4 Implementation of run

The monadic types created through the use of the given transformers will always be functions from a set of initial values (states, environments, and continuations) to a term that contains the computed value. For example, the underlying type of a monad with a single state of type Int is the polymorphic type Int -> (a, Int), a function from an initial state to a pair containing the result value. If an environment of type Bool were added, the underlying type would become Bool -> Int -> (a, Int), a function requiring an initial environment and an intial state. Finally, if String exception handling were added, the type would become Bool -> Int -> (Either String a, Int). In each case, to run computations in the monad, we have only to apply the monadic function value to a set of initial values and then extract the computed value from the resulting term.

There are four cases that must be considered to implement this idea. The most simple is the case in which the base monad is the identity monad and the continuation transformer has not been applied in the monad's construction. There are then three transformers that complicate the term from which the computed value must be extracted: the state and writer transformers, which embed the result in a pair, and the error transformer, which embeds the result in a sum (an Either constructed type). But because the monad is constructed from the composition of transformers, the computed value will be embedded in a term whose type is the composition of pairs and sums. The extraction function is then constructed as the composition of projections out of pair and sum types. In this way, the extraction function successively deconstructs the pairs and union values containing the computed value. Figure 11 lists the createExtractExp function, which constructs the extraction function. Note that if an exception has been thrown in the monad, projection out of the union type will fail, and the error function is called.

The second case that must be considered is when the list monad serves as the base, and the continuation transformer has again not been included in the construction. In this case, the result of running a monadic computation is not a single computed value, but a list of values. More specifically, the result of applying a monadic computation to a set of initial values is a list of result terms, each containing a computed value. To extract the values, the extraction function from above is mapped across this term list. For example, the underlying type of the monad constructed by applying the state transformer to the list monad is the polymorphic type S -> [(a, S)], for some state type S. To run a computation in this monad, an initial state is first passed to the computation. The result of this application will be a list of terms of type (a, S), each containing a computed value. The extraction function fst is then mapped across the list to construct the list of values.

The third case occurs when the identity monad serves as the base, and a continuation transformer has been applied in the construction. In this case, monadic values are functions from a set of initial values, including an initial continuation, to an answer value. And since the the continuation transformer is always the first to be applied, the initial continuation will always be the last argument passed to the monadic value. We will see, however, that the initial continuation must be slightly augmented before it can be passed.

An example will demonstrate. Suppose a monad is constructed that includes a mutable state and continuations. The underlying type of the monad is S -> ((a, S) -> Ans) -> Ans, where S is the type of the state and Ans is the answer type of continuations. Notice that the type of the continuation argument is (a, S) -> Ans. However, the type of the initial continuation is a -> Ans. That is, the intermediate result of type a must be extracted from a term of type (a, S) before it can be passed to the initial continuation. Therefore, the continuation argument should be constructed as the composition of the extraction function (fst) with the initial continuation. The procedure naturally generalizes to cases with additional transformers.

The last case occurs when the list monad serves as the base, and a continuation transformer has been applied in the construction. This case is very similar to the previous one. Let's take a look at the previous example, but this time use the list monad, not the identity monad, as the base. The type of this monad is S -> ((a, S) -> [Ans]) -> [Ans], where the S and Ans types are as before. Notice that this monad's type is almost identical to the previous one's, but with a list of answer values computed instead. In particular, the type of the continuation argument is (a, S) -> [Ans]. But the initial continuation should still have the type a -> Ans. The continuation argument is then constructed as before, but with the computed answer value wrapped as a list of single element. That is, the continuation argument is constructed as (\x -> [x]) . k . f, where k is the initial continuation, and f is the extraction function.

5.5 Declaration Construction

After the monad has been fully constructed, all that remains is to generate a sequence of Haskell declarations from the resulting Monad value. To more clearly demonstrate this process, we will slightly abuse some of the TH syntax. In particular, we will make rather liberal use of the splice and bracket operators to avoid complicated expressions in the TH data constructors. The code we present, therefore, is not always valid TH, but a template from which the actual code can be easily written.

Figure 12 lists the functions used to construct several of the declarations for the monad. These functions take a monad name and a computed Monad tuple and then generate the various declarations. Although the list is not exhaustive, these functions clearly demonstrate the general construction technique.

As mentioned in Section 5.2, the monad's type declaration is constructed by wrapping the "inner" monadic type constructor phi

```
createTypeDecl :: MonadName -> Monad -> DecQ
createTypeDecl n m = [d| newtype $n $tvar = $n { $deN :: $(phi tvar) } |]
   where phi  = getTypeCon m
         tvar = [t| a |]
         deN  = "de" ++ n

createBindDecl :: MonadName -> Monad -> DecQ
createBindDecl n m = [d| (>>=) = \x -> \f -> $n ($bind_code ($deN x) ($deN . f) |]
   where bind_code = getBind m
         deN       = "de" ++ n

createReturnDecl :: MonadName -> Monad -> DecQ
createReturnDecl n m = [d| return = $n . $ret_code |]
   where ret_code = getReturn m
```

Figure 12. Declaration construction

inside a `newtype` declaration. The monad name is used as the identifier for both the type and data constructors of this new "outer" monadic type. In addition, a field selector function is declared and identified by prepending `de` to the monad name.

5.6 The Recursion Problem

Consider the following example in which the type of the monad's environment is defined in terms of the monad itself, assuming prior declarations for types `Key` and `Value`:

```
$(mkMonad "M" [EnvT "Env" [t| [(Key, M Value)] |]])
```

Unfortunately, recursive declarations like this are not permitted. TH will attempt to resolve all type constructor names in the quasi-quoted environment type before computing the declarations for the monad. The type constructor `M`, therefore, will not be in scope when TH attempts to interpret the environment type.

The solution is to refrain from taking advantage of the quasi-quotation mechanism for `Layer` construction. MonadLab provides the function `envTRec :: String -> String -> Layer` that allows a `String` representation of the environment type to be used in place of the `Q Type` representation in the `EnvT` constructor. The above example could be correctly written

```
$(mkMonad "M" [ envTRec "Env" "[(Key, M Value)]" ])
```

In this new version, `envTRec` will parse the type string to create a value of type `Q Type` that will be passed to the `Env` constructor. The names of any type constructors, however, will not be resolved until the monad declarations are processed, as is necessary.

5.7 Mutually Recursive Declarations

A second problem occurs when the monad declaration is a member of a set of mutually recursive declarations. This problem is a result of the way TH processes declarations in a source file. First, declarations are grouped either as contiguous sequences of ordinary Haskell declarations or as sets of declarations introduced by a single splice. The groups are then processed in a top-to-bottom manner, where declarations are in scope only in their group and those below it. For example, suppose a file contains the following sequence of declarations:

```
data A = ...

$mkTypeB  --Inserts declaration for type B
```

Type A will be in scope when type B is declared. However, type B will not be in scope when type A is declared.

Let's now consider the following mutually recursive declarations:

```
data Value = Num Int
           | Fun (Value -> M Value)

$(mkMonad "M" [EnvT "Env" [t| [(String, Value)] |]])
```

The declaration of `Value` cannot precede that of `M` because `M` will not be in scope at the declaration of `Value`. However, the declaration of `M` also cannot precede that of `Value` because then `Value` will not be in scope at the declaration of `M`.

To rectify the problem, we must remove the reference to `M` from the declaration of `Value`. Instead, we parameterize `Value` over the type variable `m` as

```
data Value m = Num Int
             | Fun (Value m -> m (Value m))
```

The recursion can then be reintroduced in the declaration of `M` as

```
$(mkMonad "M" [envTRec "Env" "[(String, Value M)]"])
```

We note that such refactoring is not required for declarations in the front end, where TH is used only to generate declarations, not compile them.

6. Related Work

Espinosa's doctoral dissertation described the definition of a DSL called "Semantic Lego" embedded in Scheme containing a language of monads similar to that of MonadLab. However, Semantic Lego implements monad transformers in much the same way as is done in Haskell's MTL (without the type inference, of course)—i.e., programmers must order the transformers themselves.

Alternative approaches to implementing monads have occurred in the past. Filinski (1999) introduces a general approach to translating layered monads into a λ-calculus with first-class continuations. Separating monad specification from implementation as MonadLab does introduces the possibility of other implementation strategies. The High Assurance Security Kernel Laboratory[1] at the University of Missouri is building compilers that translate monadic separation kernels (Harrison and Hook 2008) directly into machine language. These kernels are built from particular combinations of monad transformers (especially, resumption monad transformers for concurrency). These compilers are not intended to be as general as Filinski's. Rather, they will take advantage of the restriction

[1] HASK home page: `http://monadgarden.cs.missouri.edu/wiki`.

to concurrency monads to produce verifiable object code with predictable space and time behavior. The original motivation behind MonadLab was to codify the monadic underpinnings of these kernels in a source language.

There have recently been a number of proposed categorical alternatives to monad transformers as a structure for combining monads (Lüth and Ghani 2002; Power 2006). Some of these (especially Lüth and Ghani, 2002) would seem to be an attractive target representation for MonadLab, although we have not explored this approach as of this writing. Another approach, based in operational semantics, is taken by Unimo (Lin 2006). Unimo encodes the syntax of monadic operations with generalized data types (Peyton Jones et al. 2004) and then encapsulates the semantics of monads in an evaluation function.

The "expression problem" (Wadler 1998) concerns the difficulty of extending data types with new constructors in a manner that does not require revision and recompilation of existing functions. One approach to the expression problem builds modular data types through the explicit encoding of a co-product in Haskell (Weaver et al. 2007; Swierstra 2008). The advantage of this approach is that the modular construction of data types is type-checked. MonadLab sidesteps the expression problem via the generate-then-type-check paradigm followed by Template Haskell. The entire implementation of a MonadLab program is generated before it is type-checked, thereby avoiding the necessity of encoding monadic syntax and semantics within Haskell's type system.

7. Conclusions & Future Work

The MonadLab compiler relies on Template Haskell's meta-programming constructs to generate monad implementations. Template Haskell's meta-programming capabilities are weaker than those of MetaML (Taha and Sheard 2000) and MetaOCaml[2] in that Template Haskell programs have (at most) two stages and MetaML and MetaOCaml support arbitrary staging levels. While Haskell code is being constructed in processing the first stage, Template Haskell performs little or no type checking. The generate-then-type-check paradigm of TH permits MonadLab to ignore types until the entire monad implementation has been generated.

One cannot define C's familiar printing function, (`printf` *format args*), in Haskell, because the type of `printf` depends on *format* (Sheard and Peyton Jones 2002). Template Haskell allows one to write a type-safe version of `printf` because the definition for the function can be generated using particular inputs. The situation with MonadLab is similar. One cannot write the MonadLab compiler in Haskell, either. Speaking very roughly, its type would have to be of the form: "*MonadLab*" $\rightarrow$ "*Monad*", but the type of the *Monad* result depends on the input declaration. Template Haskell's extra expressiveness makes the language well-suited to implementing first class monads as in MonadLab.

MonadLab is not without its limitations. Firstly, the current MonadLab implementation is not easily extensible. Adding new base monads or monad transformers requires non-trivial modification of the source. Secondly, MonadLab does not support parameterized monad definitions. Using the MTL, one can write:

```
type MyMonad s a = StateT s (Maybe a)
```

MonadLab does not support this kind of abstraction. And thirdly, as has already been discussed with respect to the error transformer, the built-in ordering of transformer application precludes certain non-proper morphism semantics available with the MTL. Each of these shortcomings provides opportunities for future work and will be addressed in later versions of MonadLab.

[2] The MetaOCaml home page is: `http://www.metaocaml.org/`.

How does one explain what a monad is to people who have never heard of them? How can one describe research results based on monadic structuring to other researchers who may, at best, have only heard the term "monad"? Anyone who has ever attempted to do either task will recognize their inherent difficulty. One can choose to speak at a "bird's eye" level, risking the possibility that your talk will have little impact. Or, one can try to speak at a "worm's eye" level, which will almost certainly overwhelm many in your audience. Where is the middle ground? It should be noted that this challenge exists whether you are lecturing to undergraduates or presenting a paper to seasoned computer scientists.

One can envision MonadLab as a teaching tool for undergraduates. MonadLab can also serve as a vehicle for quickly, yet effectively, explaining monadic ideas to colleagues. Making monads first-class is a step towards making monads—and, by association, Haskell—more accessible. Typically, when a student first learns programming, he starts from simple "Hello World" examples in the language being studied and, through a series of experimental modifications, learns what each language construct "does." Because MonadLab hides the semantic boilerplate underlying monad construction, we believe that undergraduates will be able to apply this same experimental approach to learning monadic programming in Haskell. The proof of this pudding is in the tasting, of course, so starting Winter semester of 2009, MonadLab will be integrated into an undergraduate level course on functional programming in Haskell (based on Graham Hutton's excellent new text (Hutton 2007)) and language interpreters at the University of Missouri.

References

William Harrison and James Hook. Achieving information flow security through monadic control of effects. Invited submission to: *Journal of Computer Security*, 2008. 46 pages. Accepted for Publication.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

Pericles Kariotis. Code Repository for MonadLab. June 2008. Available from `http://monadgarden.cs.missouri.edu/MonadLab`.

Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 333–343. ACM Press, 1995.

Chuan-Kai Lin. Programming monads operationally with Unimo. In *International Conference on Functional Programming ICFP'06*, pages 274–285, 2006.

Christoph Lüth and Neil Ghani. Composing monads using coproducts. In *International Conference on Functional Programming ICFP'02*, pages 133– 144. ACM Press, September 2002.

Ian Lynagh. Template Haskell: A report from the field. May 2003a.

Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. May 2003b.

Nikos S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, 2001. Expanded version available as a tech. report from the author by request.

Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

John Power. Generic models for computational effects. *Theoretical Computer Science*, 364(2):254–269, 2006.

Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. URL: http://www.cse.unsw.edu.au/~sseefried/papers.html, March 2004.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Tim Sheard and Simon Peyton Jones. Notes on Template Haskell version 2. November 2003.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

Phillip Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages (POPL)*, pages 1–14. ACM Press, 1992.

Phillip Wadler. The expression problem. November 1998. Available from `www.daimi.au.dk/~madst/tool/papers/expression.txt`.

Philip Weaver, Garrin Kimmell, Nicolas Frisby, and Perry Alexander. Modular and generic programming with InterpreterLib. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 473–476. ACM Press, 2007.