Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 52 (2004) 101–129
Science of Computer Programming
www.elsevier.com/locate/scico

Monadification of functional programs

Martin Erwig*, Deling Ren

School of EECS, Oregon State University, Corvallis, OR 97331, USA

Received 31 May 2003; received in revised form 6 February 2004; accepted 15 February 2004
Available online 7 June 2004

Abstract

The structure of monadic functional programs allows the integration of many different features by just changing the definition of the monad and not the rest of the program, which is a desirable feature from a software engineering and software maintenance point of view. We describe an algorithm for the automatic transformation of a group of functions into such a monadic form. We identify two correctness criteria and argue that the proposed transformation is at least correct in the sense that transformed programs yield the same results as the original programs modulo monad constructors.

The translation of a set of functions into monadic form is in most cases only a first step toward an extension of a program by adding new features. The extended behavior can be realized by choosing an appropriate monad type and by inserting monadic actions into the functions that have been transformed into monadic form. We demonstrate an approach to the integration of monadic actions that is based on the idea of specifying context-dependent rewritings.

1. Introduction

Monads can be used to structure and modularize functional programs. The monadic form of a functional program can be exploited for quite different purposes, such as compilation or integration of imperative features. Despite the usefulness of monads, many functional programs are not given in monadic form because writing monadic code is not as convenient as writing other functional code. It would therefore be useful to have tools for converting non-monadic programs into monadic form, a process that we like to call *monadification*.

* Corresponding author. Tel.: +1-541-737-8893; fax: +1-541-737-3014.
E-mail address: erwig@cs.orst.edu (M. Erwig).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.03.004

Monadification-like algorithms for the purpose of compilation have been known for some time now [4,6]. The idea of these algorithms is to transform *all* functions in a program into monadic form. While these algorithms are quite simple and work well for their purpose, they are not well suited for the introduction of monads at selected places in a source program. On the other hand, Ralf Lämmel has identified the selective introduction of monads into functional programs as a useful source code transformation in [8]. He gives a specification of monad introduction for lambda calculus through a structured operational semantics.

In this paper we present the first detailed treatment of an algorithm for the selective introduction of monads into functional programs. We identify correctness criteria for monadification and investigate the correctness of the algorithm presented. We also demonstrate the principal limitation of all known monadification algorithms.

The rest of this paper is structured as follows. In the remainder of this introduction we illustrate the use of monads and discuss general issues concerning the automatic introduction of monads into functional programs. In Section 2, we define correctness criteria for monadification and point out a principal limitation of monadification. Moreover, we collect requirements of the monadification operator by considering several small functions that illustrate implications for monadifications in different situations. These requirements prepare for a definition that is developed in Section 3. In Section 4 we apply the correctness criteria to our monadification algorithm. The concept of *runnable monads* is discussed in Section 5. We show how runnable monads can be sometimes used to circumvent the principal limitation of monadification and how they can be used to limit the proliferation of monads all over a program. In Section 6 we describe how to add monadic actions to monadified functions using a rewriting approach. We discuss related work in Section 7. Finally, we present some conclusions in Section 8.

1.1. Why monads?

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 [14]. The notion of monad originates in category theory [12]. Moggi [13] used monads to structure semantics definitions, which paved the way for using monads in functional languages [18]. An excellent survey is given by Wadler in [19].

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] In addition,

[1] In the Haskell 98 standard [15], 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.

the monadic structure requires `return` to be a left and right unit of `>>=` and `>>=` to be associative in a certain sense; that is, the definition of the monad operations should obey the following laws:

```
        m >>= return = m
 return x >>= f      = f x
(m >>= f) >>= g      = m >>= (\x -> f x >>= g)
```

These laws are not enforced by Haskell; it is the programmer's responsibility to define the operations in such a way that these laws hold.

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 borrowed from Richard Bird's book [1, 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 run-time 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 follows.

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

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

```
instance Monad Maybe where
  Just x  >>= f = f x
  Nothing >>= f = 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 (`f`). 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 2).

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 applying `return` to 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`. Therefore, the results of recursive calls to the function `eval` cannot be directly used any longer 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.

Performing this unwrapping explicitly, that is, by 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 objects involved 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 operation inverse to the wrapping performed by `return`.

The monadified version of `eval` is given below using the `do` notation [1]. 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 process of `eval`'s monadification consists of two parts. First, the `Maybe` monad is employed to hide the error status. Second, the adaptation in the last two lines in the above code catches the exception and correctly sets the error status. The first change should preserve the type correctness and the semantics of `eval`. The second change, the introduction of 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 run-time errors any longer.

1.2. Automatic introduction of monads

Not only are monads difficult for beginners, they are awkward for experts, too—for example, they force the programmer to specify evaluation order. Therefore, even experienced Haskell programmers often begin the development of a functional program by writing non-monadic functions, later changing it to monadic form. 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, turning one or more functions into monadic computations is a frequently occurring task for functional programmers. We call this process *monadification*.

A tool for the *automatic* monadification of functional programs gives programmers the freedom to select monads on demand; they are not urged to adopt a monadic (and thus more imperative) style from the start. This freedom means an important aid for the process of development of functional programs, because a programmer does not have to worry about extensions of his or her program that might require a monadic structure for some of the functions. Monadification was also proposed in [17], but there it is assumed that programmers resort to the traditional approach for the adaptation by using a text editor. In [11] interpreters are written in a particular monadic style that facilitates the extensibility by new features. A drawback of this approach is that the program has to be written in a monadic style right from the beginning.

Automatic monadification has several advantages over manual monadification:

(1) *Reliability*. Since an automatic monadification operator works 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 integer values may all raise divide-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.

Examining the `eval` and various other examples of monadification (see, for example, [1,18]), we can observe that monadification is mostly a mechanical process that can be described by a systematic change of the source program. Such a transformation can be captured by the definition of a monadification operator. This monadification operator should preserve 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 be allowed to expose a possibly different behavior after monadic actions have been introduced. For example, the monadification of `eval` did not monadify the `+` or `div` operation, because the goal was to adapt the behavior of `eval` and not that of other operations.

2. The essence of monadification

Monadification is a source-level program transformation. The goal of monadification is to transform a given function f of type t_1 `->` t_2 `->` $\cdots$ `->` t_k `->` t into a function $\hat{f}$ of type to t_1 `->` t_2 `->` $\cdots$ `->` t_k `->` $m\ t$, where m is a monad type constructor. Note that t is an arbitrary type and can be, in particular, a function type, such as t_{k+1} `->` t_{k+2} `->` $\cdots$ `->` t_n. In other

words, the notion of "return value" is relative in the presence of higher-order functions; that is, a multi-parameter function can be considered to have more than one "return type". For instance, a function of type t_1 `->` t_2 `->` t_3 can be considered to return values of type t_3 or t_2 `->` t_3. Therefore, monadification can be performed on different return types, which means that the specification of the monadification of a function requires in addition to the function's name the number k of parameters that are not part of the monadified result type.

We can characterize the correctness of monadification by two properties of the results produced. Ideally, the behavior of a monadified function is identical to that of the original function, except that the return value is wrapped in a monad; in cases when the original function does not terminate with a result, the monadified function should not terminate either. These requirements can be formally expressed by referring to a semantics definition of the language to be transformed. To this end we assume an operational semantics that defines the reduction of expressions as a binary relation $\rightarrow$, whose reflexive, transitive closure is denoted by $\twoheadrightarrow$. Such a semantics is given, for example, in [14] where the semantics of monadic operations is a particular focus.

The correctness requirements can be formalized through the definition of completeness and soundness of monadified functions.

$\hat{f}$ is called a *complete monadification* of f if

$$\texttt{return}\ (f\ x_1 \ldots x_k) \twoheadrightarrow y \Longrightarrow \hat{f}\ x_1 \ldots x_k \twoheadrightarrow y$$

$\hat{f}$ is called a *sound monadification* of f if

$$\hat{f}\ x_1 \ldots x_k \twoheadrightarrow y \Longrightarrow \texttt{return}(f\ x_1 \ldots x_k) \twoheadrightarrow y$$

Note that the terms "complete monadification" and "sound monadification" refer to the result of the monadification transformation and not to the transformation itself.

Before we develop our monadification algorithm, we discuss a general limitation of all known monadification algorithms.

Proposition 1. *There are functions for which it is impossible to find a sound monadification for arbitrary arguments.*

Consider, for example, the task of monadifying the following function for one parameter:

```
f :: Int -> Int -> Int
f x y = if x == 0 then y else f y (x-1)
```

What we need is a function definition of the following form:

```
fM :: Int -> m (Int -> Int)
fM x = e1
```

We have to replace `e1` by an expression of type `m (Int -> Int)`. Of course, we cannot use an arbitrary expression of that type because we want the resulting function definition to behave like the original function except for the monad. Therefore, the defining expression of the function must be somehow retained. The expression

```
return (\y -> if x == 0 then y else f y (x-1))
```

$$p ::= c \mid v \mid p\ p$$
$$e ::= c \mid v \mid \backslash v \texttt{->} e \mid e\ e \mid \texttt{let}\ v\texttt{=}e\ \texttt{in}\ e \mid \texttt{case}\ e\ \texttt{of}\ \{p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n\texttt{}\}$$

Fig. 1. The syntax of the object language.

works well except that it ignores the monad in the recursive call to `f`. We cannot simply replace `f` by `fM` because `fM y` has a monadic type (and not a function type) and can therefore not be applied to `(x-1)`. The only way to make the recursion work is to bind `fM y` to a variable, say g, to obtain access to the integer function of the monadic value. Therefore, we have to replace `f y (x-1)` by an expression of the form `do {g <- fM y; e2}`. The problem is now: whatever we try to substitute for `e2`, we obtain a monadic value for the whole `do` expression, which can never match the type `Int` of `y`. Not surprisingly, none of the known monadification algorithms can monadify the function `f`.

To prepare for the definition of a monadification operator, we will identify the rules that govern correct monadification by considering a number of small examples.

We consider as an object language lambda calculus extended by `case` expressions and `let` expressions. The syntax is defined in Fig. 1.

For syntactic convenience we make use of the `do` notation, which can be translated into lambda calculus based on the following equalities [15]:

$$\begin{aligned}
&\texttt{do}\ \{e\} &&= e\\
&\texttt{do}\ \{e\texttt{;}stmts\} &&= e\ \texttt{>>=}\ \backslash_\ \texttt{->}\ \texttt{do}\ \{stmts\}\\
&\texttt{do}\ \{x\ \texttt{<-}\ e\texttt{;}stmts\} &&= e\ \texttt{>>=}\ \backslash x\ \texttt{->}\ \texttt{do}\ \{stmts\}
\end{aligned}$$

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 being wrapped 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 straightforward 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 of 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)
```

Alternatively, we can move monadification into the function of an application while increasing the number of lambda abstractions to be crossed by 1. This approach leads to simpler code. For the above example we obtain the following definition:

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

Monadification is more complicated in the case of recursive function definitions, because the corresponding recursive calls change their types. Not properly handled, these subexpressions would introduce type errors. Let us consider a simple example:[2]

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

If we simply wrap a `return` around 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 = \n -> do {x <- fM (n-1); return (n*x)}
```

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

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 problem case can be exemplified by the following function, in which a local variable `n` 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, the solution is to move the monadification into all branches of the `case` expression. Another reason for not lifting the expression `f (n-1)` is that in the original program, it is evaluated only when the second alternative of the `case` is matched. Lifting might cause this expression to be evaluated more than is 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 whole expression, operations on the `case` expression can be simply moved down to the bodies.

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)
```

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 kinds of operations are involved in monadifying a function definition:

- *Navigating*. Locate the return expressions 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.
- *Binding*. After locating return expressions, we identify recursive calls and bind them to (fresh) variables. Then we replace the recursive calls with these variables.
- *Wrapping*. After having removed recursive calls from return expressions, we apply `return`.

It is worth mentioning that if two or more recursive calls exist in a return expression and are to be bound to variables, we can choose different orders of binding. For a purely functional program, this is not a concern because the order does not affect the semantics, although it might have an impact on the efficiency in the case of parallel execution. But the effects in the underlying monad often depend on the order. If the user wishes to add extra actions to the program, the order in which variables are bound does matter. The implementation of the algorithm can either choose a predefined scheme or be parametrized to leave the choice up to the user. In any case, the user should take into consideration the relevant order. Two standard strategies for ordering bindings resulting from recursive calls are preorder and postorder, that is, arrange the bindings according to order in which the corresponding recursive calls have been encountered in a preorder or postorder traversal of the expression to be transformed. Since we are monadifying a group of functions with one and the same monad, it does not seem to be required to offer a different choice of order for different functions because the appropriate ordering is probably determined by the monad and not by the functions using the monad. We believe that such a choice should be a global parameter and not a parameter that affects individual monadification steps. Therefore, we will not parametrize the monadification algorithm presented in the next section by this aspect.

3. Automatic monadification

The general scenario is to monadify not just one, but a set of functions. These functions are monadified simultaneously so that all their definitions are navigated and calls to any of these functions will be bound. The function definitions in a program can be grouped into three categories:

- functions to be monadified;
- functions whose definitions contain calls to monadified functions, but which should have been originally not monadified;
- functions that are not affected by monadification.

The first set of functions is selected by the user. This selection then determines the remaining two function sets. A problem with functions in the second group is that monadification destroys type correctness since calls to monadified functions return values of type `m t` in contexts where values of type `t` are expected. There are two ways to deal with this problem: first, functions could be moved into the first set until the second set becomes empty. Second, values can be extracted from monads at all call sites. This approach is discussed further in Section 5. In this section we focus on the monadification of set of functions and assume for simplicity that the second group of functions is empty. Therefore, we view a program P as a collection of n function definitions that are to be monadified plus a set of definitions P' that are not affected by the monadification.

$$P = \{f_1 = e_1, \ldots, f_n = e_n\} \cup P'.$$

Our goal is to define an operator $\mathcal{M}$ that is applied to each function definition and yields the corresponding monadified version. To uniquely identify the result type to be monadified, $\mathcal{M}$ needs for each function the number of its parameters, which has to be ultimately

provided by the user. The function definitions together with the parameter information are called the *monadification context*.

$$F = \{(f_1 = e_1, k_1), \ldots, (f_n = e_n, k_n)\}.$$

We refer to the components of the ith context element as f_i^F, e_i^F, and k_i^F, respectively.

$\mathcal{M}$ is applied to P and yields the following monadified program $\hat{P}$:

$$\hat{P} = \mathcal{M}(P) = \{\hat{f}_1 = \mathcal{M}_F(e_1), \ldots, \hat{f}_n = \mathcal{M}_F(e_n)\} \cup P'.$$

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 6. We make the following assumption for the function definitions of $f_1, \ldots, f_n$: any call f_i^F is always applied to at least k_i^F arguments where k_i^F is the number of parameters for f_i^F; that is, there is no partial application of f_i^F that leaves calls to f_i^F "undersaturated" with arguments. This condition can be checked through the predicate $\forall i, j.\mathcal{F}(e_j^F, 0, f_i^F, k_i^F)$ where the judgment $\mathcal{F}(e, i, f, j)$ represents the fact that in a context where e is applied to i arguments, all references to f are applied to at least j arguments. Therefore, $\forall i, j.\mathcal{F}(e_j^F, 0, f_i^F, k_i^F)$ requires that in any function definition to be monadified, any function to be monadified is applied to at least k arguments. $\mathcal{F}$ is defined in Fig. 2. The parameter i is used to count the number of arguments that have already been provided by the context of the expression; it is needed to allow the checking of partial applications in the rule APP.

Note that requiring saturated function calls is not really a limitation of the algorithm because we can always supply additional arguments for undersaturated calls through η-expansion before applying the monadification.

3.1. 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 call to f with k arguments. This property is captured in the definition of the predicate $\mathcal{R}(f, k, e)$, which is inductively defined in Fig. 3.

In addition, we also need the information of whether or not e contains a recursive call to f as a *strict* subexpression, that is, e contains a call to f, but e itself is not a call to f. We write $\mathcal{S}(f, k, e)$ if e has this property.

$$\mathcal{S}(f, k, e) = e \neq \mathtt{f}\ e_1 \ldots e_m \wedge \mathcal{R}(f, k, e).$$

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 problem, we shall not lift such e' either. If e' can be safely lifted outside e, we say e' is a *liftable* subexpression of e and write $\mathcal{L}(e', e)$. For the definition of the liftability predicate $\mathcal{L}$ we employ the notion of contexts.

$$\text{CON}\ \frac{}{\mathcal{F}(c, i, f, k)} \qquad \text{VAR}\ \frac{i \geq k}{\mathcal{F}(f, i, f, k)} \qquad \frac{v \neq f}{\mathcal{F}(v, i, f, k)}$$

$$\text{APP}\ \frac{\mathcal{F}(e_1, i+1, f, k) \qquad \mathcal{F}(e_2, i, f, k)}{\mathcal{F}(e_1\ e_2, 0, f, k)}$$

$$\text{ABS}\ \frac{}{\mathcal{F}(\backslash f\texttt{->}e, i, f, k)} \qquad \frac{\mathcal{F}(e, 0, f, k) \quad v \neq f}{\mathcal{F}(\backslash v\texttt{->}e, i, f, k)}$$

$$\text{LET}\ \frac{}{\mathcal{F}(\texttt{let}\ f\texttt{=}e_1\ \texttt{in}\ e_2, i, f, k)} \qquad \frac{\mathcal{F}(e_1, 0, f, k) \quad \mathcal{F}(e_2, i, f, k) \quad v \neq f}{\mathcal{F}(\texttt{let}\ v\texttt{=}e_1\ \texttt{in}\ e_2, i, f, k)}$$

$$\text{CASE}\ \frac{\mathcal{F}(e, 0, f, k) \quad f \notin FV(p_i') \Rightarrow \mathcal{F}(e_i, i, f, k)}{\mathcal{F}(\texttt{case}\ e\ \texttt{of}\ \{p_1\texttt{->}e_1\texttt{;}\ \ldots\texttt{;}\ p_n\texttt{->}e_n\}, i, f, k)}$$

Fig. 2. Saturated function calls.

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

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

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

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

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

Fig. 3. Expressions containing calls to f.

A *context* is essentially an expression with a hole, written as $\langle\cdot\rangle$. We can apply a context C to an expression e, written as $C\langle e\rangle$, which denotes the expression obtained by filling the hole of C with e.

The syntax of contexts is given in Fig. 4.

The definitions for free and bound variables extend in a natural way from expressions to contexts.

With the context notation we can define $\mathcal{L}$ as follows:

$$\mathcal{L}(e', e) = (e = C\langle e'\rangle \wedge BV(C) \cap FV(e') = \varnothing \wedge C \neq C'\langle \texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}\ p_i\ \texttt{->}\ C''\texttt{;}\ \ldots\}\rangle).$$

$$
\begin{aligned}
C \ ::= \ & \langle\cdot\rangle \mid \texttt{\textbackslash}v\texttt{->}C \mid C\ e \mid e\ C \mid \texttt{let}\ v\texttt{=}C\ \texttt{in}\ e \mid \texttt{let}\ v\texttt{=}e\ \texttt{in}\ C \mid \\
& \texttt{case}\ C\ \texttt{of}\ \{p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}\ p_n\texttt{->}e_n\texttt{\}} \mid \\
& \texttt{case}\ e\ \texttt{of}\ \{p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}\ p_i\texttt{->}C\texttt{;}\ \ldots\ \texttt{;}\ p_n\texttt{->}e_n\texttt{\}}
\end{aligned}
$$

Fig. 4. The syntax of contexts.

$$
\begin{aligned}
\mathcal{N}_F(0, e) &= \mathcal{W}_F(e) \\
\mathcal{N}_F(n, \texttt{\textbackslash}v\texttt{->}d) &= \texttt{\textbackslash}v\texttt{->}\mathcal{N}_F(n-1, d) \\
\mathcal{N}_F(n, \texttt{case}\ e'\ \texttt{of}\ \{p_i\texttt{->}e_i\}) &= \\
& \begin{cases} \mathcal{N}_F(n, \texttt{\textbackslash}z\texttt{->case}\ e'\ \texttt{of}\ \{p_i\texttt{->}e_i\ z\}) & \text{if } \mathcal{R}(f_j, k_j, e') \\ \texttt{case}\ e'\ \texttt{of}\ \{p_i\texttt{->}\mathcal{N}_F(n, e_i)\texttt{\}} & \text{otherwise} \end{cases} \\
\mathcal{N}_F(n, e\ e') &= \mathcal{N}_F(n+1, e)\ e' \\
\mathcal{N}_F(n, e) &= \mathcal{N}_F(n, \texttt{\textbackslash}z\texttt{->}e\ z)
\end{aligned}
$$

Fig. 5. The definition of the $\mathcal{N}$ operator.

The last condition, which restricts the possible `case` contexts, guarantees that lifting does not increase the strictness of functions, at the expense of not being able to lift from RHSs of `case` rules. If increased strictness is not considered a problem, one could drop the last condition and thus extend the number of possible liftings, which will generally result in less complex monadified code.

3.2. Locating return expressions

We define a navigation operator $\mathcal{N}$ that moves monadification across lambda abstractions and into applications and eventually passes the result expressions found to the wrapping operator $\mathcal{W}$. $\mathcal{N}$ is also parametrized by a context F. More precisely, $\mathcal{N}_F(n, e)$ 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. When a lambda abstraction is lacking, η-expansion can be employed to generate the required abstraction. In the case of an application we can move $\mathcal{N}$ into the function. The definition of $\mathcal{N}$ is shown in Fig. 5. In all expressions, 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 $n > 0$. The variables f_j and k_j range over all f and k elements in the monadification context F. The metavariable e in the last line works as a catch-all case and matches constants, variables, and `let` expressions.

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

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

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

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

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. Using the simple form of η-expansion would force us to pass an application $(\texttt{case}\ e'\ \texttt{of}\ \{p_i\ \texttt{->}\ e_i\})\ z$ to $\mathcal{W}$. If no topmost call to some f_j in this expression is liftable, we have to resort to applying $\mathcal{N}$ again, with the `case` expression as the parameter, which leads to an infinite loop. By using the above transformation, we are able to avoid this non-terminating situation.

3.3. Wrapping return expressions

Having exposed a return expression, we need to change its type from t to $m\ t$. This is done by the operator $\mathcal{W}$, which takes a context F and the expression to be wrapped.

First, if there are no calls to any f_i^F inside e, e will be wrapped by a `return` unless e is a direct call to some f_i^F because in this case its type is already monadic. The condition is formally captured by the predicate $\forall i.\neg\mathcal{S}(f_i^F, k_i^F, e)$, and we get in this case

$$\mathcal{W}_F(e) = \begin{cases} \hat{f}_j\ e_1\ \ldots\ e_{k_j} & \text{if } \exists\ j : e = f_j^F\ e_1 \ldots e_{k_j} \\ \texttt{return}\ e & \text{otherwise.} \end{cases}$$

Otherwise, that is, if a topmost call to any f_i^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 contexts. The condition that e contains a liftable call to some f_i^F is expressed by the following formula:

$$e = C\langle f_i^F\ e_1\ e_2 \ldots e_{k_i^F}\rangle \wedge \mathcal{L}(f_i^F\ e_1\ e_2 \ldots e_{k_i^F}, e).$$

To additionally ensure that C locates a non-nested call to some f_i^F, that is, a call that is not nested inside a call to some other f_j^F, we also require

$$\nexists C' \neq \langle\cdot\rangle, j, e'_1\ e'_2 \ldots e'_{k_j^F} : C\langle z\rangle = C'\langle f_j^F\ e'_1\ e'_2 \ldots e'_{k_j^F}\rangle$$

(for a fresh variable z). Note that for $C' = \langle\cdot\rangle$, we allow the extraction of a nested call because this case covers the situation when e is a call to some f_j *and* contains a call to some f_i as a subexpression. In this case we have to lift and bind any such f_i. The recursive application of $\mathcal{W}$ eventually replaces f_j with $\hat{f}_j$. We combine these two conditions in the predicate $\mathcal{C}_\mathcal{L}$, which is defined as follows:

$$\begin{aligned} &\mathcal{C}_\mathcal{L}(e, C, f_i^F\ e_1 \ldots e_{k_i^F}) \\ &\quad = (e = C\langle f_i^F\ e_1 \ldots e_{k_i^F}\rangle \wedge \mathcal{L}(f_i^F\ e_1 \ldots e_{k_i^F}, e) \wedge \\ &\qquad \nexists C' \neq \langle\cdot\rangle, j, e'_1 \ldots e'_{k_j^F} : C\langle z\rangle = C'\langle f_j^F\ e'_1 \ldots e'_{k_j^F}\rangle). \end{aligned}$$

Now if $\mathcal{C}_\mathcal{L}(e, C, f_i^F\ e_1 \ldots e_{k_i^F})$ holds, we obtain the following definition for $\mathcal{W}$:

$$\begin{aligned} \mathcal{W}_F(e) = {} & \texttt{do}\ \{z \texttt{<-}\ \mathcal{W}_F(f_i^F\ e_1\ e_2\ \ldots\ e_k^F);\ \mathcal{W}_F(C\langle z\rangle)\} \\ & \text{where } z \notin \mathit{VARS}(e). \end{aligned}$$

Why do we require that $z \notin VARS(e)$? Because z must 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 any f_i^F in e is liftable, we have to scrutinize the syntactic structure of e (since e contains calls to some f_i^F, e cannot be a variable or constant).

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

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

Case $e = \backslash v \texttt{->} e'$. This is a case for which monadification fails as we have discussed in Section 2. A possible remedy is discussed in Section 5.

Case $e = e_0\ e_1\ \ldots\ e_m$ where e_0 is not an application, that is, $e_0 \neq e'e''$ for any e', e''. A general solution to this case is to apply $\mathcal{W}$ to $e_1\ \ldots\ e_m$, which makes their types monadic, bind them to fresh variables with respect to e, say $z_1, \ldots, z_m$, and also apply $\mathcal{N}$ to e_0. This requires the recursive application $\mathcal{N}_F(m, e_0)$ because e_0 is regarded as a function of m parameters and has to change its return type to a monadic type. Only when e_0 is a recursive `let` expression, that is, if $e_0 = \texttt{let}\ v\texttt{=}C\langle v\rangle\ \texttt{in}\ e'$, and contains a non-liftable call to an f_i^F, will $\mathcal{N}_F(m, e_0)$ 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. We capture this latter condition in the predicate $\mathcal{C}_{\bar{\mathcal{L}}}$, which is defined as follows:

$$\mathcal{C}_{\bar{\mathcal{L}}}(e_0, C, f_i^F\ e_1\ \ldots\ e_{k_i^F}) = (e_0 = C\langle f_i^F\ e_1\ \ldots\ e_{k_i^F}\rangle \wedge \neg\mathcal{L}(f_i^F\ e_1\ \ldots\ e_{k_i^F}, e)).$$

Hence, we get the following definition for $\mathcal{W}$ if $e_0 = \texttt{let}\ v\texttt{=}C\langle v\rangle\ \texttt{in}\ e' \Longrightarrow \neg\mathcal{C}_{\bar{\mathcal{L}}}(e_0, C, f_i^F\ e_1\ \ldots\ e_{k_i^F})$:

$$\mathcal{W}_F(e) = \texttt{do}\ \{z_1 \texttt{<-}\ \mathcal{W}_F(e_1); \ldots; z_m \texttt{<-}\ \mathcal{W}_F(e_m)\texttt{;}\ \mathcal{N}_F(m, e_0)\ z_1\ \ldots\ z_m\}.$$

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; see Section 1.1 and [1].

Case $e = \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2$. If $\forall i.\neg\mathcal{R}(f_i^F, k, e_1)$ holds, which means there are no calls to any f_i^F in e_1, we can wrap e by only wrapping e_2:

$$\mathcal{W}_F(e) = \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ \mathcal{W}_F(e_2).$$

In the case where x is not recursively defined, that is, $x \notin FV(e_1)$, e is treated like a β-redex:

$$\mathcal{W}_F(e) = \texttt{do}\ \{x \texttt{<-}\ \mathcal{W}_F(e_1); \mathcal{W}_F(e_2)\}.$$

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

1. If $\forall i \neg \mathcal{S}(f_i^F, k_i^F, e)$:

$$\mathcal{W}_F(e) = \begin{cases} \hat{f}_j\ e_1\ \ldots\ e_{k_j} & \text{if } \exists j : e = f_j^F\ e_1\ \ldots\ e_{k_j} \\ \texttt{return}\ e & \text{otherwise} \end{cases}$$

2. Otherwise, if $\mathcal{C}_{\mathcal{L}}(e, C, f_i^F\ e_1\ \ldots\ e_{k_i^F})$:

$$\mathcal{W}_F(e) = \texttt{do}\ \{z\ \texttt{<-}\ \mathcal{W}_F(f_j^F\ e_1\ \ldots\ e_k^F);\ \mathcal{W}_F(C\langle z\rangle)\}$$
$$\text{where } z \notin \mathit{VARS}(e)$$

3. Otherwise, if

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

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

(b) $e = e_0\ e_1\ \ldots\ e_m$ where $e_0 \neq e'\ e''$ for any e', e'' and $e_0 = \texttt{let}\ v\texttt{=}C\langle v\rangle\ \texttt{in}\ e' \Longrightarrow \neg\mathcal{C}_{\bar{\mathcal{L}}}(e_0, C, f_i^F\ e_1\ \ldots\ e_{k_i^F})$:

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

(c) $e = \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2$:

$$\mathcal{W}_F(e) = \begin{cases} \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ \mathcal{W}_F(e_2) & \text{if } \forall i \neg \mathcal{R}(f, k, e_1) \\ \texttt{do}\ \{x\ \texttt{<-}\ \mathcal{W}_F(e_1);\ \mathcal{W}_F(e_2)\} & \text{if } x \notin \mathit{FV}(e_1) \end{cases}$$

4. Otherwise, $\mathcal{W}$ fails.

Fig. 6. The definition of the $\mathcal{W}$ operator.

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

$$\mathcal{M}_F(e_i) = \mathcal{N}_F(k_i, e_i).$$

The definition of $\mathcal{W}$ is summarized in Fig. 6.

4. Correctness of monadification

In Section 2 we have identified two correctness criteria for the monadification of functions. Before we evaluate the monadification algorithm $\mathcal{M}$ presented in Section 3 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. (In practice, this should not be a common case because the return type is higher order.)
- 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 classifying the calls as not liftable (see the definition of $\mathcal{L}$ in Section 3.1) and eventually moving the monadification down into `case` expressions. However, the problem is generally always present in situations in which 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 might be evaluated after having been lifted. This generally causes an "increased strictness" of monadified functions. In these cases, the monadified functions are not complete. Recall the functions `c` and `r` from Section 2 and the monadified function `rM`, which is actually produced by our algorithm, repeated here for convenience.

```
rM x = do {y <- rM x; return (cy)}
```

The behavior of `rM` 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 (which is probably the case for most monads). Therefore, the argument `rM` is evaluated in this example, and inevitably `rM` would not terminate. We could try to circumvent this problem by avoiding binding the result of the function application and substituting `y` in the return expression by `rM x`:

```
rM' x = do return (c (rM' x))
```

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

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

Proposition 2. *Given* $f = e$, $\hat{f} = \mathcal{M}_F(e)$ *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, that is, when no recursive evaluation takes place. In the case of recursion, the inductive hypothesis and the left unit monad law can be applied to conclude soundness.

Although the results of monadification are not complete due to increased strictness, monadification produces complete results under eager evaluation.

Proposition 3. *Given* $f = e$, $\hat{f} = \mathcal{M}_F(e)$ *is a complete monadification of* f *under eager evaluation.*

The completeness follows from soundness whenever $\hat{f}$ is not less defined than f. Under eager evaluation this condition is satisfied.

Moreover, 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 a 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.

Finally, our monadification algorithm preserves the well typing of the monadified functions. Of course, external calls (from definitions in P') to monadified functions will no longer be type correct. Similarly, if instances of class member functions are monadified, the type signature of the class member must be monadified and so must all other instance definitions. However, if the set of functions to be monadified is closed with respect to mutual calls, the well typing of the whole program is ensured. Note that monadification is *not* type preserving since the types of some functions are changed to monadic type. However, the programs that are produced by monadification are type correct if the input programs are type correct. This property follows from the fact that our algorithm will eventually change all the uses of a symbol `f` whose definition changes from a type `t` to `m t`. In particular, the value of `f` will be bound to a variable, which has type `t`. Since this variable is substituted for the occurrence of `f` used, the well typing of the context is re-established. In those cases when a symbol cannot be changed, our algorithm stops with an error message and does not produce a possibly ill-typed program.

5. Runnable monads

In Section 3 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 as well as the class member definition and all other instance definitions for a class instance definition. 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 *runnable monads* provides a (partial) solution to both of these problems. A runnable monad is a monad that provides a `run` operation, which can be considered the operation dual to `return` and which has already been defined for some monads in Haskell. We can define runnable monads as a subclass of `Monad` as follows:

```
class Monad m => MonadRun m where
  run :: m a -> a
```

Basically, the purpose of `run` is to extract the value from a monad.

For example, the `Maybe` type constructor can be made an instance of `MonadRun` by simply defining `run (Just x) = x`. As another example we define a `MonadRun` 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 a state transformer can be made an instance of `Initializable` by providing an initial value. For instance, an initial value for integers could be `0`.

```
class Initializable s where
  initValue :: s
  initValue = undefined

instance Initializable Int where
  initValue = 0
```

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

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

The above definition assumes the following definition for the state transformer data type:

```
data ST s a = Trans (s -> (a,s))
```

In addition to the three basic monad laws (see Section 1.1 and [1]), a runnable monad should also satisfy the following inversion law [7]:

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

This law ensures that values injected into a monad can be recovered by `run`. It is easy to check that this law holds for the `Maybe` and the `ST` monads.

The value of the `run` 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 `run` to get a proper non-monadic value instead. With this approach, the function `f` from Section 2 can be monadified as follows:

```
fM = \x -> return (\y -> if x == 0 then y else run (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 `run` is used. Moreover, monadic actions cannot be inserted at these points. On the other hand, `run` provides a way to make monadification work in some cases.

Another use for `run` is to limit the effect that the monadification of a function has on the rest of a module. By wrapping `run` 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.

6. Adding actions to monads

So far, we have developed an algorithm for converting a group of functions into monadic form. In most cases the goal of this transformation is to add further code to these functions.

This code is sometimes also referred to as *monadic actions*. In the introductory example from Section 1, the conditional expression for handling divide-by-zero exceptions is such a monadic action. After discussing several approaches to integrating actions into monads in Section 6.1, we define the syntax and semantics of a language that can be used to express such updates in Section 6.2. In Section 6.3, we discuss the type safety of the approach.

6.1. Three options for adding actions

A simple, but inflexible, approach is to (always) insert one particular 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 function $\mathcal{W}_a$ in almost the same way as $\mathcal{W}$ except changing `return` e everywhere to a `>>= return` e. However, this solution is rather 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 to describe the insertion of actions. The idea of symbolic rewriting is to match a pattern against an expression to obtain a variable binding, then replace the expression with another pattern, with the variables substituted according to the variable binding. The following rewrite rule describes the adaptation that is needed for the divide-by-zero exception handling in the example from Section 1:

```
return (x `div` y) → if y == 0 then Nothing else return (x `div` y)
```

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 replaced by the conditional expression on the right-hand side, with x and y replaced by `i` and `j`, respectively.

The purely syntactic rewriting approach is limited by the fact that only literal occurrences of rewrite patterns can be identified. For example, if the program contains the expression `div i j`, the above rule would not match. We can imagine using rewriting modulo an equational theory to extend the applicability of matching, but the general problem remains.

Even though rewrite rules can incorporate context information through the use of metavariables in patterns, they cannot refer to parts of the context that are not being rewritten. However, this feature is sometimes very useful.

An example to illustrate this idea is the extension of the `eval` function we have seen earlier by means of an output trace [1]. First, we define an `Out` monad that couples an output string with the result value. Since 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++s',x') where Out (s',x') = f x
```

The call `return x` couples the 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 `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.

```
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 non-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 the automatic transformation more challenging.

To describe such a transformation we need a rewriting system that is capable of expressing rewrite rules with variable context dependencies. We can imagine the following rewrite rule for the task:

```
case e' of p -> ⟨return e → out (show p ++ '' =''  ++
                                 show e); return e⟩
```

The intended meaning is to match an enclosing context of a `case` expression, bind the metavariable p, and then perform the rewrite rule shown.

6.2. A context update language

The last approach leads to an update language that can express context-dependent rewritings. A context update is a rewrite rule that has been placed into a context. Contexts were defined in Fig. 4. We can apply a context to a rewrite rule to obtain a context update. A rewrite rule is either a simple rule $p_1 \rightarrow p_2$ or the composition of two rules $r_1; r_2$. Rewrite rules can contain metavariables that match any expression in the object program. Therefore, we extend the syntax of the object language defined in Fig. 1 to include metavariables for patterns (p) and expressions (e) that are used to describe contexts. The definition of contexts from Fig. 4 now refers to this extended form of expressions, which causes contexts to possibly contain metavariables.

$$
\begin{array}{lcl}
u & ::= & C\langle r\rangle \\
r & ::= & p \to p \mid r; r \\
p & ::= & c \mid v \mid m \mid p\ p \mid \texttt{case}\ e\ \texttt{of}\ \{q_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}q_n\texttt{->}e_n\texttt{}\} \\
q & ::= & c \mid q\ q \\
e & ::= & c \mid v \mid m \mid e\ e \mid \texttt{\textbackslash}v\texttt{->}e \mid \texttt{let}\ v\texttt{=}e\ \texttt{in}\ e \mid \\
 & & \texttt{case}\ e\ \texttt{of}\ \{p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\texttt{}\}
\end{array}
$$

Fig. 7. The syntax of the update language.

We require that contexts and patterns are linear in the sense that a metavariable does not appear more than once. However, within a context rewrite rule, a metavariable that occurs in the context might be reused in the patterns of rewrite rules where it is just replaced by the binding obtained from matching the context. We restrict patterns of rewrite rules to not include expressions that create bindings to prevent the illegal creation of unbound variables. The restricted form of case expressions

$$\texttt{case}\ e\ \texttt{of}\ \{q_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}q_n\texttt{->}e_n\texttt{}\}$$

disallows the use of variables in the left-hand sides of `case` rules (q) for the same reason. On the other hand, having case expressions allows us to express updates dealing with conditionals, such as the one for handling divide-by-zero errors.

When an update is applied to an expression, its context is matched against the expression and metavariables that occur in the context are bound to parts of the expression. After the context is matched, the rewrite rule in the hole is applied to the expression that matches the hole of the context. It will be applied recursively to all the subexpressions. All simple rules in a composition of rewrite rules are tried on the expression until one of them applies. A simple rule is applied as follows. The rule is instantiated by substituting those metavariables for which a binding is already provided by the matched context. Then the left-hand side of the rule is matched against the expression to obtain bindings for the remaining uninstantiated metavariables in the rule. These bindings are then used to replace the metavariables in the right-hand side to build the resulting expression. The semantics of the update language is defined in Fig. 8 by rules that define the judgment $[\![u]\!]_\sigma e = e'$, which expresses that the update u changes the expression e to e' under that mapping σ from metavariables to expressions. This context update language is general enough to express all the approaches discussed in Section 6.1.

The first three inference rules in the first line define the core rewriting semantics. The first rule in the second line defines how a composition of rewrite rules in a context is applied recursively. The next group of rules from line two to four define the recursive application of rules in the context. We write μr for a rule r that has to be recursively applied. The rules implement a "stop-top-down" strategy (see, for example, [9]) that does apply the rewrite rule recursively at possibly different places in the expression, but does not move recursively into expressions that have been rewritten. This is exactly the behavior we need

$$\frac{\sigma'(p_1) \gtrdot_{\sigma} e \quad \sigma(\sigma'(p_2)) = e'}{[\![p_1 \to p_2]\!]_{\sigma'} e = e'} \qquad \frac{r_1 \gtrdot_{@} e \quad [\![r_1]\!]_{\sigma} e = e'}{[\![r_1; r_2]\!]_{\sigma} e = e'} \qquad \frac{r_1 \not\gtrdot_{@} e \quad [\![r_2]\!]_{\sigma} e = e'}{[\![r_1; r_2]\!]_{\sigma} e = e'}$$

$$\frac{[\![\mu r]\!]_{\sigma} e = e'}{[\![\langle r \rangle]\!]_{\sigma} e = e'} \qquad \frac{[\![r]\!]_{\sigma} e = e'}{[\![\mu r]\!]_{\sigma} e = e'} \qquad \frac{[\![r]\!]_{\sigma} e_1 = e_1' \quad [\![r]\!]_{\sigma} e_2 = e_2'}{[\![\mu r]\!]_{\sigma}\, e_1\ e_2 = e_1'\ e_2'}$$

$$\frac{[\![r]\!]_{\sigma} e_1 = e_1' \quad [\![r]\!]_{\sigma} e_2 = e_2'}{[\![\mu r]\!]_{\sigma}\, \texttt{let } v\texttt{=}e_1 \texttt{ in } e_2 = \texttt{let } v\texttt{=}e_1' \texttt{ in } e_2'}$$

$$\frac{[\![r]\!]_{\sigma} e_i = e_i' \text{ for } 1 \le i \le n \quad [\![r]\!]_{\sigma} e = e'}{[\![\mu r]\!]_{\sigma}\, \texttt{case } e \texttt{ of } \{p_1\texttt{->}e_1; \ldots; p_n\texttt{->}e_n\} = \texttt{case } e' \texttt{ of } \{p_1\texttt{->}e_1; \ldots; p_n\texttt{->}e_n\}}$$

$$\frac{[\![u]\!]_{\sigma} e = e'}{[\![\backslash v\texttt{->}u]\!]_{\sigma} e = \backslash v\texttt{->}e'} \qquad \frac{[\![u]\!]_{\sigma\cup\sigma'} e = e' \quad e_0 \gtrdot_{\sigma'} e_1}{[\![u\ e_0]\!]_{\sigma} e\ e_1 = e'\ e_1} \qquad \frac{[\![u]\!]_{\sigma\cup\sigma'} e = e' \quad e_0 \gtrdot_{\sigma'} e_1}{[\![e_0\ u]\!]_{\sigma} e_1\ e = e_1\ e'}$$

$$\frac{[\![u]\!]_{\sigma\cup\sigma'} e = e' \quad e_0 \gtrdot_{\sigma'} e_1}{[\![\texttt{let } v\texttt{=}u \texttt{ in } e_0]\!]_{\sigma}\, \texttt{let } v\texttt{=}e \texttt{ in } e_1 = \texttt{let } v\texttt{=}e' \texttt{ in } e_1}$$

$$\frac{[\![u]\!]_{\sigma\cup\sigma'} e = e' \quad e_0 \gtrdot_{\sigma'} e_1}{[\![\texttt{let } v\texttt{=}e_0 \texttt{ in } u]\!]_{\sigma}\, \texttt{let } v\texttt{=}e_1 \texttt{ in } e = \texttt{let } v\texttt{=}e_1 \texttt{ in } e'}$$

$$\frac{\begin{array}{c}[\![u]\!]_{\sigma\cup\sigma'} e = e' \quad p_i \gtrdot_{\sigma_{i1}} p_i' \quad e_i \gtrdot_{\sigma_{i2}} e_i' \quad \sigma' = \cup_i(\sigma_{i1} \cup \sigma_{i2}) \\ [\![\texttt{case } u \texttt{ of } \{p_1\texttt{->}e_1; \ldots; p_n\texttt{->}e_n\}]\!]_{\sigma}\, \texttt{case } e \texttt{ of } \{p_1'\texttt{->}e_1'; \ldots; p_n'\texttt{->}e_n'\} \\ = \texttt{case } e' \texttt{ of } \{p_1'\texttt{->}e_1'; \ldots; p_n'\texttt{->}e_n'\}\end{array}}{}$$

$$\frac{\begin{array}{c}[\![u]\!]_{\sigma\cup\sigma'\cup\sigma''} e = e' \quad p_i \gtrdot_{\sigma_{i1}} p_i' \quad (e_i \gtrdot_{\sigma_{i2}} e_i' \text{ for } i \ne j) \\ \sigma' = (\cup_i \sigma_{i1}) \cup (\cup_{i\ne j} \sigma_{i2}) \qquad e_0 \gtrdot_{\sigma''} e_1 \\ [\![\texttt{case } e_0 \texttt{ of } \{\ldots p_j\texttt{->}u; \ldots\}]\!]_{\sigma}\, \texttt{case } e_1 \texttt{ of } \{\ldots p_j'\texttt{->}e; \ldots\} \\ = \texttt{case } e_1 \texttt{ of } \{\ldots p_j'\texttt{->}e'; \ldots\}\end{array}}{}$$

Fig. 8. The semantics of context updates.

for adding monadic actions because we generally have to replace more than one `return` expression that occurs, for example, in different branches of a `case` expression, but we do not want to replace recursive occurrences of `return` within other `return` expressions that are rewritten, because we want to add actions only for one particular monad. The remaining rules starting in line five are congruence rules that define the matching of the context. In the definition of the semantics we need a judgment of the form $r \gtrdot_{@} e$ that tells whether or not the update r applies to e. A rule r is applicable to e if we can apply the first inference rule in Fig. 8 to perform the update. We also use a judgment $p \gtrdot_{\sigma} e$ that infers under which binding of metavariables to expressions the pattern p matches the expression e, that is, $\sigma(p) = e$.

Note that the non-terminals e from Fig. 7 that are used in contexts might contain metavariables that are bound during the process of context matching described by the congruence rules. In contrast, the non-terminals e from Fig. 1 just represent expressions to which context rewrite rules are applied. These two different versions of non-terminals can be easily distinguished syntactically in the inference rules as follows. The non-terminals e used for representing contexts always occur (in judgments) inside of the semantics brackets $[\![\ ]\!]$, whereas the "ordinary" expression non-terminals e always occur outside of these brackets. This distinction has to be kept in mind when reading conditions, such as $e_0 \succcurlyeq_{\sigma} e_1$.

6.3. Type preservation

An important property of updates is type preservation, that is, when the updates are applied to well-typed expressions, they should ensure that any resulting expressions is of the same type as the original expression. A context update is type preserving if it satisfies the following two conditions:

(1) For any update $C\langle e_1 \to e_1'; \ldots; e_n \to e_n' \rangle$:

$$\bigcup_{1 \le i \le n} FV(e_i) \subseteq FV(C).$$

This condition ensures that no unbound variables are introduced by the update.

(2) For any rule $p_1 \to p_2$, the type of p_2 is the same as that of p_1. This condition ensures that we replace expressions with expressions of the same type. From the substitution lemma [20] it follows that the result of applying the update to the whole expression is type correct and has the same type.

7. Related work

Ralf Lämmel has employed program-transformation techniques to reduce the need for anticipation in developing reusable software [8]. One example he considered is the transformation of programs into monadic form. In his approach he has employed a program-transformation technique called *sequencing* [4] to flatten an expression into `let` expressions. This intermediate result is then transformed into a monadic computation. Lämmel's approach is type directed in contrast to our syntax-directed transformation method, which does not depend on the presence of typing information. Moreover, his transformation is given by inference rules in natural semantics style, whereas we describe a transformation algorithm. The type-directed approach seems to works quite well. Nevertheless, we favor a syntactic, algorithmic approach because it does not rely on the availability of the type information for all objects involved, although we exploit information about the number of arguments, which is derived from the type information. This fact simplifies the implementation of a monadification tool, and it also makes it more efficient, because the syntactical algorithm can always be directly applied, no matter whether or not other definitions (internal to the current module or external ones) have been changed since the last compilation of the program. In contrast, a type-directed approach has to generally re-parse and type-check all modules that have changed.

Another approach to monadification is known for a long time in the domain of compiler transformations [4,6]. The idea of these algorithms is to transform *all* functions and intermediate results in a program into monadic form. Hatcliff and Danvy give two transformations, a call-by-value and a call-by-name version. Below we present their call-by-value transformation,[3] adjusted to our abstract syntax from Fig. 1:

$$\begin{aligned}
M(c) &= \texttt{return}\ c \\
M(v) &= \texttt{return}\ v \\
M(\backslash v \texttt{->} e) &= \texttt{return}\ (\backslash v \texttt{->} M(e)) \\
M(e\ e') &= \texttt{do}\ \{\texttt{f} \texttt{<-}\ M(e);\ \texttt{x} \texttt{<-}\ M(e');\ \texttt{f x}\} \\
M(\texttt{let}\ v \texttt{=} e\ \texttt{in}\ e') &= \texttt{let}\ v \texttt{=} M(e)\ \texttt{in}\ M(e')
\end{aligned}$$

The call-by-name transformation is very similar. The only two differences are: (i) only recursively defined variables (by `let`) are wrapped by `return`, in contrast to all variables in the call-by-value transformation, and (ii) the monadification of a function argument is not bound to a variable, but directly monadified in place:

$$M(e\ e') = \texttt{do}\ \{\texttt{f} \texttt{<-}\ M(e);\ \texttt{f}\ \ M(e')\}.$$

While these algorithms are straightforward and simple, the translation of only a subset of functions into monadic form is complicated by the need to delimit the effect of monadic code to only the required places. One could imagine using one of these simple monadification algorithms to monadify all function calls in a program and then trying to get rid of unwanted monadifications by selecting the `Id` monad for all but the required places, and simplifying the resulting code by applications of the monad laws. This approach is possible since the monadification introduces monad expressions that are principally independent of one another and can therefore be of different monad types. However, this approach does not work in general. We illustrate the limitations of the simple algorithms by two examples. First, consider the following expression of type `(a -> b) -> a -> b`:

```
\f -> \x -> f x
```

The call-by-value translation yields the following monadified form:[4]

```
return (\f -> return (\x ->
                do {g <- return f; y <- return x; g y}))
```

This expression has the following type:

```
(Monad m1, Monad m2, Monad m3) =>
    m1 ((a -> m2 b) -> m3 (a -> m2 b))
```

If the goal is to monadify the original expression for $k = 2$ parameters, the resulting expression must have the following type:

```
(a -> b) -> a -> m b
```

[3] The algorithms given in [6] do not deal with data types or case expressions.

[4] The expression can be simplified to `return (\f -> return (\x -> do {f x}))`.

However, there is no choice for `m1`, `m2`, and `m3` to instantiate the above monadic type to this type. The call-by-name translation produces the following result:

```
return (\f->return (\x -> do {g <- f; g x}))
```

The type of this expression shown below suffers from the same limitations as the type of the call-by-value translation:

```
(Monad m1, Monad m2, Monad m3) =>
    m1 (m2 (a -> m2 b) -> m3 (a -> m2 b))
```

On the other hand, there is a simple monadification of the original expression that has the required type, namely the expression

```
\f -> \x -> return (f x)
```

which is exactly what our algorithm produces.

The approach of these simple algorithms to wrap *all* variables and constant into a monad requires the ability to choose monads independent of one another (`Id` versus `m`) to be able to selectively monadify a function at a particular result type. However, in cases when the result type of a higher-order argument (like `f` in the example) contributes to the overall result type of the function to be monadified, the monadification of that argument forces its result type to be wrapped by the same monad as the overall result type (here `m2`). This fact limits the applicability of these algorithms with respect to selective monadification essentially to first-order functions. Another limitation is revealed by recursive definitions, which will be illustrated next.

As we have already seen from the call-by-value and call-by-name translations, the main source of variation in the simple algorithm is the translation for application. One might think of still another version (not described in [6]) that can be obtained from the call-by-value transformation by wrapping the result of the application in an additional `return`:

$$M(e\ e') = \texttt{do \{f <- } M(e)\texttt{; x <- } M(e')\texttt{; return(f x)\}}$$

This variation ensures the independence of result types that was a problem with the other two translations. The application of this algorithm yields the following result that differs from the call-by-value result only by the additional `return` around `g y`:

```
return (\x -> return (\y ->
              do {g <- return f; y <- return x; return (g y)}))
```

The type of this expression is general enough to allow the instantiation to the desired result type:

```
(Monad m1, Monad m2, Monad m3) => m1 ((a -> b) -> m2 (b -> m3 b))
```

Alas, the transformation is unsound in the sense that it can produce ill-typed expressions. Consider again the example from Section 2, which can be monadified on its result type by our algorithm as follows:

```
f :: Monad m => Int -> Int -> m Int
f x y = if x == 0 then return y else f y (x-1)
```

The third algorithm produces the following definition (we have simplified the result according to the monad laws):

```
f x = return (\y->if x==0 then return y else return (f y (x-1)))
```

However, this definition contains a type error.

The call-by-value and call-by-name transformations given by Hatcliff and Danvy work well for this example and both yield the following translation:

```
f x = return (\y->if x==0  then return y
                           else do {g <- f y; g (x-1)})
```

The definition is type correct, but again due to there being too many monads involved the type is too constrained to be instantiated as required.

```
Monad m => Int -> m (Int -> m Int)
```

The examples shown demonstrate the limitations of the simple monadification algorithms with regard to the monadification of functions on selected parameters and that a tailor-made algorithm is required instead.

Other work on local monad transformations can be found in the context of CPS transformations. For example, the approach taken in [16] translates only those functions that have the same continuation. This, like all the other work on CPS transformations and monadic normal forms, is targeted at compiling functional languages. In contrast, our work is concerned with source code transformations of programs that are still to be used by programmers.

8. Conclusions

We have shown how function definitions can be automatically converted into a monadic form by a process called *monadification*. The transformation developed is safe since it preserves syntax and type correctness of the transformed program. Moreover, automatic monadification can preserve the semantics of the original program to a large degree. One could even argue that automatic monadification preserves the semantics as much as possible.

Monadification is an example of a generic program transformation that can be effectively used as a very general functional *refactoring* [5]. In many cases, such refactorings are only preparatory steps toward adding further functionality to programs. In the monadic setting this means adding monadic actions. We have addressed this task by employing a simple but effective context-dependent rewriting system, which preserves the types of the monadified program.

We have an initial prototype implementation of a monadification tool written in Haskell, which can be run as a stand-alone tool. Although quite a few desirable features are still missing, the current prototype can reproduce all the examples discussed in this paper. Future versions will contain an analysis phase that can detect closeness of a set of functions with respect to calls of monadified functions, the possibility to specify the addition of actions, and an intelligent “pretty-reprinting”, that is, a pretty-printing of changed code

parts that retains as much of the original program layout as possible. Ultimately, we plan to integrate our monadification tool into the Haskell refactoring tool "HaRe" [10]. Monadification could then be a refactoring that is offered through the menus of ordinary text editors.

A drawback of using Haskell as a metaprogramming language is that although Haskell can guarantee the syntactic correctness of transformed object programs, it cannot guarantee type correctness. Alternatively, we can implement monadification in a dedicated type-safe metaprogramming language that ensures the preservation of syntactic as well as type safety. To this end, we have developed an update calculus that can express type-preserving updates on functional languages such as Haskell [2,3]. The monadification operator defined in this paper will also serve as a benchmark for this update language.

Acknowledgements

We are very grateful to Ralf Lämmel who has provided many useful comments on this paper. We also thank the anonymous reviewers for their helpful comments.

References

[1] R.S. Bird, Introduction to Functional Programming Using Haskell, Prentice-Hall International, London, UK, 1998.
[2] M. Erwig, D. Ren, A rule-based language for programming software updates, in: 3rd ACM SIGPLAN Workshop on Rule-Based Programming, 2002, pp. 67–77.
[3] M. Erwig, D. Ren, Type-safe update programming, in: 12th European Symp. on Programming, LNCS, vol. 2618, 2003, pp. 269–283.
[4] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen, The essence of compiling with continuations, in: ACM Conf. on Programming Languages Design and Implementation, 1993, pp. 237–247.
[5] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.
[6] J. Hatcliff, O. Danvy, A generic account of continuation-passing styles, in: 21st ACM Symp. on Principles of Programming Languages, 1994, pp. 458–471.
[7] J. Hughes, The design of a pretty-printing library, in: Advanced Functional Programing, LNCS, vol. 925, 1995, pp. 53–96.
[8] R. Lämmel, Reuse by program transformation, in: G. Michaelson, P. Trinder (Eds.), Functional Programming Trends 1999, Intellect, 2000.
[9] R. Lämmel, J. Visser, A Strafunski application letter, in: 5th Symp. on Practical Aspects of Declarative Languages, LNCS, vol. 2562, 2003, pp. 357–375.
[10] H. Li, C. Reinke, S.J. Thompson, Tool support for refactoring functional programs, in: ACM SIGPLAN Haskell Workshop, 2003.
[11] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: 22nd ACM Symp. on Principles of Programming Languages, 1995, pp. 333–343.
[12] S. Mac Lane, Categories for the Working Mathematician, Springer-Verlag, 1971.
[13] E. Moggi, Notions of computation and monads, Information and Computation 93 (1) (1991).
[14] S. Peyton Jones, Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, in: T. Hoare, M. Broy, R. Steinbrüggen (Eds.), Engineering Theories of Software Construction, IOS Press, 2001, pp. 47–96.
[15] S. Peyton Jones, Haskell 98 Language and Libraries, Cambridge University Press, Cambridge, UK, 2003.
[16] J. Reppy, Local CPS conversion in a direct-style compiler, in: 3rd ACM Workshop on Continuations, 2001.
[17] P. Wadler, The essence of functional programming, in: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, 1992, pp. 1–14.

[18] P. Wadler, Monads for functional programming, in: Advanced Functional Programing, LNCS, vol. 925, 1995, pp. 24–52.
[19] P. Wadler, How to declare an imperative, ACM Computing Surveys 29 (3) (1997) 240–263.
[20] A.K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.