Designing Aspects for Side-Effect Localization

Kung Chen Jia-Yin Lin
National Chengchi University, Taiwan
{chenk,g9405}@cs.nccu.edu.tw

Shu-Chun Weng
National Taiwan University, Taiwan
b92103@csie.ntu.edu.tw

Siau-Cheng Khoo
National University of Singapore, Singapore
khoosc@comp.nus.edu.sg

Abstract

Computation performed in many typical aspects involve side effects. In a purely functional setting, adding such aspects using techniques such as monadification will generally lead to crosscutting changes. This paper presents an approach to provide side-effecting aspects for purely lazy functional languages in a user transparent fashion. We propose a simple yet direct state manipulation construct for developing side-effecting aspects and devise a systematic monadification scheme to translate the woven code to a purely monadic style functional code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimizaation

General Terms Design, Languages, Theory, Verification

Keywords Aspect-oriented programming, Side-effect, Lazy semantics, Monadification

1. Introduction

This paper concerns developing aspects that are inherently stateful in a purely functional setting. Many typical aspects, such as profiling, display refresh, and tracing, all need to maintain a state which depends on the execution state of the base program. We refer to them as *side-effecting aspects*. To work properly, these aspects need to monitor the state of the base program and update their internal state correspondingly. Although we can hide the hairy details of state manipulation by using monads [12], yet weaving a monadic aspect into a base program usually entails a comprehensive rewriting of the base program. We argue that it is better to support side-effecting aspects directly for localizing concerns and to automate the conversion of the woven code into monadic style via source-to-source transformation techniques. Such a process has been pioneered by Lämmel [9] and is referred to as *monadification* by Erwig and Ren[2].

In our previous work of AspectFun [1], an aspect-oriented lazy functional language with a Haskell-like syntax, we have developed a state-based implementation for control-flow related advice which uses a reader monad to keep function execution states (entry and exit) and employs a monadification step to convert the woven program. In this paper, we generalize this approach to the language level by providing constructs for writing side-effecting aspects directly and systematic monadification procedures for implementing them. Specifically, we propose to equip AspectFun aspects with user-defined mutable variables for performing side-effecting operations and extend its compiler with a more powerful monadification module based on cached state monad transformers to realize them.

At first sight, this may seem an easy task of just employing a state monad as the repository for mutable variables and lifting all non-monadic functions into monadic ones. This is simply not the case. First, lazy semantics makes the monadification process more difficult. Without proper control, monadification of a side-effecting aspect may lead to wrong order of evaluation and result in code that interferes with the execution of base program by unnecessarily forcing the evaluation of arguments or duplicating the evaluation. Therefore, besides reconciling the different needs of evaluation order between side-effecting operations and lazy evaluation, we must make an effort to ensure that the monadification itself will be non-interfering by preserving the lazy semantics. Second, the base program could be monadic already. Hence, we need to employ monad combination mechanisms such as monad transformers [10] to integrate both monadic aspects and monadic base program.

To illustrate the challenges, we present the problem of employing aspect to trace invocations of function calls in lazy functional language. As tracing involves side-effect, we introduce a state-aware monadic aspect to capture tracing function. In addition, output of tracing may force early evaluation of function arguments, leading to alteration of evaluation order of the base program. In Section 3, we present a general solution to such problem through introducing cached state.

The rest of the paper is organized as follows. Section 2 first reviews our base language, AspectFun, and describes the language constructs we design for writing side-effecting aspects. Section 3 identifies the issue of preserving laziness when moandifying such aspects with functional base programs and presents the monadification scheme we employ to implement side-effecting aspects in AspectFun. Section 4 illustrates how we can use monad transformers to handle the case when the base program is monadic and outlines a unified monadification scheme that accommodates both cases. Section 5 describes related work. Finally, Section 6 summarizes and discusses the future work.

2. Extending AspectFun with Side-Effecting Aspects

This section describes the language constructs we propose for developing side-effecting aspects in AspectFun. After giving a brief overview of AspectFun, we shall present the proposed extension for manipulating states in aspects along with some examples. To ease the presentation of the examples, we shall use pattern matching and freely employ functions available in the Haskell prelude and few Haskell constructs that are not yet implemented in AspectFun.

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

2.1 AspectFun Overview

Figure 1 shows the syntax of AspectFun. We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc). An AspectFun program is a sequence of top-level declarations followed by a main expression. Top-level definitions include global variables and function definitions, as well as aspects. An *aspect* declaration provides two specifications: An *advice*, which is a function-like expression named via the prefix $n@$; and a *pointcut designator*, around $\{\overline{pc}\}$, designates when the advice will be executed. In aspect-oriented programming [6], the specific program execution points to trigger an advice is called *join points*. Here, we focus on join points at function invocations. Thus a pointcut basically specifies a function whose invocations may trigger the execution of advice. The act of triggering an advice during a function application is called *weaving*. The argument variable arg is bound to the actual argument of the named function call.

Programs	π	::=	d in $\pi \mid e$
Declarations	d	::=	$x = e \mid f\ \overline{x} = e \mid f :: t \rightarrow t \mid$
			$n@$advice around $\{\overline{pc}\}\ (arg) = e$
Arguments	arg	::=	$x \mid x :: t$
Pointcuts	pc	::=	$ppc \mid pc + cf \mid pc - cf$
Primitive PC's	ppc	::=	$f\ \overline{x} \mid$ any $\mid$ any$\backslash[\overline{f}] \mid n$
Cflows	cf	::=	cflow$(f) \mid$ cflow$(f(_ :: t)) \mid$
			cflowbelow$(f) \mid$ cflowbelow$(f(_ :: t))$
Expressions	e	::=	$c \mid x \mid$ proceed $\mid \lambda x.e \mid e\ e \mid$
			if e then e else $e \mid$ let $x\ =\ e$ in e
Types	t	::=	*Int* $\mid$ *Bool* $\mid a \mid t \rightarrow t \mid [t]$
Predicates	p	::=	$(f : t)$
Advised Types	ρ	::=	$p.\rho \mid t$
Type Schemes	σ	::=	$\forall \bar{a}.\rho$

Figure 1. Syntax of the AspectFun Language

Advice may be executed *before*, *after*, or *around* a join point. Specifically, *around* advice is executed in place of the indicated join point, allowing the call to the advised function to be replaced. A special keyword `proceed` may be used inside the body of *around* advice. It is bound to the function that represents "the rest of the computation" at the advised join point. As both *before* advice and *after* advice can be simulated by *around* advice that uses `proceed`, we only need to consider *around* advice in this paper.

Precisely, a pointcut, pc, may be either a primitive pointcut or a composite pointcut. A primitive pointcut, ppc, specifies a function (f) or an advice name (n) the invocations of which will be advised. A sequence of pointcuts, $\overline{pc}$, indicates the union of all the sets of join points selected by each. A primitive pointcut can also be a catch-all keyword any. When used, the corresponding advice will be triggered whenever a function is invoked. Name based primitive pointcuts can be composed with control-flow based pointcuts (cflow and cflowbelow) to form composite pointcuts, which inspects the run-time stack of function execution.

In Figure 1, the argument variable arg may contain a *type scope*, the t in $x :: t$. When such a type scope is present, the applicability of a piece of advice is bounded by its pointcut as well as its type scope. Specifically, when the function in the pointcut is polymorphic, a type scoped argument only matches executions of the function with arguments of types that are subsumed by their scope. This is particularly useful as many functional languages are polymorphically typed.

Expressions in AspectFun are pretty standard and are evaluated with a lazy semantics. As mentioned above, the special keyword `proceed` may be used inside the body of around advice. When applied, `proceed` will resumes the execution of advised functions or other advice that also designates the same function as its join point, as in AspectJ.

AspectFun is polymorphically and statically typed. It introduces a concept of *advised types* [13] that extend types with predicates of the form $(f : t)$. Advised types are inspired by Haskell's type classes and are used to capture the need of advice weaving based on type context. As a result, AspectFun is able to statically resolve type scopes on pointcut and weave aspects into base program. We have built a compiler that employs a type-directed static weaver to translate an AspectFun program into executable Haskell code. The readers are referred to [1] for more details.

2.2 Side-Effecting Aspects

We now describe how we extend AspectFun to support side-effecting aspects. The essential construct we add to AspectFun is user-defined *mutable variables* declared at top-level. We use `var` as the keyword to begin such a declaration. The precise syntax is as follows.

Declarations d ::= ... | var $id :: t\ = e$

Such mutable variables are declared with a monomorphic and ground type, t, and an optional initializing expression, e. Side-effecting aspects employ mutable variables to keep pertinent state information. For example, the following declaration introduces a mutable variable `profileMap` whose type is `Map.Map String Int` with initial value `empty`[1]. Later, we shall use it to develop a profiling aspect.

```
var profileMap :: Map.Map String Int = Map.empty
getProfileMap :: Map.Map String Int
setProfileMap :: Map.Map String Int -> ()
```

Also associated with each mutable variable declared is a pair of getter and setter functions. With them, the user can write helper functions to develop side-effecting aspects. For example, the following function, `incProfile`, will increase the calling count of a function whose name is passed to it.

```
incProfile :: String -> ()
incProfile fname =
  let! pMap = getProfileMap     --strict evaluation
   in let newMap =
             case of Map.lookup fname pMap of
              Nothing -> Map.insert fname 1 pMap
              Just v  -> Map.insert fname (v+1) pMap
      in setProfileMap newMap
```

Here we provide another construct, `let!`, for users to override the default lazy evaluation semantics of the normal `let` expressions. In other words, the expression used in the local definition of a `let!` expression will be evaluated eagerly unlike its `let` counterpart. This is necessary because operations such as `getProfileMap` are state-aware and require the current state snapshot before continuing the subsequent evaluations. This will become clearer when monadification transformation is introduced later.

Besides mutable variables, output is also an important element for side-effecting aspects such as tracing aspects. Hence we also provide a function, `putMsg :: String -> String -> ()`, for performing output in aspects. The first string parameter is the name of aspect which puts the second parameter (the message) into an

[1] The `Map` is an alias of the `Data.Map` in Haskell prelude.

internal buffer. Together with the getter and setter functions, they form the *state API* of an aspect. Finally, in a side-effecting aspect, it is often required to serialize the execution of state-affecting expressions. Hence we also add sequencing expressions, $(e_1; e_2)$, to AspectFun.

Given the above background, we shall present three examples to illustrate our side-effecting aspects. The first example declares two variables, `profileMap` and `memoMap`, and then defines two aspects, `profiler` and `memoFib`, to profile and memoize the function `fib` for computing the Fibnoacci numbers.

Example 1

```
var profileMap :: Map.Map String Int
var memoMap    :: Map.Map Int Int

fib n = if n <= 1 then 1
        else fib (n - 1) + fib (n - 2) in
--aspect 1
profiler@advice around {fib} (arg) =
               incProfile "fib"; proceed arg in
--aspect 2
memoFib@advice around {fib} (arg) =
  case lookupCache arg of
     Just v -> v
     Nothing -> let! v = proceed arg
                 in insertCache arg v; v  in
fib 10
```

Note that, in the above code, we use `let!` expression to get the return value of `proceed`. The reason is quite obvious. Since an invocation of `proceed` inside advice is defined to resume the execution of the advised function as well as other (possibly side-effecting) aspects that also designate the same function as their join point, we should ensure that the `proceed` call is indeed executed immediately so that the state maintained by its side-effecting aspects is consistent.

The second example is a tracing aspect for the tail recursive factorial function, adapted from Kishon's thesis work on program monitoring [7].

Example 2

```
fac n acc = if n == 0 then acc
                      else fac (n - 1) (n * acc)

var indent :: String = ""
tracer@advice around{fac, (*)} (arg) =
\arg2 ->
  let! ind = getIndent in
    setIndent ("| " ++ ind);
    putMsg "tracer" (ind++tjp++" receives ["++
           show arg ++ ", " ++ show arg2 ++ "]");
    let! result = proceed arg arg2 in
      setIndent ind;
      putMsg "tracer" (ind++tjp++" returns " ++
                       show result);
      result
```

Here the state to be maintained is the indentation string, kept by the variable, `indent`. The `tracer` aspect traces the execution of the functions, `fac` and `(*)`, respectively. The `tjp` is a keyword for referring to the function current being advised, namely the current join point[2]. The advice simply traces the arguments passed to and the results returned from the advised functions. Unfortunately, the above aspect declaration does not render proper tracing of the factorial function, as its execution interferes with the lazy semantics of the factorial function. We shall discuss this in detail in the next section.

The last example shows a case in which the base program to be advised involves its own state manipulation and is thus written in a monadic style too [3]. We illustrate this by refactoring the monadic version of the "display update" example presented by Hofer and Ostermann [5].

The context of this "display update" example [6] is a simple figure editor that manipulates typical shapes such as points and lines. Any update done on such shapes will trigger an action for display refresh. It is a model example of crosscutting concerns (i.e., display refresh) that can be nicely handled by aspect-oriented programming. In their work, Hofer and Ostermann aim to show a simulation of aspect-oriented programming with monads. To achieve this goal, besides introducing `IO` monad for state manipulation, they also introduced an additional monad, `MonadIO`, and an overloaded `withStateChange` operator to implement the crosscutting concern of display refresh.

By contrast, we use side-effecting aspects to separate the concern of display refresh from the base module of shape manipulation; thus the base module only needs to use the `IO` monad to support shape updates. We list below the main fragments of the refactored code.

Example 3

```
newtype Point = P (IORef (Int, Int))
newPoint :: Int -> Int -> IOPoint ...
setPointX, setPointY :: Point -> Int -> IO () ...
movePointBy::Point-> Int -> Int -> IO () ...
newtype Line = L (IORef (Point, Point))
newLine :: Point -> Point -> IO Line ...
getLineP1, getLineP2 :: Line -> IO Point ...
moveLineBy :: Line -> Int -> Int -> IO ()
...
sample :: Line->IO()--a test case
sample l = moveLineBy l 7 (-9)
data DisplayObject=forall a.Displayable a =>
                                  DisplayObject a
--user variable
var displayObject ::DisplayObject =
                     DisplayObject EmptyDisplay

--aspect 1:before advice
initDisplay@advice around{sample} (l) =
   setDisplayObject (DisplayObject l); proceed l

--aspect 2: after advice
moveUpdate@advice around{movePointBy,moveLineBy
                   -cflow(updateDisplay)} (arg) =
  \dx -> \dy -> updateDisplay (proceed arg dx) dy

--aspect 3: after advice
setUpdate@advice around{setPointX,setPointY
                  -cflow(updateDisplay)} (arg) =
  \newVal -> updateDisplay (proceed arg) newVal

updateDisplay f n = let a = f n in refreshDisplay; a
refreshDisplay:: IO ()
refreshDisplay = let DisplayObject d=getDisplayObject
                 in display d; putStrLn ""
```

[2] AspectFun does not support the `tjp` facility yet. Nevertheless, we can write two almost identical aspects to trace `fac` and `(*)`, respectively.

[3] Currently, AspectFun does not yet support base programs written in monadic style. The code presented here is written by hand in accordance with our existing design of an Haskell-enhanced version of AspectFun.

Here the user variable, `displayObject`, is the object to display, which is either a line or a point. The function `sample` is a test case. There are three aspects. The first one, `initDisplay` sets the object to display before running the test case, `sample`. The other two aspects, `moveUpdate` and `setUpdate`, trigger the display refresh operation when a point or a line is updated. They both have composite pointcuts: Besides the update functions, they include a control-flow based pointcut, `-cflow(updateDisplay)`, which ensures that the advice code will *not* be triggered when the `updateDisplay` function is still in execution, thus preventing repeated display refresh during a single update operation.

3. Monadifying Aspect Programs

The first step of AspectFun compilation is to weave aspects into the base program, thus producing an integrated program, which we call a *woven code*. In the presence of side-effecting aspects, it is necessary for the woven code to be generated in monadic style, in order to retain its functional purity.

This section describes how we enhance the compiler of AspectFun to support side-effecting aspects through incremental monad computation. First, we present a general framework for monadifying functional expressions in a non-strict evaluation context. Next, we illustrate the method to incorporate a state monad into the framework so that we can also monadify those state-aware functions used by side-effecting aspects properly. Finally, we identify further requirements for preserving laziness by examining the tracing aspect example, and describe an extension of the framework to fulfill these requirements.

3.1 Monadifying Pure Expressions

We begin with a general framework for monadifying functional expressions. Like the pioneering work of Lämmel [9], our monadification transformation also consists of two major steps, namely A-normalization [3] and monad introduction.

3.1.1 A-Normalization

Given an expression, the A-normalization step converts it into a sequential version according to the call-by-value sequencing. Such normalized expressions, called A-normal forms, are a popular intermediate representation used in compilers for functional languages. Essentially, in an A-normal form, function applications and the condition parts of if-expressions are all flattened by let-expressions.

Let's take the profiling of the `fib` function presented before as an example. The input to our A-normalization step is the following woven Haskell code generated by the AspectFun compiler.

```
let profiler proceed arg = incProfile "fib";
                           proceed arg in
let fib n = if n <= 1 then 1
            else profiler fib (n - 1) +
                 profiler fib (n - 2) in
profiler fib 10   --main
```

The aspect, `profiler`, becomes an ordinary function with an additional parameter, `proceed` that captures the continuation to the advised function. Moreover, all invocations of the `fib` function are now left to the `profiler` function.

After A-normalization, the above profiler program is converted to the following code.

```
let profiler proceed arg = incProfile "fib";
                           proceed arg in
let fib n = let nleq1 =  n <= 1 in
             if nleq1 then 1
             else let nm2 = n - 2 in
                    let fibm2 = profiler fib nm2 in
                      let nm1 = n - 1 in
                        let fibm1 = profiler fib nm1 in
                          (+) fibm1 fibm2    in
profiler fib 10  --main
```

3.1.2 Monad Introduction

The second step of the monadification transformation is monad introduction. This aims to lift computations in the input expressions to a designated monad, $(M, \mathsf{return}, \gg)$. Its essence can be captured by the monadification operator $\mathcal{M}$ defined over types as follows.

$$\mathcal{M}(t_1 \to t_2) \quad \Rightarrow \quad \mathcal{M}(t_1) \ \to \ \mathcal{M}(t_2) \tag{1}$$

$$\mathcal{M}(a) \quad \Rightarrow \quad M\ a \tag{2}$$

where rule (1) applies to functional types and rule (2) applies on non-functional types.

We note that the monadification schemes proposed by Lämmel [9] and Erwig and Ren [2] do not lift arguments of functions to monadic space. By contrast, we lift function arguments to monadic space in order to capture the computation of arguments inside the aspect monad and thus support the non-strict evaluation semantics of AspectFun.

The concrete steps for lifting computations to monadic space are formalized as a rewriting function, $[\![\cdot]\!]_\Gamma$, that converts an A-normalized expression, e, to a monadified version, e^M, over the designated monad, M. The subscript Γ is a type environment containing the types for the free identifiers occurring in e. Figure 2 displays the definition of $[\![\cdot]\!]_\Gamma$, implicitly parameterized over a monad M.

The key parts of the moandification function $[\![\cdot]\!]_\Gamma$ can be summarized as follows. Constants and primitive functions are lifted to the monadic space by the `return` operation and the `liftM` operation of the designated monad, respectively. For `if` expressions, since their boolean conditions had been turned into a monadified expression, we need to apply a do-binding to trigger its evaluation. The remainding cases are quite straightforward.

The following code shows the monadified version of the `fib` function defined earlier[4].

```
fibM :: M Int -> M Int
fibM n =
   do let leq_n_one = (liftM2 (<=)) n (return 1)
      nleq1 <- leq_n_one
      if nleq1 then return 1
      else do let nm2 = (liftM2 (-)) n (return 2)
              let fibnm2 = profilerM fibM nm2
              let nm1 = (liftM2 (-)) n (return 1)
              let fibnm1 = profilerM fibM nm1
              (liftM2 (+)) fibnm1 fibnm2
```

$[\![\cdot]\!]_\Gamma$ as defined possesses the following two good properties. First, its output expression has the desired monadified type.

Proposition 1 (Type Lifting) *Given an A-normalized expression e and a type environment Γ, if $\Gamma \vdash e : t$, then, regardless of the underlying monad,*

$$\mathcal{M}(\Gamma) \vdash [\![e]\!]_\Gamma : \mathcal{M}(t)$$

where $\mathcal{M}(\Gamma)$ is the pointwise application of $\mathcal{M}$ to the type part of all bindings in Γ.

Second, it preserves the semantic value of the input expression. This can be specified by replacing the underlying monad with the Identity monad.

Proposition 2 (Semantics Preserving) *Given an A-normalized expression e and a type environment Γ, if $\Gamma \vdash e : t$ and $e \longmapsto^*_\beta v$,*

[4] A fold over the do-bindings is performed to polish the code.

$$
\begin{array}{lllll}
 & [\![.]\!]_\Gamma & : & e \longrightarrow e^M & \\
(\textsc{Const}) & [\![c]\!]_\Gamma & = & \mathsf{return}\ c & \\
(\textsc{Prim}) & [\![p]\!]_\Gamma & = & \texttt{liftMn}\ p \quad \textit{where}\ \text{n is the arity of primitive function } p & \\
(\textsc{Var}) & [\![x]\!]_\Gamma & = & x & \\
(\textsc{If}) & [\![\mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_\Gamma & = & \textit{let}\ e_1^M\ =\ [\![e_1]\!]_\Gamma & \\
 & & & \quad\ e_2^M\ =\ [\![e_2]\!]_\Gamma & \\
 & & & \textit{in if}\ isConst(a)\ \textit{then}\ \mathsf{if}\ a\ \mathsf{then}\ e_1^M\ \mathsf{else}\ e_2^M & \\
 & & & \quad \textit{else}\ \mathsf{do}\ \{x' \leftarrow\ a; \mathsf{if}\ x'\ \mathsf{then}\ e_1^M\ \mathsf{else}\ e_2^M\} & \\
 & & & \textit{where}\ x'\ \text{is fresh} & \\
(\textsc{Lam}) & [\![\lambda x.e]\!]_\Gamma & = & \lambda x.[\![e]\!]_\Gamma & \\
(\textsc{App}) & [\![e\ a]\!]_\Gamma & = & [\![e]\!]_\Gamma\ [\![a]\!]_\Gamma & \\
(\textsc{Let}) & [\![\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2]\!]_\Gamma & = & \textit{let}\ e_1^M\ =\ [\![e_1]\!]_\Gamma & \\
 & & & \quad\ e_2^M\ =\ [\![e_2]\!]_\Gamma & \\
 & & & \textit{in}\ \mathsf{do}\ \{\mathsf{let}\ x = e_1^M;\ e_2^M\} & \\
 & \textit{where}\ a \in \text{Atoms} & ::= & c \mid x &
\end{array}
$$

Figure 2. Monadification Function

then `runIdentity`$([\![e]\!]_\Gamma) = v$*, where the underlying monad,* M*, is set to the* `Identity` *monad and* `runIdentity` *is the standard function for extracting values from the computation of* $[\![e]\!]_\Gamma$*.*

3.2 Monadifying State-Aware Functions

Given the monadification framework presented above, we now proceed to specialize it by introducing state monads to support side-effecting aspects. The essence of our scheme is a *state monad* that encapsulates state information maintained by those state-aware functions assisting the user in developing side-effecting aspects. Specifically, state information consists of two parts: a user variable record and an output buffer. We shall refer to them as *aspect state* and the state monad encapsulating them as *aspect monad*.

Since the specific content of the user variable record depends on individual program, we provide the following generic state monad, `GM` based on the standard state monad of Haskell. The `putMsgM` function extracts its string arguments out of the monad and appends them to the internal output buffer. In addition, two utility functions, `getUserVar` and `modifyUserVar`, are supplied to facilitate the generation of the monadified versions of state accessor functions for user variables.

```
type GM v = State (v, OutputBuf)
     -- v is a program-specific type
OutputBuf = [(String, String)]--(advName,msg) pair
putMsgM :: GM v String -> GM v String -> GM v ()
putMsgM a m =
        do a' <- a; m' <- m
           modify $ \(u, ms) -> (u, (a', m'):ms)
getUserVar :: GM v v
getUserVar = do (uv,_) <- get
                return uv
modifyUserVar :: (v -> v) -> GM v ()
modifyUserVar trans = modify $ \(u, s) -> (trans u, s)
```

The definition of the aspect monad for a specific program is derived from its declarations of user variables. Take the profiler aspect as an example, the enhanced AspectFun compiler will generate the following definition of a specialized aspect monad and the associated accessor functions for its user variable, `profileMap`.

```
--one variable one field
data UserVar = U {profileMap::Map.Map String Int}
--aspect monad
type M = GM UserVar
--state accessor functions
getProfileMapM :: M (Map.Map String Int)
getProfileMapM = getUserVar>>= \u -> return $ profileMap u
setProfileMapM :: M (Map.Map String Int) -> ()
setProfileMapM var =
  do var' <- var
     modifyUserVar $ \u -> u{ profileMap = var' }
```

Functions such as `getProfileMapM` defined above, as well as those which invoke them are state-aware; their invocations mostly require immediate access to the underlying state monad. Yet, as mentioned before, AspectFun is a lazy language. Hence we provide let!-expression to enable the user to override the default lazy evaluation semantics when applying such state-aware functions. In the context of monadic space, this means that we have to trigger the computation embodied in the definition of a let!-expression. Hence we need to enhance the monadification with a new case to handle let!-expressions as follows.

$$
\begin{array}{l}
[\![\mathsf{let!}\ x\ =\ e_1\ \mathsf{in}\ e_2]\!]_\Gamma = \\
\quad \textit{let}\ e_1^M\ =\ [\![e_1]\!]_\Gamma \\
\quad\quad\ e_2^M\ =\ [\![e_2]\!]_\Gamma \\
\quad \textit{in}\ \ \mathsf{do}\ \{x' \leftarrow e_1^M;\ \mathsf{let}\ x = \mathsf{return}\ x';\ e_2^M\} \\
\quad \textit{where}\ x'\ \text{is a fresh identifier}
\end{array}
$$

Here the do-binding triggers the computation captured in the monadified expression, e_1^M, ensuring that the state information fetched in the expression e_1^M is the current snapshot.

Alternatively, we could leave the monadification function unchanged by treating the let!-expressions as a syntactic sugar as follows.

$$\mathsf{let!}\ x = e_1\ \mathsf{in}\ e_2 \rightarrow \mathsf{let}\ x = e_1\ \mathsf{in}\ (x; e_2)$$

However, as will become clear in the next sub-section, this approach will cause many unnecessary operations in a lazy evaluation context.

As to the sequencing expression, $e_1; e_2$, its monadification is rather straightforward. We simply wrap the monadification of each component expression within a do-expression as follows.

$$[\![e_1;\ e_2]\!]_\Gamma\ =\ \mathsf{do}\ \{[\![e_1]\!]_\Gamma\ ;\ [\![e_2]\!]_\Gamma\}$$

Let us revisit the profiling example defined previously to illustrate the enhanced monadification function. The `fib` function does not use any let!-expressions, so its monadified version remain un-

changed. The `profiler` aspect employs a sequencing expression. Hence its body becomes a do-expression after monadification. Finally, the monadification of the helper function `incProfile` is more involved, because of the use of the let!-expressions. Applying the enhanced monadification function to it, we obtain the following monadified version for the `incProfile` function.

```
profilerM :: (M Int -> M Int) -> (M Int -> M Int)
profilerM proceed arg = do incProfileM (return "fib")
                           proceed arg
incProfileM fname =
   do pMap'  <- getProfileMapM   -- access user variable
      let pMap = return pMap'
      let lookupResult' = (liftM2 Map.lookup) fname pMap
      lookupResult <- lookupResult'
      let newMap = case lookupResult of
          Nothing -> (liftM3 Map.insert)
                       fname (return 1) pMap
        (Just v') -> do let v = return v'
                        let np1 = (liftM2 (+)) v (return 1)
                        (liftM3 Map.insert)
                                 fname np1 pMap
      setProfileMap newMap
```

3.3 Preserving Laziness

With the introduction of the specialized state monad and the eager approach of monadifying let!-expressions, we can support the desired order of evaluation required for side-effecting aspects. However, monadification brings along its own "side effect". Whereas an expression may be evaluated once under lazy semantics despite being referred to (by an identifier) multiple times in a program, its monadified counterpart may be evaluated at every reference. When the monad involved is state-aware or can perform IO operation, multiple evaluations of the monad can interfere eccentrically with the underlying base program that obeys lazy semantics. This issue emerges when we attempted to monadify the tracing aspect of Example 2. As shown in [7], according to the lazy semantics, the tracing result of `(fac 3 1)` should be

```
fac receives [3, 1]
| fac receives [2, 3]
| | fac receives [1, 6]
| | | fac receives [0, 6]
| | | | times receives [1, 6]
| | | | | times receives [2, 3]
| | | | | | times receives [3, 1]
| | | | | | times returns 3
| | | | | times returns 6
| | | | times returns 6
| | | fac returns 6
| | fac returns 6
| fac returns 6
fac returns 6
```

However, our monadified tracing aspect of `fac` does not yield the same result. Consider the following code for the tracing example generated by our monadification function[5].

```
tracerFacM :: (M Int -> M Int -> M Int) ->
              (M Int -> M Int -> M Int)
tracerFacM  proceed arg arg2 =
  do getIndentResult  <- getIndentM
     let ind = return getIndentResult
     let ind' = (liftM2 (++)) (return "| ") ind
     setIndent ind'
     let show_arg2 = (liftM show) arg2
     let str_1 = (liftM2 (++)) show_arg2 (return "]")
     let str_2 = (liftM2 (++)) (return ",") str_1
     let show_arg = (liftM show) arg
     let str_3 = (liftM2 (++)) show_arg str_2
     let str_4 = (liftM2 (++)) (return "fac receives [")
                               str_3
     let str_5 = (liftM2 (++)) ind str_4
     putMsgM (return "tracerFacM") str_5
     proceedResult  <- proceed arg arg2
     let result = return proceedResult
     setIndent ind
     let s_result = (liftM show) result
     let str_6 = (liftM2 (++)) (return "fac returns ")
                               s_result
     let str_7 = (liftM2 (++)) ind str_6
     putMsgM (return "tracerFacM") str_7
     result
facM :: M Int -> M Int -> M Int
facM n acc =
  do let eq_n_zero = (liftM2 (==)) n (return 0)
     neq0  <- eq_n_zero
     if neq0 then acc
     else do let nmacc  = (tracerMulM (liftM2 (*)) n acc
             let nm1  = (liftM2 (-)) n (return 1)
             (tracerFacM facM) nm1 nmacc
mainM =  (tracerFacM facM) (return 3) (return 1)
```

Running the above monadified tracing program with `(facM (return 3) (return 1))` yields the following incorrect trace.

```
fac receives [3, 1]
| | times receives [3, 1]
| | times returns 3
| fac receives [2, 3]
| | | | times receives [3, 1]
| | | | times returns 3
| | | times receives [2, 3]
| | | | times receives [3, 1]
| | | | times returns 3
| | | times returns 6
| | fac receives [1, 6]

⋮

| | | | | | times receives [3, 1]
| | | | | | times returns 3
| | | | | times receives [2, 3]
| | | | | | times receives [3, 1]
| | | | | | times returns 3
| | | | | times returns 6
| | | | times returns 6
| | | fac returns 6
| | fac returns 6
| fac returns 6
fac returns 6
```

From the generated trace, we can see that some expressions, such as `times 3 1`, are evaluated more than once and in wrong orders. In other words, the monadified tracing program obtained not only changes the order of evaluation but also duplicates the evaluation of some expressions, thus delivering wrong order of tracing messages. This result is disturbing: Aspects such as tracing are usually perceived as a non-interference aspect in typical imperative aspect-oriented programs. However, for aspect-oriented functional language with lazy semantics, such aspects can turn out to be interfering despite the introduction of monadic computation.

A closer look at the monadified aspect code reveals the source of the problem: Calling the lifted `show` function, (`liftM show`), with the argument `arg2`, which in turns invokes the `show` function to obtain string representations of the arguments. This will lead to premature evaluation of the invocation of the multiplication, which is also being traced. Later, when the call to `facM` is resumed via `proceed` call, the multiplication call will be triggered and traced again. Hence the problem here is how to preserve the lazy

[5] The `tracerMulM` is very similar to `tracerFacM`, and thus omitted.

semantics of the base program while weaving aspects which are perceived to be non-interfering, such as tracing. Unfortunately, existing monadification schemes such as [9, 2, 3, 4] do not address these issues.

There are indeed two issues involved. First, although the use of any *strict function* in an aspect will result in evaluation of function arguments and thus change the order of evaluation, the monadification process should at least ensure that no duplication of evaluation occurs. Second, the `show` function aggrevates the situation by explicitly displaying this subtle change in the evaluation order to the trace user. As pointed out by Kishon, we should find an alternative display function that does not evaluate its argument and do a post lookup process to retrieve the value of its argument, a thunk or an evaluated value, to be compliant with lazy semantics.

We employ two techniques to address this issue of aspect interference and the need of `show` function, respectively. The first one is to maintain a *cache of function arguments* and wrap it around the original aspect monad to form a new aspect monad. The cache stores the values of function arguments which are either a thunk or an evaluated value, just like any typical implementation of lazy evaluation. This is to ensure that the arguments will not be evaluated more than once. The new aspect monad, its monad operation code and other auxiliary definitions are sketched in Figure 3.

```
-- Cells: thunks or values
data Cell = forall s a.Cell Bool (CState s a)
type Cache = Map.Map Int (Maybe Cell)
newtype CState s a = CState{
  realrunCState :: (s, Cache) ->
                   (Either a Int, (s, Cache))}
runCState :: CState s a -> (s, Cache) -> (a, (s, Cache))
runCState a (s, cs) =
         let (ea, (s', cs')) = realrunCState a (s, cs)
          in fromCacheEither ea (s', cs')
type M a = CState (UserVar, OutputBuf) a
instance Monad (CState s) where
  return t = CState $ \(s, cs) -> (Left t, (s, cs))
  ma >>= k = CState $
    \(s, cs) -> --(Left a) or (Right n)
    let (ea, (s', cs')) = realrunCState ma (s, cs)
        (ra, (s'', cs'')) = fromCacheEither ea (s', cs')
    in realrunCState (k ra) (s'', cs'')

instance MonadState s (CState s) where
  put s' = CState $ \(s, cs) -> (Left (), (s', cs))
  get    = CState $ \(s, cs) -> (Left s,  (s,  cs))

fromCacheEither :: forall s a. Either a Int ->
                       (s, Cache) -> (a, (s, Cache))
fromCacheEither (Left  a) (s, cs) = (a, (s, cs))
fromCacheEither (Right n) (s, cs) =
   ... evaluate the thunk of this cell via fromCell
                                and store its result
fromCell :: Cell -> (s, CacheSet) ->
                    (Either a Int, (s, CacheSet))
fromCell (Cell _ c) = realrunCState (unsafeCoerce# c)
--Functions for manipulating the cache
getNewCacheLoc :: CState s Int
getNewCacheLoc  = ...get a new cell loc from cache ...
setCache :: Int -> CState s a -> CState s a
setCache n t = ...put thunk t into loc n of the cache ...
add2Cache:: CState s a -> CState s (CState s a)
add2Cache arg = do n <- getNewCacheLoc
                   return $ setCache n arg
```

Figure 3. Cache-extended State Monad

The cache is a map from integers (locations) to cells containing thunks or values . The type `(CState s a)` is the key element of the new aspect monad. It can be viewed as an extended state monad wrapped by a cache of cells. When feeding an extended state, $(s, cacheSet)$, to run, the new aspect monad will produce an "either-object": either a real value, `(Left a)`, or a cell location, `(Right n)`, of the cache. In the definition of the bind operator ($\gg=$) of the new aspect monad, we first activate such an action using `realrunCState` to obtain an either-object, and then feed it to the function `fromCacheEither` which may look up the cache to trigger the monadic computation stored therein via the `fromCell` function. Note that, due to the use of `forall` quantifier in the definition of `Cell` type, we have to use the GHC extension of `unsafeCoerce` function in the `fromCell` function.

Also shown in Figure 3 are three functions for manipulating the cache. Function `getNewCacheLoc` extends the cache and returns the new location. Function `setCache` puts a monadic computation into the designated location of the cache. Finally, function `add2cache` employs them to put a monadified function argument computation into the cache.

With the introduction of (`CState s a`), the issue of duplicated evaluation of function arguments is resolved. Yet the lifted `show` function still makes the tracing messages out of order. We need to provide a special version of `show` to preserve the desired message order. The following function, `showM`, is the version we have designed for this purpose.

```
showM :: M Int -> M String
showM a =  case fst $ realrunCState a
                      (emptyM, emptyCacheSet) of
       Left  v -> return $ show v
       Right n -> return $ "<M'M:" ++ show n ++ "|"
```

Specifically, the new `showM` function does a "dry run" of the monad computation using an empty state, and if the result is a cell location, it returns a marker ("`<M'M:`") and a cell location to signal that its argument is kept in the cell. Afterwards, we provide a post processing function `deserialize` to traverse the output buffer and replace such marked locations with the value stored the specified cell of the cache.

Finally, the monadification function of Section 3.1 needs to be enhanced by applying the `add2Cache` function to those let-bound expressions which are fully applied function calls. Specifically, the (LET) part of $[\![\cdot]\!]_\Gamma$ needs to be refined as follows.

$$
\begin{array}{l}
[\![\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2]\!]_\Gamma = \\
\quad \mathit{let}\ e_1^M\ =\ [\![e_1]\!]_\Gamma \\
\qquad\ e_2^M\ =\ [\![e_2]\!]_\Gamma \\
\quad \mathit{in\ if}\ e_1\ \text{is of functional type or a constant} \\
\qquad \mathit{then}\ \mathsf{do}\ \{\mathsf{let}\ x = e_1^M;\ e_2^M\} \\
\qquad \mathit{else}\ \mathsf{do}\ \{x \leftarrow \mathit{add2cache}\ \$\ e_1^M;\ e_2^M\}
\end{array}
$$

Following this enhancement, the revised monadification of the tracing program is as follows. Running it with `facM (return 3)` `(return 1)` will produce the same result as described in [7].

```
tracerFacM :: (M Int -> M Int -> M Int) ->
              (M Int -> M Int -> M Int)
tracerFacM  proceed arg arg2 =
  do getIndentResult  <- getIndentM
     let ind = return getIndentResult
     ind' <- add2Cache $ (liftM2 (++)) (return "| ") ind
     setIndentM ind'
     s_arg2 <- add2Cache $ showM arg2
     str_1 <- add2Cache $ (liftM2 (++)) s_arg2 (return "]")
     str_2 <- add2Cache $ (liftM2 (++)) (return ",") str_1
     s_arg <- add2Cache $ showM arg
     str_3 <- add2Cache $ (liftM2 (++)) s_arg str_2
```

```
    str_4 <- add2Cache $
             (liftM2 (++)) (return "fac receives [") str_3
    str_5 <- add2Cache $ (liftM2 (++)) ind str_4
    putMsgM (return "tracerFac") str_5
    proceedResult  <- proceed arg arg2
    let result = return  proceedResult
    setIndentM ind
    show_res <- add2Cache $ showM result
    str_6 <- add2Cache $
             (liftM2 (++)) (return "fac returns ") show_res
    str_7 <- add2Cache $ (liftM2 (++)) ind str_6
    putMsgM (return "tracerFac") str_7
    result

facM :: M Int -> M Int -> M Int
facM n acc =
  do eq_n_zero <- add2Cache $ (liftM2 (==)) n (return 0)
     neq0 <- eq_n_zero
     if neq0 then acc
     else do nmacc <- add2Cache $
                      (tracerMulM (liftM2 (*)) n acc
             nm1 <- add2Cache $ (liftM2 (-)) n (return 1)
             (tracerFacM facM) nm1 nmacc
```

Lastly, note that in Section 3.2, we decided against treating let!-expression as syntactic sugar. With the introduction of cache into our system, it is worth noticing that we do not have to introduce `add2cache` operations to the monadified let!-definition.

4. Transforming Monadic Programs

Although AspectFun does not yet support monadic base programs, we can still describe how to extend our modification transformation when the base program is already monadic.

4.1 Using Monad Transformers

In the presence of monadic base programs, we need to employ the state monad transformer mechanism to combine the monad of the base program with the aspect monad. For example, the display update program in Example 3 uses the `IO` monad, hence the aspect monad for it is defined as follows.

```
type S m a = StateT (UserVar, OutputBuf) m a
type M a = S IO a
```

In general, the monadification operator $\mathcal{M}$ should be extended as follows:

$$\mathcal{M}(t_1 \to t_2) \quad \Rightarrow \quad \mathcal{M}(t_1) \to \mathcal{M}(t_2) \tag{3}$$

$$\mathcal{M}(a) \quad \Rightarrow \quad MT\ N\ a \tag{4}$$

$$\mathcal{M}(N\ (t_1 \to t_2)) \quad \Rightarrow \quad MT\ N\ (\mathcal{M}(t_1) \to \mathcal{M}(t_2)) \tag{5}$$

$$\mathcal{M}(N\ a) \quad \Rightarrow \quad MT\ N\ a \tag{6}$$

where N is the monad used in the base program (base monad), and MT is the monad transformer being used. In Example 3, N is `IO` and MT is `StateT (UserVar, OutputBuf)`.

Finally, the monadification function, $[\![\cdot]\!]_\Gamma$, also needs to be adjusted. There are two categories of changes. Firstly, we must apply proper lifting operations when passing computed values between the base monad and the aspect monad. Essentially, we shall use `liftM` operator to lift operations on the base monad before applying them, and use `liftN` operator to lift results of computations in the base monad, `N`. The following enhanced versions of (PRIM) and (APP) illustrate the ideas.

(PRIM) $[\![p]\!]_\Gamma = \texttt{liftMn}\ p$
where n is the arity of the primitive function or the base monad operation p

(APP) $[\![e_1\ e_2]\!]_\Gamma =$
if $isFullAppBaseMonadOP(e_1)$
then $\mathsf{do}\ \{x \leftarrow [\![e_1]\!]_\Gamma\ [\![e_2]\!]_\Gamma;\ \texttt{liftN}\ x\}$
else $[\![e_1]\!]_\Gamma\ [\![e_2]\!]_\Gamma$

Secondly, we need to extend the $[\![\cdot]\!]_\Gamma$ function to handle the bind ($\gg=$) and the return operations of the base monad.

(BIND) $[\![\mathsf{do}\ \{x \leftarrow e_1;\ e_2\}]\!]_\Gamma =$
$\mathsf{do}\ \{x' \leftarrow [\![e_1]\!]_\Gamma;\ \mathsf{let}\ x = \mathsf{return}\ x';\ [\![e_2]\!]_\Gamma\}$

(RETURN) $[\![\mathsf{return}\ e]\!]_\Gamma = [\![e]\!]_\Gamma$

In the case of (BIND), we need to use the return of the new monad to move the result of do-binding action back to the new monad. As to the case of (RETURN), we simply drop the return of the base monad and return the monadified expression.

The following code snippets show the original version of the `getLineP1` function and its monadified version.

```
getLineP1 :: Line -> IO Point
getLineP1 (L l) =
    do (p1,_) <- readIORef l
       return p1

getLineP1M :: M Line -> M Point
getLineP1M ll =
    do (L lBindout) <- ll              --PatternMatching
       let l = return lBindout         --Bind
       bmOP <- (liftM readIORef) l     --Prim
       (_, p1BindOut) <- liftIO bmOP   --App
       let p1 = return p1Bindout       --Bind
       p1                              --Return
```

4.2 Unified Monadification Scheme

We started from a simple state monad of user variables and output buffer, and then extended it with a cache facility. Now we generalized the state monad along another direction using monad transformer. It would be nice to combine these different enhancements under a unified monadification framework. Specifically, we devise a cache-extended state monad transformer that can accommodate the aspect monads presented so far as special cases. This monad transformer, `CStateT`, is defined in terms of another monad transformer, `CacheT` as follows.

```
newtype CacheT m a = CacheT{
        realrunCacheT :: Cache ->
                         m (Either a Int, Cache)}
type CStateT s m a = CacheT (StateT s m) a

instance MonadTrans CacheT where
    lift ma = CacheT $
                \cs -> ma >>= \a -> return (Left a, cs)

instance Monad m => Monad (CacheT m) where
    return t = CacheT $ \cs -> return (Left t, cs)
    ca >>= k = CacheT $ \cs ->
        do (ea, cs')  <- realrunCacheT ca cs --Either a
           (ra, cs'') <- fromCacheEither ea cs'
           realrunCacheT (k ra) cs''

instance MonadIO m => MonadIO (CacheT m) where
    liftIO = lift . liftIO
...
```

Given the above definitions, we can easily derive the respective aspect monads defined in previous subsections.

1. The aspect monad of Section 3.2 can be replaced with the following one:

```
type M a = CStateT (UserVar, OutputBuf) Identity a
                                        -- identity monad
```

2. The aspect monad of Section 3.3 can be replaced with the following one:

```
type M a = CStateT (UserVar, OutputBuf) Identity a
```

3. The aspect monad of Section 4.1 can be replaced with the following one:

```
type M a = CStateT (UserVar, OutputBuf) IO a
```

Figure 4 shows a summary of the monads we developed along the way towards our goal.

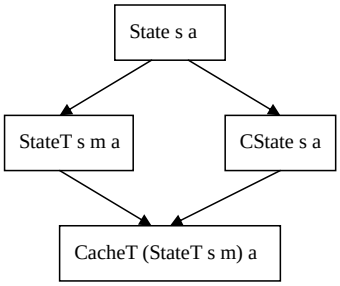

Figure 4. Summary of the Monad Transformations

5. Related Work

Research works about monadification can be traced back to those on continuation passing style conversion [3, 4], where compiler-based transformation rules were developed to convert all functions and intermediate results in a program into monadic form. In particular, the A-normalization technique was introduced in [3]. Although transformations rules for both call-by-value and call-by-name were presented, no concerns about lazy semantics (call-by-need) were discussed.

Our monadification scheme is inspired by the monad introduction transformation of Lämmel [9], in which a set of type-directed transformation rules were devised to convert A-normalized expressions into monadic computation. The rules are given in natural semantics style and exhibit a degree of non-determinism to support the case of monadifying only selected functions. Recently, Erwig and Ren [2] have developed a set of syntax-directed rewriting rules that can convert a group of selected functions into a monadic form and identified the correctness criteria for the conversion. Once again, neither of them addresses the issues related to lazy semantics either.

In this work, the monadification transformation is performed after type inference and static weaving of the base program and its side-effecting aspects. Hence we have full type information of the expression available for monadification. Moreover, our monadification scheme differs from theirs by lifting function parameters to the monadic space, too. While this decision enables us to derive a simple monadification function for transforming the woven code in a lazy context, it prohibits us from being able to monadify only selected functions, as was done by the above two approaches. In particular, any library functions for AspectFun must also be monadified if they cannot be simply lifted to work with side-effecting aspects. However, none of the approaches, including ours, can handle the case that the source code of external functions invoked in the monadified function is unavailable.

Kishon's thesis work [7, 8] developed a semantics-directed program monitoring framework. The main tool his framework employed for collecting program execution information is code instrumentation. His annotation labels for marking program points to monitor are just like pointcuts in aspect-oriented programming. But the instrumentation is done at the interpreter (semantics) level, not source-level. Hence it is easier for his framework to utilize semantic entities such as environment and store to implement a thunk-based cache for performing lazy tracing.

The potential relation between aspects and monads was first suggested by De Meuter [11]. The recent work of Hofer and Ostermann [5] explored this subject in further depth and presented a detail comparison between aspects and monads in terms of two dimensions: their capabilities and effects on modularity. Our example of "display update" is based on the code of their work.

6. Conclusions and Future Work

We have proposed a simple state manipulation construct for developing aspects that can perform side-effecting operations in aspect-oriented lazy functional languages. Such aspects are good for monitoring the execution state of the base program in a modular manner. We have also presented a systematic monadification scheme to realize them by translating the woven code to a purely monadic style functional code. Along the way, we have identified the difficulties involved in monadifying such side-effecting aspects in a lazy functional setting and proposed a solution which employs a cached state monad transformers to reconcile the gap between side effects and lazy semantics.

The AspectFun compiler has been extended accordingly to support the proposed constructs for developing side-effecting aspects. However, it is clear that the generated monadic code can be improved in certain ways. Indeed, a closer examination of the monadified code generated by our compiler for the tracing example reveals that most of the calls to `add2Cache` function can be optimized away. Specifically, all such calls inside `tracerFacM` can be eliminated since the variables receiving the call results, such as `s_arg` and `str_1`, are used only once therein. However, since `arg` and `arg2` are used more than once in the tracer aspect, the two calls to `add2cache` for binding `nmacc` and `nm1` inside `facM` cannot be eliminated. Hence we plan to investigate how to optimize the monadic code via some static analysis techniques. In particular, we speculate that the type-based usage analysis developed in [14] can be adapted to serve our purpose.

Another direction we intend to explore is how to support selective monadification. Currently, our scheme requires full monadification, namely all functions referenced in the program be either monadified or be lifted to work in the monadic space. In practice, this may not be acceptable, as many library functions, such as `map`, are higher-order functions that cannot be simply lifted to work with monadified functions. We shall look into the feasibility of devising a type-directed approach to address the need of partial monadification.

7. Acknowledgment

This work is partly motivated by a conversation between the first author and Hidehiko Masuhara during AOAisa 2007.

References

[1] Kung Chen, Shu-Chun Weng, Meng Wang, Siau-Cheng Khoo, and Chung-Hsin Chen. A compilation model for aspect-oriented polymorphically typed functional languages. In *Static Analysis, 14th International Symposium, SAS 2007*, volume 4634 of *LNCS*, pages 34–51. Springer-Verlag, 2007.

[2] Martin Erwig and Delin Ren. Monadification of functional programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

[3] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, 1993.

[4] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–247, 1993.

[5] Christian Hofer and Klaus Ostermann. On the relation of aspects and monads. In *Foundations of Aspect-Oriented Languages Workshop at AOSD*, pages 37–46. ACM Press, 2007.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, 2001.

[7] Amir Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring*. PhD thesis, Yale University, June 1992.

[8] Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, 1995.

[9] Ralf Lämmel. Reuse by program transformation. In *Functional Programming Trends 1999. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop*. Intellect, 2000.

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

[11] W. De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

[12] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[13] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. In *PEPM '06: Workshop on Partial Evaluation and Program Manipulation*, pages 78–87. ACM Press, 2006.

[14] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–28, January 1999.