Monadification of Functional Programs[⋆]

Martin Erwig and Deling Ren

Oregon State University
Department of Computer Science
Corvallis, OR 97331, USA
[erwig|rende]@cs.orst.edu

Abstract. The structure of monadic functional programs allows the integration of many different features into such programs by just changing the definition of the monad and not the program, which is a desirable feature from a software engineering and maintenance point of view. We describe an algorithm for the automatic transformation of a function into such a monadic form. We argue that the proposed transformation is sound and under certain conditions also complete. We also show how invertible monads can be used to extend the scope of the proposed transformation and can help to prevent the proliferation of monads over a program.

Keywords: Haskell, Monad, Program transformation

1 Introduction

Monads provide a standardized way to integrate a variety of language features into functional languages, such as I/O interaction, state-based computation, or exception handling [16]. The notion of monad originates in category theory [12]. Eugenio Moggi [14, 15] used monads to structure semantics definitions, which paved the way for using monads in functional languages [20]. An excellent survey is given by Phil Wadler in [21].

Despite their usefulness, monads are difficult to understand for beginners, and even experienced Haskell programmers often begin the development of a functional program by writing non-monadic functions and later add monadic aspects to the code. In other situations, monads are added to functions only temporarily, for example, for debugging purposes or to implement other tracing functionality. These monads are often to be removed later from the program. In any case, the task of turning one or more functions into monadic computations is a routine exercise for functional programmers. We call this process *monadification*.

From a more general point of view, it has been observed that monads can support aspect-oriented programming [13]. Aspect-oriented programming is concerned with adding functionality to a program that is orthogonal to the program's functional decomposition [1, 5]. One example is exception handling that cannot be localized in one module, but spreads the whole program. A program that is given in monadic form is prepared for the addition of aspects because the aspects can be realized through the definition of the monad operations; the monadic program structure ensures that the aspects are executed at the appropriate places

[⋆] Technical Report TR02-60-10, Department of Computer Science, Oregon State University, October 2002 in the program (these are called “join points”). One criticism of this approach to aspect-oriented programming has been that a programmer has to provide the join points explicitly and in advance, by providing the monadic program structure, which compromises much of the benefits of aspect-oriented programming [9]. With the proposed monadification this limitation of the monadic approach can be eliminated. To some degree, monadification can detect join points automatically.

In a different context, Ralf Lämmel uses a program transformation technique called *sequencing* [6] to flatten a expression into `let` expressions [10]. This intermediate result is then transformed into a monadic computation. However, the described approach is limited to several special cases and does not consider the general case of monadification of arbitrary function definitions. A major limitation is that Lämmel’s approach cannot deal with general cases of recursive definitions, in particular, when the recursive calls might not be *liftable* (which is explained below) due to locally used variables.

In Haskell a monad is a unary type constructor with two associated functions, which is expressed by a type class (more precisely, as a constructor class) `Monad`.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

This definition expresses that any type constructor `m` can be regarded as a monad once these two operations have been defined (the function `>>=` is also called *bind*).[1]

As an example for monadification, we consider the task of adding exception handling code to a function definition. Consider the following simple expression data type and a corresponding evaluating function, which we have taken from Richard Bird’s book [3, Chapter 10].

```
data Expr = Con Int
          | Plus Expr Expr
          | Div Expr Expr

eval :: Expr -> Int
eval (Con x)    = x
eval (Plus x y) = eval x + eval y
eval (Div x y)  = eval x `div` eval y
```

One limitation of the shown definition of `eval` is that it does not handle exceptions. For example, when `eval` is applied to the argument `Div (Con 1) (Con 0)`, a runtime error will occur. In order to capture such exceptions, `Int` values can be wrapped by the `Maybe` monad, which is a type constructor defined as:

```
data Maybe a = Just a | Nothing
```

[1] In the Haskell 98 standard [17], the monad class contains two further functions: (i) a variation of `(>>=)`: `m >> f = m >>= \_->f` and (ii) a function `fail` that is invoked on pattern matching failure in `do` expressions. For this paper, these differences are not relevant. In addition, the monadic structure requires `return` to be a left and right unit of `>>=` and `>>=` to be associative in a certain sense.

A `Just` constructor denotes a normal state associated with a value of type `a` while a `Nothing` constructor denotes an error state where no value should be stored. Instances of the two basic monad operations, `return` and `>>=`, are defined for the `Maybe` type as follows.

```
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return        = Just
```

The `>>=` operation works as follows. If the previous computation has produced a proper value (indicated by the enclosing constructor `Just`), the value obtained so far (`x`) is passed on for further computation (`k`). But if an error has occurred (indicated by the constructor `Nothing`), this error state is propagated, regardless of the following computation. In Haskell, the `do` notation is provided as a convenient syntax for monadic programming. Expressions using `do` are translated into calls to the monadic functions `return` and `>>=` (see Section 3).

We want to use the `Maybe` type in the `eval` function in the following way. Whenever a computation can be performed successfully, the corresponding result value is injected into the `Maybe` type by wrapping a call to `return` around it. On the other hand, any erroneous computation should result in the `Nothing` constructor. This strategy has an important implication on the definition of `eval`. First of all, the result type of `eval` changes from `Int` to `Maybe Int`. As a consequence of this, the results of recursive calls to the function `eval` cannot be directly used anymore as arguments of integer operations, such as `+` or `div`. Instead, we have to extract the integer values from the `Maybe` type (if possible) or propagate the `Nothing` constructor through the computation. Doing this "by hand", that is, by explicitly pattern matching all `Maybe` subexpressions in `eval` with `case` expressions can become extremely tedious for larger programs. At this point the fact that `Maybe` is defined as an instance of the `Monad` class comes into play: the monad performs the unwrapping of values and propagation of `Nothing` automatically through the function `>>=`. However, this function has to be placed in `eval` at the proper places to make the monadic version of `eval` work. The (changed) types of the involved objects more or less dictate how this has to be done. In short, all recursively computed values have to be bound to variables that can then be used as arguments of integer operations—this binding process is the inverse operation to the wrapping performed by `return`.

The monadified version of `eval` is given below using the `do` notation. In this paper we use the naming convention to append an `M` to names of monadified functions.

```
evalM :: Expr -> Maybe Int
evalM (Con x)    = return x
evalM (Plus x y) = do i <- evalM x
                      j <- evalM y
                      return (i+j)
evalM (Div x y)  = do i <- evalM x
                      j <- evalM y
                      if j==0 then Nothing
                              else return (i `div` j)
```

The monadification consists of two part: the `Maybe` monad is employed to hide the error status; the adaptation in the last two lines in the above code catches the exception and correctly set the error status. The first change should preserve type correctness and semantics. The second change, the introduction of extra actions, changes the semantics, but does not change the types.

The advantage of `evalM` over `eval` is its proper handling of divide-by-zero errors, which do not cause runtime errors anymore.

After having examined this and various other examples of monadification (see Section 2), we can observe that monadification is mostly a mechanical process that can be described by a systematic change of the source program. We call this systematic method of monadifying source programs *programmed monadification*. Such a transformation can be captured by the definition of a monadification operator, which can be implemented, for example, with a meta language or as a stand-alone tool. Such a monadification operator should have nice properties, such as preserving syntax and type correctness of the transformed program. Moreover, monadification should change the program as little as possible and as much as needed, that is, the monadified program should behave similarly to the original program, only those parts that should be changed/improved by the introduction of the monad should expose a different behavior. We will formalize this idea in Section 5. Programmed monadification has several advantages over manual monadification:

1. *Reliability.* Programmed monadification will be done on the abstract syntax level, no syntax errors can be introduced. Moreover, if the monadification operator is well designed and implemented, type correctness of the resulting program can be also guaranteed. The proper design of the monadification operator is the main contribution of this paper.
2. *Reusability.* We may need to monadify different functions with the same monad. For example, various functions that need to manipulate on integer values may all raise divided-by-zero exceptions. We can use the same monadification program for adding exception handling repeatedly.
3. *Versatility.* A function can be monadified with different monads, producing different functions for different purposes.
4. *Efficiency of Transformations.* Using a tool to perform repeated monadification tasks is also much faster than performing all the required changes with a text editor.

The rest of this paper is structured as follows. We demonstrate in Section 2 the scope of monadification by several practical examples. In Section 3, we collect requirements of the monadification operator by considering several small functions that illustrate implications on monadifications in different situations. These requirements prepare for a formal definition that is developed in Section 4. In Section 5 we define several correctness criteria for monadification and give corresponding correctness results for our monadification operator. The concept of *invertible monads* is introduced in Section 6. We show how invertible monads can be used to lift a limitation of the monadification operator and how they can be used to limit the proliferation of monads all over a program. Section 7 briefly discusses the aspect of adding monadic actions to monadified functions. Finally, we present some conclusions in Section 8.

2 Examples of Monadifications

By applying different monads, and inserting different monadic actions, the evaluator can be monadified in different ways, exhibiting various behaviors fitted for different applications. In this section, we will demonstrate some of these monadifications.

2.1 Exception Handling

In Section 1, we have seen how the evaluator is monadified by a `Maybe` monad to realize a simple form of exception handling. The `Maybe` monad simply regards all errors as the same, without any information about the error. This is sometimes not enough. The following `Exc` monad associates an error message with the error state, which can be shown to the user.

```
data Exc a = Raise String | Return a
```

The monad instance for `Exc` is similarly to `Maybe`, where an exception is indicated by `Raise` and propagated through the whole computation:

```
instance Monad Exc where
  return x         = Return x
  (Raise x)  >>= _ = Raise x
  (Return a) >>= f = f a
```

With the `Exc` monad, we can now represent the divided-by-zero error by associating a string with the `Raise` constructor:

```
evalM :: Expr -> Exc Int
evalM (Con x)    = return x
evalM (Plus x y) = do i <- evalM x
                      j <- evalM y
                      return (i+j)
evalM (Div x y)  = do i <- evalM x
                      j <- evalM y
                      if j==0 then Raise "divide by 0"
                              else return (i ‘div‘ j)
```

2.2 Counting Operations

We will consider a state transformer monad that takes an initial state and returns a value paired with the final state. A simple example of a state transformer is an integer counter, whose state is represented by an integer. The definition of a general state transformer is defined as:

```
data ST s a = Trans (s -> (a,s))
instance Monad (ST s) where
  return x        = Trans (\s -> (x,s))
  (Trans c) >>= f = Trans (\s -> let (x,s’) = c s
                                     Trans d = f x
                                 in d s’)
```

The call `return x` produces a state transformer which returns `x` as the value and leaves the state unchanged. The call `(Trans c) >>= f` applies the first state transformer to the initial state `s`, yielding intermediate value/state pair `(x,s')`; then it applies state transformer `f x` to `s'`. We need an operation that increases the counter by 1, and returns no value.

```
tick :: ST Int ()
tick = Trans (\c -> ((),c+1))
```

To count the number of operations performed during the evaluation of an expression, we take the original evaluator, monadify it with `ST`, and perform a small change to the code to cope with counting:

```
evalM :: Expr -> ST Int Int
evalM (Con x)    = return x
evalM (Plus x y) = do i <- evalM x
                      j <- evalM y
                      tick
                      return (i+j)
evalM (Div x y)  = do i <- evalM x
                      j <- evalM y
                      tick
                      return (i `div` j)
```

2.3 Producing Output

In the examples we have seen so far, the monadification was done in a similar way: take the original program, which is not monadic, wrap the return value with a call to `return`, and add some *local* "action" before `return`. The following example deviates from this schema in that it requires access to variables that are not defined near to the location of the action. First, we define an `Out` monad that couples an output string with the result value. The string is to be printed at the end; we have to thread all the output strings through the whole computation.

```
data Out a = Out (String,a)
instance Monad Out where
  return x = Out ("",x)
  Out (s,x) >>= f = Out (s',x') where Out (s1,x') = f x
                                      s'          = s++s1
```

The call `return x` wraps value `x`, together with an empty string. The call `Out (s,x) >>= f` applies `f` to `x`, returns its result value and appends the output to the `s`.

A basic operation on `Out` is to add a string to the output and return no value:

```
out :: String -> Out ()
out x = Out (x,())
```

To add an execution trace to the evaluator, the original evaluator is monadified with `Out`. The code for tracers is also added:

e ::= c | v | `\`v`->`e | e e | `let` v`=`e `in` e | `case` e `of` `{`p_1`->`e_1`;` $\ldots$ `;`p_n`->`e_n`}`

Fig. 1. Syntax of the object language.

```
eval :: Expr -> Out Int
eval (Con x)    = do out (show (Con x)++"="++show x)
                     return x
eval (Plus x y) = do i <- eval x
                     j <- eval y
                     out (show (Plus x y)++"="++show (i+j))
                     return (i+j)
eval (Div x y)  = do i <- eval x
                     j <- eval y
                     out (show (Div x y)++"="++show (i `div` j))
                     return (i `div` j)
```

The interesting aspect of this example is that the changes required for the original program are not local in the sense that they cannot be achieved by just adding a context-independent expression before `return`. Rather, the inserted expressions need to refer to the parameter of the function, which makes automatic transformation more challenging.

Now we have seen several examples in which monads extend the functionality of programs. Next we consider how monadification can be automated.

3 The Essence of Monadification

From the above examples, we can see that the process of extending a program with monads is rather mechanical. Therefore, we would like monadifications to be performed automatically by a meta program. In this section, we try to identify the rules that govern correct monadification by considering a number of small examples.

Given a function f of type t_1`->`t_2`->`$\ldots$`->`t_k`->`t, we want to change f so that the type of the return value of f is changed from t to some monadic type $m\ t$, that is, after the change, the type of f is changed to t_1`->`t_2`->`$\ldots$`->`t_k`->`$m\ t$, where m is a monad type.

In a functional setting, a function can also be a value. A multi-parameter function can be considered to have more than one "return type". For instance, if a function has type t_1`->`t_2`->`t_3, the "return type" of it could be either t_3 or t_2`->`t_3. Therefore, monadification can also be performed on different return values. In this paper, we consider the right-most type as the return type of a function unless stated otherwise.

For the sake of simplicity, we consider as an object language lambda calculus extended by `case` expression and `let` expressions. The syntax is defined in Figure 1. For syntactic convenience we make use of the `do` notation, which can be translated into lambda calculus based on the following equalities.

`do {`e`}` $=$ e
`do {`e`;`$stmts$`}` $=$ e `>>= \_ -> do {`$stmts$`}`
`do {`x `<-` e`;`$stmts$`}` $=$ e `>>= \x -> do {`$stmts$`}`

Next we will examine several examples to better understand how to monadify functions in different situations.

The first example demonstrates the notion of a *return expression*, which is an expression that is subject to wrapping by the monad operation `return`.

```
f :: Int -> Int -> Int
f = \x -> \y -> x+y
```

If we consider `f` as a two-parameter function, after stripping off two lambda abstractions, `x+y` is the expression that defines the result. The most direct way to monadify the function is to wrap a call to `return` around the return expression.

```
fM :: Monad m => Int -> Int -> m Int
fM = \x -> \y -> return (x+y)
```

The body of `f` could be of any syntactic form. It might be the case that the lambda abstractions are embedded in other syntactic structures, such as `case` expressions or applications. Here is such an example where lambda abstractions are embedded in a `case` expression.

```
f :: Int -> Int -> Int
f = \x -> case x of
            0 -> \y -> y+1
            n -> \z -> z-1
```

The definition of `f` contains two return expressions: `y+1` and `z-1`. To monadify this function, `return` should be applied to both of them.

```
fM :: Monad m => Int -> Int -> m Int
fM = \x -> case x of
             0 -> \y -> return (y+1)
             n -> \z -> return (z-1)
```

Moreover, a function can be defined in terms of other functions, or be the result of an application. In these cases, the number of parameters to the function does not match the number of lambda abstractions in the function definition. For example:

```
f :: Int -> Int
f = (\x -> \y -> x+y) 0
```

The syntactic structure of this one-parameter function is an application instead of a lambda abstraction. In this form, there is no return expression to apply `return` to. However, the above definition is η-equivalent to the following definition.

```
f' = \z -> (\x -> \y -> x+y) 0 z
```

After the return expression has been exposed, it can now be monadified in the usual way.

```
fM :: Monad m => Int -> m Int
fM = \z -> return ((\x -> \y -> x+y) 0 z)
```

All the examples we have seen so far are non-recursive functions. In the case that the definition of `f` contains calls to itself, the monadification is more complicated because the corresponding subexpressions change their types due to the monadification of `f`. Not properly handled, these subexpressions would introduce type errors. Let us look at a simple example.[2]

```
f :: Int -> Int
f = \n -> n*f (n-1)
```

If we simply wrap a `return` to the return expression `n*f (n-1)`, the result `return (n*f (n-1))` is *not* type correct since the type of `f (n-1)` is `m Int` and not `Int`, which is required for the application of `*`. The solution is to bind the expression `f (n-1)` to a variable, say `x`, and use `x` in place of `f (n-1)`:

```
fM :: Monad m => Int -> m Int
fM = do {x <- fM (n-1); return (n*x)}
```

Still, this is not a complete solution. An expression being bound and lifted out may contain local variables, which will become free variables after the lifting. This case can be exemplified by the following function where the local variable is introduced by the second alternative of the `case` expression.

```
f :: Int -> Int
f = \x -> case x of
            0 -> 1
            n -> n*(f (n-1))
```

Since the scope of `n` is limited to the second body of the `case` expression, we should be careful not to lift `f (n-1)` outside that scope. In this case, a correct way is to move the operation down to all bodies of the `case` expression. Another reason that we do not want to lift `f (n-1)` is that in the original program, it is evaluated only when the second alternative of the `case` is matched. Lifting it out might cause it to be evaluated more than necessary, which increases the strictness of the program.

```
fM :: Monad m => Int -> m Int
fM = \x -> case x of
             0 -> return 1
             n -> do {y <- fM (n-1); return (n*y)}
```

Since all bodies of a `case` expression have the same type as the type of the whole expression, operations on the `case` expression can be simply moved down to the bodies.

Such scoping problems can also be introduced by lambda abstractions because in a lambda abstraction the type of the body differs from that of the whole expression by an "arrow". Consider the following function.

```
f :: Int -> Int
f = \n -> (\x -> n*(f x)) (n-1)
```

[2] This function, as some other examples that appear in the rest of the paper, is non-terminating. However, this is not really relevant because it could be easily changed into a terminating definition by adding a `case` expression. To reveal the essential structure, we use the simpler non-terminating forms instead.

In this example, the return expression is `(\x -> n*(f x)) (n-1)`, the recursive call `f x` needs to be lifted and bound. But the scope of `x` is within the lambda abstraction. Here, we can monadify the anonymous function `(\x -> n*(f x))` and change its type from `Int -> Int` to `Int -> m Int`.

```
fM :: Monad m => Int -> m Int
fM = \n -> (\x -> do {y <- fM x; return (n*y)}) (n-1)
```

In summary, three steps are involved in monadifying a function:

– *Navigating.* Locate the return expressions (wrapping points) in the function definition. The basic approach is to move down k lambda abstractions. Navigating might be taken down into `case` expressions. Whenever we cannot find enough lambda abstractions, we use η-expansion to create additional abstractions.
– *Wrapping.* Apply `return` to the expressions located in the navigation step. Before applying `return`, locate recursive calls and bind them to (fresh) variables (see next item); special care is required in case that recursive calls cannot be safely moved outside.
– *Binding.* Identify recursive calls and bind them to (fresh) variables; special care is required in case that recursive calls involve local variables.

The last two steps are performed together for practical reasons since variables to which recursive calls are bound are generally used in return expressions.

4 Automatic Monadification

We want to monadify a Haskell function f that is defined by $f = e$ into a function called $f_{\mathcal{M}}$. We define an operator $\mathcal{M}$ such that $f_{\mathcal{M}} = \mathcal{M}(f, k, e)$ is the sought monadification of f. In this section, we are only concerned about the refactoring aspect of monadification, that is, we ignore the insertion of monadic actions. We will address this issue later in Section 7. $\mathcal{M}$ takes three parameters: the name of the function to be monadified, the number of parameters to the function and the definition. The name of the function and the number of parameters to the function are needed to identify recursive calls. We make the following assumption for f: if f is recursively defined, it is always applied to at least k arguments in e, that is, there is no partial application of f that leaves calls to f "undersaturated" with arguments. This condition can be checked through the predicate "for all subexpressions e' of e: $\mathcal{R}(f, 0, e') \Rightarrow \mathcal{R}(f, k, e')$". However, this is not a limitation of the algorithm because we can always supply additional arguments for such calls through η-expansion before applying monadification.

4.1 Characterizations of Subexpressions

The definition of the monadification operator is steered by properties of expressions. First, we need a predicate that tells whether or not e contains a recursive call to f with k arguments. This property is captured in the definition of the predicate $\mathcal{R}(f, k, e)$. $\mathcal{R}$ is inductively defined in Figure 2.

In addition, we also need the information whether or not e contains a recursive call to f as a *strict* subexpression, that is, e contains a recursive call to f, but e itself

$$\text{CALL}\ \frac{}{\mathcal{R}(f,k,f\ e_1\ e_2 \ldots e_k)} \qquad \text{ABS}\ \frac{\mathcal{R}(f,k,d)}{\mathcal{R}(f,k,\backslash v\texttt{->}d)}$$

$$\text{APP}\ \frac{\mathcal{R}(f,k,e_1)}{\mathcal{R}(f,k,e_1\ e_2)} \qquad \frac{\mathcal{R}(f,k,e_2)}{\mathcal{R}(f,k,e_1\ e_2)}$$

$$\text{LET}\ \frac{\mathcal{R}(f,k,e_1)}{\mathcal{R}(f,k,\texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2)} \qquad \frac{\mathcal{R}(f,k,e_2)}{\mathcal{R}(f,k,\texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2)}$$

$$\text{CASE}\ \frac{\mathcal{R}(f,k,e')}{\mathcal{R}(f,k,\texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\}})} \qquad \frac{\mathcal{R}(f,k,e_j) \quad \text{for some } j \in \{1,\ldots,n\}}{\mathcal{R}(f,k,\texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n\texttt{\}})}$$

Fig. 2. Recursive calls.

is not a call to f. We write $\mathcal{S}(f,k,e)$ if e has this property. $\mathcal{S}$ can be defined in terms of $\mathcal{R}$ as follows. $\mathcal{S}(f,k,e) = \mathcal{R}(f,k,e) \wedge \nexists\ e_1, e_2, \ldots, e_k$ such that $e = f\ e_1\ e_2\ \ldots\ e_k$.

Another relationship between an expression e and its subexpressions e' is whether it is safe to lift e' to the outside of e and bind it to a variable. As we have elaborated in the examples, if e' contains a variable that is local to e, say x, we shall not lift e' because otherwise x would become unbound. Moreover, in the case that e' resides in the body of a `case` expression, lifting e' might change the termination behavior of the program, that is, lifting might make the program less lazy. To avoid this, we shall not lift such e' either. If e' can be safely lifted outside e, we say e' is a *liftable* subexpression of e. The liftability predicates $\mathcal{L}$ is defined in Figure 3.

$$\text{ABS}\ \frac{\mathcal{L}(d,e') \qquad v \notin FV(e')}{\mathcal{L}(\backslash v\texttt{->}d,e')} \qquad \text{APP}_1\ \frac{\mathcal{L}(e_1,e')}{\mathcal{L}(e_1\ e_2,e')} \qquad \text{APP}_2\ \frac{\mathcal{L}(e_2,e')}{\mathcal{L}(e_1\ e_2,e')}$$

$$\text{LET}_1\ \frac{\mathcal{L}(e_1,e') \qquad v \notin FV(e')}{\mathcal{L}(\texttt{let}\ v\texttt{=}e_1\ \texttt{in}\ e_2,e')} \qquad \text{LET}_2\ \frac{\mathcal{L}(e_2,e') \qquad v \notin FV(e')}{\mathcal{L}(\texttt{let}\ v\texttt{=}e_1\ \texttt{in}\ e_2,e')}$$

$$\text{CASE}\ \frac{\mathcal{L}(e_0,e')}{\mathcal{L}(\texttt{case}\ e_0\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\}},e')}$$

Fig. 3. Liftable subexpressions.

4.2 Locating Return Expressions

We define a navigation operator $\mathcal{N}$ that moves monadification across lambda abstractions and eventually passes the found result expressions to the wrapping operator $\mathcal{W}$. More precisely, $\mathcal{N}(f,k,n,e)$ tries to "strip off" n lambda abstractions from e and then passes the result to $\mathcal{W}$. $\mathcal{N}$ can be defined inductively as follows.

For the base case, when $n = 0$, e can be directly wrapped. Otherwise, the syntactic structure of e is scrutinized, in case a lambda abstraction is not present, η-expansion is necessary. In all expressions below, z has to be a fresh variable with respect to e, that is, z has to be chosen such that $z \notin FV(e)$. Moreover, we assume

$n > 0$.

$$\begin{array}{ll}
\mathcal{N}(f,k,0,e) & = \mathcal{W}(f,k,e) \\
\mathcal{N}(f,k,n,c) & = \mathcal{N}(f,k,n,\texttt{\\}z\texttt{->}c\ z) \\
\mathcal{N}(f,k,n,v) & = \mathcal{N}(f,k,n,\texttt{\\}z\texttt{->}v\ z) \\
\mathcal{N}(f,k,n,\texttt{\\}v\texttt{->}d) & = \texttt{\\}v\texttt{->}\mathcal{N}(f,k,n-1,d) \\
\mathcal{N}(f,k,n,\texttt{let}\ v\texttt{=}e_1\ \texttt{in}\ e_2) & = \mathcal{N}(f,k,n,\texttt{\\}z\texttt{->(let}\ v\texttt{=}e_1\ \texttt{in}\ e_2\texttt{)}\ z) \\
\mathcal{N}(f,k,n,e_1\ e_2) & = \mathcal{N}(f,k,n,\texttt{\\}z\texttt{->(}e_1\ e_2\texttt{)}\ z) \\
\mathcal{N}(f,k,n,\texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\}}) & = \\
 & \begin{cases} \mathcal{N}(f,k,n,\texttt{\\}z\texttt{->case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\ z\texttt{\}}) & \text{if } \mathcal{R}(f,k,e') \\ \texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}\mathcal{N}(f,k,n,e_i)\texttt{\}} & \text{otherwise} \end{cases}
\end{array}$$

It is worth mentioning that we use a slightly specialized version of η-expansion in the transformation of `case` expression. Because of the following fact:

$$\texttt{(case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\})}\ e = \texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\ e\texttt{\}}$$

we can customize the η-expansion for `case` expressions as follows.

$$\texttt{case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\}} = \texttt{\\}z\texttt{->case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\ z\texttt{\}}$$

The reason for using this relationship instead of the general law is related to the definition of the wrapping operator $\mathcal{W}$ that will be discussed below. If we had used the simple form of η-expansion, we had to pass an application $\texttt{(case}\ e'\ \texttt{of}\ \texttt{\{}p_i\texttt{->}e_i\texttt{\})}\ z$ to $\mathcal{W}$. If no topmost call to f in this expression is liftable, we will resort to applying $\mathcal{N}$ again, with the `case` expression as the parameter. This will lead to an infinite loop. By using the above transformation, we are able to avoid this non-terminating situation.

4.3 Wrapping Return Expressions

Having exposed return expressions, we need to change their types from t to $m\ t$. This is done by the operator $\mathcal{W}$, which takes the name of the function being monadified, the number of its parameters (that is, the number of parameters that are not being monadified in addition to the result type), and the expression to be wrapped.

First, if there are no recursive calls to f inside e, e will be wrapped by a `return` unless e is a direct call to f because in this case its type is already monadic. The condition is formally captured by the predicate $\neg\mathcal{S}(f,k,e)$.

$$\mathcal{W}(f,k,e) = \begin{cases} f_{\mathcal{M}}\ e_1\ e_2\ \ldots\ e_k & \text{if } e = f\ e_1\ e_2\ \ldots\ e_k \\ \texttt{return}\ e & \text{otherwise} \end{cases}$$

Otherwise, that is, if a topmost call to f inside e is liftable, the corresponding subexpression is lifted and bound to a fresh variable. The condition for this case is expressed formally by using the notion of contexts. A context is an expression with a "hole" and is written as $C\langle\,\rangle$. With contexts we can express the condition that, for example, e' is a subexpression of e by $e = C\langle e'\rangle$. Therefore, the condition that e contains a liftable call to f is expressed by: $e = C\langle f\ e_1\ e_2\ \ldots\ e_k\rangle \wedge \mathcal{L}(e, f\ e_1\ e_2\ \ldots\ e_k)$. To additionally ensure that $C\langle\,\rangle$ locates a topmost call to f, that is, a call that is not nested inside another call to f, we also require $\nexists C', e'_1\ e'_2\ \dots\ e'_k : C\langle z\rangle = C'\langle f\ e'_1\ e'_2\ \dots\ e'_k\rangle$ (for a fresh variable z).

$$\begin{aligned}\mathcal{W}(f,k,e) &= \texttt{do \{}z\ \texttt{<-}\ \mathcal{W}(f,k,f\ e_1\ e_2\ \dots\ e_k)\texttt{;}\ \mathcal{W}(f,k,C\langle z\rangle)\texttt{\}} \\ &\quad \text{where } z \notin \mathit{VARS}(e)\end{aligned}$$

Note that we require that $z \notin \mathit{VARS}(e)$, that is, z does not conflict with *any* variable in e, not only the free variables. This is because z replaces a subexpression of e and must not be captured by a binder in e.

Finally, if no topmost recursive call to f in e is liftable, we have to scrutinize the syntactic structure of e (since e contains calls to f, e cannot be a variable or constant).

$\boxed{e = \texttt{case}\ e'\ \texttt{of \{}p_i\texttt{->}e_i\texttt{\}}}$ In this case, we have to wrap and bind e' and move the operation down to the bodies e_i:

$$\mathcal{W}(f,k,e) = \texttt{do \{}z\ \texttt{<-}\ \mathcal{W}(f,k,e')\texttt{;}\ \texttt{case}\ z\ \texttt{of \{}p_i\texttt{->}\mathcal{W}(f,k,e_i)\texttt{\}\}}$$

$\boxed{e = \texttt{\textbackslash}v\texttt{->}e'}$ We cannot deal with this case. Consider the following example.

```
f :: Int -> Int -> Int
f = \x -> (\y -> f y x)
```

For $k = 1$, we regard the function `\y -> f y x` as the return value, that is, we need to change the type of this function from `Int -> Int -> Int` to `Int -> m (Int -> Int)`. The recursive call `f y` should be bound to a variable, but at the same time it cannot be lifted outside the return expression itself. It is possible to monadify this function to a function with type `Int -> m (Int -> m Int)`, for example,

```
fM = \x -> return (\y -> do {z <- fM y; z x})
```

However, this is not helpful for our monadification algorithm that fails in this case. A possible remedy is discussed in Section 6.

$\boxed{e = e_0\ e_1\ \dots\ e_m}$ where e_0 is not an application, that is, $e_0 \neq e'\ e''$ for any e', e''. If e_0 is a non-recursive `let` expression `let` x=e'_0 `in` e''_0 (that is, $x \notin FV(e'_0)$), it is converted to an equivalent β-redex `(\`x`->`e''_0`)` e'_0. A general solution to this case is to apply $\mathcal{W}$ to $e_1 \dots e_m$, which makes their types monadic, bind them to fresh variables with respect to e, say $z_1 \dots z_m$, and also apply $\mathcal{N}$ to e_0. This requires the recursive application $\mathcal{N}(f,k,m,e_0)$ because e_0 is regarded as a m-parameter function and has to change its return type to a monadic type. Only when e_0 is a recursive `let` expression and contains a non-liftable call to f, $\mathcal{N}(f,k,m,e_0)$ will apply η-expansions to e_0 and eventually pass it, applied to m arguments, down to $\mathcal{W}$ again, which would cause an infinite loop. So monadification stops with an error in this case. In all other cases, we have the following definition:

$$\mathcal{W}(f,k,e) = \texttt{do \{}z_1\texttt{<-}\mathcal{W}(f,k,e_1)\texttt{;}\ \dots\texttt{;}z_m\texttt{<-}\mathcal{W}(f,k,e_m)\texttt{;}\mathcal{N}(f,k,m,e_0)\ z_1 \dots z_m\texttt{\}}$$

This solution might introduce unnecessary bindings in some of the e_i ($1 \leq i \leq m$) (but not all). We can eliminate these by optimizing the resulting expression through the *left unit* monad law [3].

$\boxed{e = \texttt{let } x\texttt{=}e_1 \texttt{ in } e_2}$ In case x is not recursively defined, namely, $x \notin FV(e_1)$, e is treated like a β-redex.

$$\mathcal{W}(f,k,e) = \texttt{do \{}x \texttt{ <- } \mathcal{W}(f,k,e_1)\texttt{; } \mathcal{W}(f,k,e_2)\texttt{\}}$$

If x is recursively defined, but if $\neg\mathcal{R}(f,k,e_1)$ holds, which means there are no non-liftable calls in e_1, we can still wrap e by only wrapping e_2:

$$\mathcal{W}(f,k,e) = \texttt{let } x\texttt{=}e_1 \texttt{ in } \mathcal{W}(f,k,e_2)$$

But if not only x is recursively defined, but also its definition contains non-liftable calls to f, we are unable to apply $\mathcal{W}$ to it. This is basically the same situation as for lambda abstraction shown above.

Finally, we have to define the monadification operator $\mathcal{M}$, which can be directly given in terms of $\mathcal{N}$:

$$\mathcal{M}(f,k,e) = \mathcal{N}(f,k,k,e)$$

5 Correctness of Monadification

We introduce two notions of correctness for monadification: *soundness* and *completeness*. Ideally, the behavior of a monadified version of a function is identical to that of the original function, except that the return value is wrapped by a monad; in cases when the original function does not terminate with a result, the monadified function should not terminate either. These requirements are formalized as completeness and soundness properties as follows.

Let $f_\mathcal{M}$ be the function obtained by monadification of f. $f_\mathcal{M}$ is called a *complete monadification* of f if

$$f\ x_1 \ldots x_k \ = \ y \quad \Longrightarrow \quad f_\mathcal{M}\ x_1 \ldots x_k \ = \ \texttt{return}\, y$$

$f_\mathcal{M}$ is called a *sound monadification* of f if

$$f_\mathcal{M}\ x_1 \ldots x_k \ = \ z \quad \Longrightarrow \quad \texttt{return (}f\ x_1 \ldots x_k\texttt{)} \ = \ z$$

Soundness is a very strong criterion, which actually might not be always desirable in practice, in particular, when monadic actions are used. For example, the monadified function `evalM` from Section 1 is not a sound monadification of `eval` because it does return a value for `y = 0` while `eval` does not.

Before we evaluate the monadification algorithm presented in Section 4 according to these criteria, we recall the restrictions of the algorithm. There are two cases that $\mathcal{M}$ cannot deal with:

– A result expression is a lambda expression,which contains a recursive call involving local variables.
– A result expression is an application whose first part is a recursive `let` expression that contains a non-liftable recursive call.

We have already discussed the problem that binding recursive calls to variables might change the termination behavior of the transformed function. For `case` expressions we were able to avoid this problem by a classifying the calls as not liftable (see the definition of $\mathcal{L}$ in Figure 3) and eventually moving the monadification down into `case` expressions. However, the problem is generally always present in situations when expressions are lifted from non-strict functions because in a lazy evaluation setting, these recursive calls might not be evaluated in the original function, but they are when being extracted. This generally causes an "increased strictness" of monadified functions. In these cases, monadification is not complete. The following functions illustrate this case:

```
g = \_ -> 0
f = \x -> g (f x)
```

Since `g` does not rely on the input to produce a return value, `f` always terminates and returns `0`. But if we apply our monadification algorithm to `f`, we get:

```
fM = do {y <- fM x; return (g y)}
```

The actual evaluation depends on the strictness of the implementation of the `>>=` operation for the monad that is being used. For example, the implementation of `>>=` for the `Maybe` monad inspects the pattern of the argument and therefore forces the evaluation of the expression that is to be bound (this is probably the case for most monads). This means that the argument `fM` is evaluated to get the result and inevitably `fM` would not terminate. To avoid this problem, we could instead bind `y` to a function to delay the evaluation.

```
fM' x = do let y _ = fM' x
           return (g (y ()))
```

This solution works well when `g` has a polymorphic type. However, if `g`'s type is constrained by a signature to, say `Int -> Int`, the shown transformation will cause a type error because `g` is applied to an argument of type `m Int`.

A different aspect of this delaying approach is that it makes the monadified code quickly unreadable. To obtain programs that will be read and maintained by humans, this approach does not seem to be appropriate.

Now we can give the main results about the correctness of our monadification algorithm. The first result is that monadification is sound.

Theorem 1. *Given $f = e$, $f_{\mathcal{M}} = \mathcal{M}(f, k, e)$ is a sound monadification of f.*

To prove the soundness, we can consider two cases. If f is not recursively defined, the soundness can be shown by a structural induction on the function definition. Otherwise, we can perform an induction on the recursive evaluation of f. The base case is when the recursion of f ends, where no recursive evaluation takes place. For cases where a positive number of steps of recursion take place, the inductive hypothesis and left unit monad law are applied to conclude soundness.

Although monadification is not complete due to increased strictness, monadification is a complete transformation under eager evaluation.

Theorem 2. *Given $f = e$, $f_{\mathcal{M}} = \mathcal{M}(f, k, e)$ is a complete monadification of f under eager evaluation.*

Since we already have soundness, the completeness follows whenever $f_{\mathcal{M}}$ is not less defined than f. Under eager evaluation this condition is satisfied.

We could prove a stronger result for normal order evaluation for a class of functions that require lifting only out of polymorphic functions by using the delayed binding transformation discussed above.

Finally, we can show that the proposed algorithm terminates on all inputs. $\mathcal{M}$ is defined in terms of $\mathcal{N}$, which eventually passes expressions to $\mathcal{W}$. Whenever a recursive definition occurs, $\mathcal{W}$ is applied recursively to smaller subexpression so that the termination follows by a structural induction on the expression. There is one exception, namely when $\mathcal{W}$ is applied to an application $e_0 \ \ldots \ e_m$ and e_0 is a recursive `let` expression containing a recursive call that is not liftable. In that case, e_0 is passed to $\mathcal{N}$ that might expand it and pass it back to $\mathcal{W}$. Since our algorithm identifies this case, $\mathcal{W}$ is guaranteed to terminate.

6 Invertible Monads

In Section 4, we have seen a situation where $\mathcal{W}$ cannot be applied to a lambda abstraction. This was due to the need to lift a subexpression out of a context that would also lift variables out of their scope.

Another problem in applying monadification to a function in a module containing other functions is that monadification is only locally type correct, that is, although it guarantees the type correctness of the monadified function, it does not guarantee that callers of the monadified function deal with the new monadic type correctly. Global type correctness can be recovered by a static analysis that identifies all calls of a monadified function and monadifies the calling functions accordingly. However, this might lead to a proliferation of monadic types all over the program. We call this the problem of *monad infestation*.

The simple concept of *invertible monads* provides a (partial) solution to both of these problems. An invertible monad is a monad that also has a `receive` operation, which can be considered the dual operation to `return`. In Haskell, we can define invertible monads as a subclass of `Monad` as follows.

```
class Monad m => MonadInv m where
  receive :: m a -> a
```

Basically, the purpose of `receive` is to extract the value from a monad; it is similar to the `run` function that is defined for some monads in Haskell.

To give an example, we show how to define the `Maybe` type constructor as an instance of `MonadInv`.

```
instance MonadInv Maybe where
  receive (Just x) = x
```

As another example we define a `MonadInv` instance for state transformers. The idea of getting the value out of the monad is to apply the state transformer to an initial state and extract the value from a value/state pair. To do this, we need to know what the initial state is. Therefore, we can define a type class `Initializable` of initializable values. Any data type that is intended to be used as a state for state transformer can be made an instance of `Initializable` by providing a initial value. For instance, an initial value for integers could be `0`.

```
class Initializable s where
  initValue :: s

instance Initializable Int where
  initValue = 0
```

Now we can capture the requirement on the state of a state transformer monad to have an initial value by a corresponding class constraint in the instance definition for `MonadInv`.

```
instance Initializable s => MonadInv (ST s) where
  receive (Trans f) = let (x,_) = f initValue in x
```

In addition to the three basic monad laws [3], an invertible monad should also satisfy the following inversion law [8].

$$\texttt{receive (return}\,x) = x \qquad \textit{Monad Inversion}$$

This law ensures that values injected into a monad can be recovered by `receive`. It is easy to check that this law holds for the `Maybe` and the `ST` monads. Formally, `receive` corresponds to the natural transformation *force* [4] that is a part of the definition of abstract Kleisli categories [12, 2].

The value of the `receive` operation lies in the fact that we can use it to "unwrap" a monadic expression at any place, meaning that we can extract the value from a monad in-place without lifting and binding. Therefore, in the process of wrapping, whenever a recursive call to f is encountered that cannot be lifted, we can apply `receive` to get a proper non-monadic value instead. With this approach, the function `f` from Section 4.3 can be monadified as follows.

```
fM = \x -> return (\y -> (receive (fM y)) x)
```

This method is sound and complete (at least for monads that satisfy the monad inversion law). However, by escaping monads the essence of monads can be lost to some degree at those places where `receive` is used. Moreover, monadic actions cannot be inserted at these points. On the other hand, `receive` provides a way to make monadification work in some cases.

Another use for `receive` is to limit the effect that the monadification of a function has on the rest of a module. By wrapping `receive` around some or all calls to the monadified function we have precise control over what other functions have to be monadified. In this way, we can effectively bound monad infestation.

7 Adding Actions to Monads

So far, we have developed an algorithm for converting a function into monadic style. In most cases the goal of this is to add further code to these monads, which is sometimes also called *monadic action*. In the introductory example from Section 1, the conditional expression for handling divided-by-zero exceptions is such a monadic action. Other examples are the call to `tick` and `out` in the monads presented in Section 2.

A simple, but inflexible, approach is to (always) insert an action before `return`. Such an action can be passed down to $\mathcal{W}$ from $\mathcal{M}$ and $\mathcal{N}$ as a parameter. We could define a $\mathcal{W}_a$ in an almost the same way as $\mathcal{W}$ except changing `return` e everywhere to a `>>= return` e. However, this solution is very limited since the context of the return expression is totally ignored, which means that the same action is inserted before every `return`. No useful adaptation can be achieved in this way.

A more general approach can be obtained by using some form of *symbolic rewriting* [19, 18, 11] to describe the insertion of actions. The idea of symbolic rewriting is to match a pattern against an expression to obtain a pattern variable binding, then substitute the expression with another pattern, with pattern variables instantiated. The following rewrite rule describes the adaptation that is needed for divided-by-zero exception handling in the example in Section 1:

$$\texttt{return (}x\texttt{ `div` }y\texttt{)} \;\rightarrow\; \texttt{if }y\texttt{==0 then Nothing else return (}x\texttt{ `div` }y\texttt{)}$$

The pattern `return (`x `` `div` `` y`)` is matched against the expression ``return (i `div` j)`` which causes x to be bound to `i` and y to be bound to `j`. The expression is then substituted by the conditional expression on the right hand side, with x and y instantiated with `i` and `j`, respectively.

Similar rewrite rules can handle the examples from Sections 2.1 and 2.2. A more challenging one is the example in Section 2.3. The action that is to be inserted does not only refer to the return expression, but also to the variables `x` and `y`, which are variables introduced by a `case` expression. To describe such a transformation we need a rewriting system that is capable of expressing rewrite rules with variable context dependencies, which is not a common feature of existing rewrite systems, but can be implemented, for example, by the approach described in [11]. We can imagine the following rewrite rule for the task.

$$\texttt{case }e'\texttt{ of }p\texttt{ -> }\langle\texttt{return }e \;\rightarrow\; \texttt{out (show }p\texttt{ ++ "=" ++ show }e\rangle$$

It can be explained as: match an enclosing context of a `case` expression, bind the meat variable p, and then perform the shown rewrite rule.

8 Conclusions

We have shown how function definitions can be automatically converted into a monadic form by a process called *monadification*. The developed transformation is safe since it preserves syntax and type correctness of the transformed program. Moreover, monadification conserves the semantics of the original program as much as possible.

Monadification is an example of a generic program transformation that can be effectively used as a very general functional *refactoring* [7]. In many cases, such refactorings are only preparatory steps toward adding further functionality to programs. In the monadic setting this means to add monadic actions. In future work we will investigate this issue further. In particular, we will explore how we can add monadic actions while still preserving syntax and type correctness of the transformed program.

References

1. ACM. *Communications of the ACM*, volume 44(10), October 2001.
2. M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall International, 1996.
3. R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall International, London, UK, 1998.
4. A. Bucalo, C. Führmann, and A. Simpson. An Equational Notion of Lifting Monad. *Theoretical Computer Science*, 2002. To appear.
5. T. Elrad, M. Askit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44(10):33–39, 2001.
6. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *ACM Conf. on Programming Languages Design and Implementation*, pages 237–247, 1993.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.
8. J. Hughes. The Design of a Pretty-Printing Library. In *Advanced Functional Programing*, LNCS 925, pages 53–96, 1995.
9. R. Lämmel. Declarative Aspect-Oriented Programming. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 131–146, 1999.
10. R. Lämmel. Reuse by Program Transformation. In G. Michaelson and P. Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000.
11. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *4th Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 137–154, 2002.
12. S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
13. W. D. Meuter. Monads as a Theoretical Foundation for AOP. In *ECOOP Workshop on Aspect-Oriented Programming*, 1997.
14. E. Moggi. Computational Lambda-Calculus and Monads. In *IEEE Symp. on Logic in Computer Science*, pages 14–23, 1989.
15. E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1), 1991.
16. S. Peyton Jones. Tackling the Awkard Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.
17. S. L. Peyton Jones, J. Hughes, et al. Report on the Programming Language Haskell 98, 1999. `http://haskell.org/onlinereport`.
18. E. Visser. Strategic Pattern Matching. In *10th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1631, pages 30–44, 1999.
19. E. Visser and Z. Benaissa. A Core Language for Rewriting. In *Workshop on Rewriting Logic and Applications*, 1998.
20. P. Wadler. Monads for Functional Programming. In *Advanced Functional Programing*, LNCS 925, pages 24–52, 1995.
21. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.