**Higher-Order and Symbolic Computation manuscript No.**
(will be inserted by the editor)

# Side-Effect Localization for Lazy, Purely Functional Languages via Aspects ⋆

**Kung Chen · Shu-Chun Weng · Jia-Yin Lin · Meng Wang · Siau-Cheng Khoo**

Received: date / Accepted: date

**Abstract** Many side-effecting programming activities, such as profiling and tracing, can be formulated as crosscutting concerns and be framed as side-effecting aspects in the aspect-oriented programming paradigm. The benefit gained from this separation of concerns is particularly evident in purely functional programming, as 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 lazy purely 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 monadic style purely functional code. Furthermore, we present a static and dynamic semantics of the aspect programs and reason about the correctness of our monadification scheme with respect to them.

**Keywords** Aspect-oriented programming · Side-effect · Lazy semantics · Monadification

⋆ This is an extension of the work presented at the ACM SIGPLAN 2009 Workshop on Partial Evaluation and Program Manipulation (PEPM'09).

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

Shu-Chun Weng
Yale University, USA
E-mail: scweng@cs.yale.edu

Meng Wang
Oxford University, UK
E-mail: menw@comlab.ox.ac.uk

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

## 1 Introduction

Aspect-oriented programming (AOP) aims at capturing crosscutting concerns. Many crosscutting concerns are orthogonal to the mainline computation that they interacts with. A few examples of such concerns are profiling, tracing, and memoization. By their own nature, aspects that implement such orthogonal concerns do not alter the evaluation result of the base program; their computations are manifested in side-effects. We refer to such aspects as *side-effecting aspects* (also know as *non-interfering aspects [2]* from a data-flow perspective). Side-effecting aspects are particularly useful because orthogonal concerns are more likely to be subject to deploying, updating, and removing from a software system, a situation where AOP solutions are doubly attractive.

Most aspect-oriented programming languages are based on object-oriented languages, where uncontrolled side-effects are the norm. The recent surge of interests in introducing aspect-oriented concepts in functional languages [3,16], in particular in purely functional languages [1,21,18], poses fresh challenges. Although we can hide the hairy details of state manipulation by using monads [19], it does require advance planning, which is fundamentally at odds with the concept of *obliviousness* in AOP.

Adding monadic effects to a pure program entails a comprehensive rewriting; it is therefore convenient to support side-effecting directly and automate the process via source-to-source transformation. Such a technique has been pioneered by Lämmel [14] and is referred to as *monadification* by Erwig and Ren [5]. In many situations, this convenience comes at a cost: primitive support for side-effects compromises referential transparency property and all the nice reasoning properties that derive from it. Our proposal, on the other hand, eliminates the need for compromise between property preservation and convenience. This is achieved through a carefully designed weaving scheme, which preserves the non-interfering nature of side-effecting aspects.

In our previous work on 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 maintain 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.

The general vision is clear: a state monad is employed as the repository for mutable variables and and all functions are lifted into monadic ones. But care must be taken in implementing of such a scheme. First of all, monadification always imposes an evaluation order, which may or may not be what is desired. Even when a preferred evaluation order is known up front, it is not a simple task to instruct the monadification process to faithfully follow in the context of lazy semantics of Haskell. Let's illustrate this point by considering a small example involving debugging Haskell programs through tracing taken from [4].

```
f x = 3 `div` x
h = ... -- arbitrarily deep computation
g = h (f 0)
```

The function *div* is partial because it may throw the divide-by-zero exception. When this happens, the Glasgow Haskell compiler (GHC) only outputs the very unhelpful message "*** Exception: divide by zero". Since there can be arbitrarily many calls to *div*, which can be arbitrarily deep in nesting, more informative tracing messages may be appreciated. However, if we follow the lazy evaluation of Haskell, the trace includes a call to $g$, followed by a call to $f$, and an arbitrarily complicated execution of $h$ before the offending call to *div* shows up. Any useful information may be overwhelmed by the "noise" of $h$'s evaluation. What one really wants here is a short trace to the exception, which skips $h$'s body following an eager evaluation order. Yet, this may not be preferred when the presence of laziness is necessary, for example when dealing with infinite values.

In this paper, we make the following contributions:

1. We extend the aspect-oriented functional language, AspectFun, with side-effecting constructs that support direct state manipulation in aspects.
2. We present a general type-directed monadification scheme that transforms woven code into monadic style purely functional code. The semantics and correctness of the scheme are discussed in detail.
3. We devise a cached state monad in Haskell to support the lazy evaluation of monadified expressions in side-effecting aspects.
4. We demonstrate with examples the effectiveness of our system in dealing with tracing, profiling, and optimization of lazy functional programs.
5. We outline a uniform monadification scheme that can also handle monadic base programs.

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 presents a general framework of monadification with respect to an abstract monad, followed by the semantics and correctness of this framework. Section 4 specializes the abstract monad to Haskell state monads for implementing side-effecting aspects in AspectFun, and describes the issues of and solutions to preserving laziness in our monadification scheme. Section 5 illustrates how we can use monad transformers to handle monadic base programs and outlines a unified monadification scheme that accommodates both cases. Section 6 describes related work. Finally, Section 7 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 a few Haskell constructs that are not yet implemented in AspectFun.

### 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*, $\mathsf{around}\ \{\overline{pc}\}$, designating when the advice will be executed. In aspect-oriented programming [11], the specific program execution points that triggers advice are 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 advice during a function application is called *weaving*. When an advice of the form "$n@\mathsf{advice\ around}\ \{\overline{pc}\}\ (arg) = e$" is triggered by a call to a function, say $f$, the argument variable $arg$ is bound to the actual argument of the $f$-call.

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

**Fig. 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 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 inspect 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` 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* [20] 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 in pointcut and statically weave aspects into base program. In previous work, we have built a compiler that employs a type-directed static weaver to translate an AspectFun program into executable Haskell code [1]. Moreover, our monadification procedure for handling side-effecting aspects is largely independent of the static weaving step, as it is performed after the weaving step during compilation. Therefore, we shall not discuss the processing of pointcuts and advice in this paper.

### 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 within the scope of an aspect. We use `var` as the keyword to begin such a declaration. The syntax of an aspect declaration is also slightly extended to include both declarations of advice and of variables. The precise syntax is as follows.

$$\begin{array}{ll} \text{Declarations } d & ::= \ldots \mid \mathsf{var}\ id :: t\ [= e] \mid n@\mathsf{advice\ around}\ \{\overline{pc}\}\ (arg) = e \\ \text{AspectDecl} & ad ::= \mathsf{aspect}\ name\ \mathsf{where}\ \overline{d} \end{array}$$

Such mutable variables are declared with a monomorphic and ground type, $t$, and an optional initializing expression, $e$. Equipped with them, advices in the same aspect are able to keep pertinent state information forming side-effecting aspects. 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
```

Associated with each mutable variable declared, there is a pair of implicitly declared getter and setter functions for interacting with the state. Their side-effects are sequenced by sequencing expressions, $(e_1; e_2)$. In particular, the variable declaration above results in the following declarations of a setter and a getter function, respectively.

```
getProfileMap :: Map.Map String Int
setProfileMap :: Map.Map String Int -> ()
```

[1] The `Map` is an alias of the `Data.Map` in Haskell's standard hierarchical libraries.

Let's look into an example of Fibonacci function `fib` benefiting from a momoization aspect to remove repeated computation and a profiling aspect.

*Example 1*

```
fib n = if n <= 1 then 1
        else fib (n - 1) + fib (n - 2) in
--aspect 1
aspect profiler where
  var profileMap :: Map.Map String Int
  advice around {fib} (arg) =
         let incProfile fname =
                   set! pMap = getProfileMap;
                   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
         in incProfile "fib"; proceed arg in
--aspect 2
aspect memoFib where
  var memoMap    :: Map.Map Int Int
  advice around {fib} (arg) =
    case lookupCache arg of
     Just v -> v
     Nothing -> set! v = proceed arg;
                insertCache arg v; v  in
fib 10
```

Caution has to be taken for operations involving state access since the order of evaluation matters. We use the keyword `set!` for sequenced bindings. They effectively force the evaluation of a binding prior to the evaluation of its body, simulating a kind of eager semantics. In `profiler`, the auxiliary function `incProfile` makes sure the state is fully evaluated before attempting to update it, removing the risk of a race condition. In `memoFib`, the inputs of the state operation `insertCache`, are evaluated before the state update. Though it is probably not the only way to correctly implement the momoization aspect, we enforce the coding convention for the sake of program comprehension.

Besides mutable variables, IO 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 internal buffer. Together with the getter and setter functions, they form the *state API* of an aspect.

The second example is a tracing aspect for the tail recursive factorial function, adapted from Kishon's thesis work on program monitoring [12].

*Example 2*

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

```
aspect tracer where
  var indent :: String = ""
  advice around{fac, (*)} (arg) = \arg2 ->
    set! ind = getIndent ;
    setIndent ("| " ++ ind);
    set! v1 = arg;
    set! v2 = arg2;
    putMsg "tracer" (ind++tjp++" receives ["++
           show v1 ++ ", " ++ show v2 ++ "]");
    set! result = proceed v1 v2 ;
    setIndent ind;
    putMsg "tracer" (ind++tjp++" returns " ++
                       show result);
    result
```

Here the state to be maintained is the indentation string, stored in 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 currently being advised, namely the current join point.[2] The advice simply traces the arguments passed to and the results returned from the advised functions via the `show` function. Note that we have used sequenced bindings to enforce a call-by-value trace, which is printed below.

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

In Section 4, we look into how a lazy trace can be obtained, which turns out to be non-trivial because the execution of any added IO operations easily interact with the trace of the base program, causing changes in evaluation order.

## 3 Monadifying Aspect Programs

The first step of AspectFun compilation is to weave aspects into the base program, thus producing an integrated program of expressions, which we call *woven code*. In the presence of side-effecting aspects, it is necessary for the woven code to be transformed

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

to a monadic style, in order to retain its functional purity. This and the following section illustrate our monadification transformation for expressions, pure or side-effecting, in a woven code.

First, we present a general framework for monadifying an expression using an abstract monad, $(M, \mathsf{return}, \gg\!=)$, in a non-strict evaluation context, and show that our monadification scheme possesses good properties with respect to the static and dynamic semantics of expressions in woven code. Next, in the following section, we specialize $M$ to a specific state monad in Haskell so that we can also define the monadified version of those state-aware functions used by side-effecting aspects.

## 3.1 Monadifying Expressions

Like the pioneering work of Lämmel [14], our monadification transformation consists of two major steps, namely A-normalization [7] and monad introduction.

### *3.1.1 A-Normalization*

Given an expression, A-normalization converts it into a form in which every intermediate computation is assigned a name by a let-expression. Such normalized expressions, called A-normal form, is a popular intermediate representation used in compilers [7] and semantic specifications [15] for functional languages. Essentially, in A-normal form, all applications are applications of an expression to a variable. The arguments of an application and the condition part of an if-expression are all captured by the binding parts of let-expressions wrapped around them.[3]

Let us 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
```

[3] Note that we conduct alpha renaming along with A-normalization to avoid any name conflicts.

```
                    let fibm2 = profiler fib nm2 in
                      let nm1 = n - 1 in
                        let fibm1 = profiler fib nm1 in
                          (+) fibm1 fibm2    in
profiler fib 10  --main
```

We note that A-normalization changes only the structure of a program, not the order of argument evaluation, as let-expressions are evaluated lazily.

As a result of A-normalization, the syntax of the expressions to be monadified can be summarized as the following three syntactic categories for ease of subsequent discussion.

$$
\begin{array}{lll}
\text{Atoms} & a & ::= c \mid x \\
\text{Pure Expressions} & e & ::= a \mid p \mid \lambda x.e \mid e\ a \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
 & & \quad\ \mathsf{if}\ a\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{Effectual Expressions} & e^! & ::= \cdots \mid e^!;\ e^! \mid \mathsf{set!}\ x = e^!;\ e^!
\end{array}
$$

Atoms include constants and variables. Besides atoms and primitives ($p$), pure expressions are A-normalized standard expressions. The changes made by A-normalization are manifested in applications and if-expressions. Effectual expressions extend pure expressions by including side-effecting constructs; they form the whole set of expressions to be monadified. Note that those state-aware functions, such as getters and `putMsg`, are considered primitives, but their monadification will not be specified until next section, when the state monad is defined.

### *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 $\mathcal{M}$ that converts an expression type to a monadic type as follows.

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

$$\mathcal{M}(t) \Rightarrow M\ t \tag{2}$$

$$\mathcal{M}(\forall \bar{a}.t) \Rightarrow \forall \bar{a}.\mathcal{M}(t) \tag{3}$$

where rule (1) applies to functional types and rule (2) applies to non-functional (atomic) types. For type schemes, we simply apply $\mathcal{M}$ to their type body. As type predicates are required only for static weaving purpose, we can safely ignore them in the monadification step.

We note that the monadification schemes proposed by Lämmel [14] and Erwig and Ren [5] 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 monad and thus support the non-strict evaluation semantics of AspectFun.

The concrete steps for lifting computations to monadic space are designed by following the above monadic type conversions. We formalize them as a set of type-directed rewriting rules, $[\![\cdot]\!]^t_\Gamma$, that converts an expression in A-normalized form, $e^!$, to a monadified version, $e$, over the designated monad, $M$. The subscript $\Gamma$ is a type environment containing the types for the free identifiers occurring in $e^!$ and the superscript $t$ is

the type of the expression to be monadified. Recall that we conduct the monadification transformation after type-directed weaving. Thus the type of every expression is available in this step. Figure 2 displays the complete set of type-directed rewriting rules, implicitly parameterized over a monad $(M, \mathsf{return}, \gg)$, along with some auxiliary functions.

Most of the rewriting rules are purely syntactic and quite simple; the only notable exception is the (Var) rule for variables, which will be explained in detail later. We summarize the other rules 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. There are two rules for rewriting if-expressions, depending on their condition part. We may need to apply a monad-binding to trigger the evaluation of their monadified condition expression. The rewriting rules for side-effecting constructs, (Seq) and (Set), are standard in using monads to handle states. The remaining cases are simply syntactic composition of the monadified components.

$$
\begin{array}{lll}
 & [\![\cdot]\!]^t_\Gamma & : e^! \longrightarrow e \\
\text{(Const)} & [\![c]\!]^t_\Gamma & = \mathsf{return}\ c \\
\text{(Prim)} & [\![p]\!]^t_\Gamma & = \texttt{liftMn}\ p \quad \textit{where } \texttt{n} \text{ is the arity of primitive function } p \\
\text{(If)} & [\![\mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]^t_\Gamma & = [\![a]\!]^{Bool}_\Gamma \gg\!= \lambda a'.\mathsf{if}\ a'\ \mathsf{then}\ [\![e_1]\!]^t_\Gamma\ \mathsf{else}\ [\![e_2]\!]^t_\Gamma \quad a' \text{ is fresh} \\
\text{(Lam)} & [\![\lambda x.e]\!]^{t_1 \to t_2}_\Gamma & = \lambda x.[\![e]\!]^{t_2}_\Gamma \\
\text{(App)} & [\![e\ a]\!]^t_\Gamma & = [\![e]\!]^{t_a \to t}_\Gamma\ [\![a]\!]^{t_a}_\Gamma \\
\text{(Let)} & [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^t_\Gamma & = \mathsf{let}\ x = [\![e_1]\!]^{t_x}_\Gamma\ \mathsf{in}\ [\![e_2]\!]^t_\Gamma \\
 & & \\
\text{(Seq)} & [\![e_1;\ e_2]\!]^t_\Gamma & = [\![e_1]\!]^{t_1}_\Gamma \gg\!= \lambda\_.[\![e_2]\!]^t_\Gamma \\
\text{(Set)} & [\![\mathsf{set!}\ x = e_1;\ e_2]\!]^t_\Gamma & = [\![e_1]\!]^{t_1}_\Gamma \gg\!= \lambda x'.\mathsf{let}\ x = \mathsf{return}\ x'\ \mathsf{in}\ [\![e_2]\!]^t_\Gamma \\
\text{(Var)} & [\![x]\!]^t_\Gamma & = \mathtt{pos}^S_{t'}(x) \\
 & & \quad \text{where } \forall \bar{a}.t' = \Gamma(x) \text{ and } S \text{ is a substitution such that } t = St'
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{pos}^S_{t_1 \to t_2}(e) = \lambda x.\ \mathtt{pos}^S_{t_2}(e\ \mathtt{neg}^S_{t_1}(x)) & & x \notin fv(e) \\
\mathtt{pos}^S_a(e) = \mathtt{flatten[}Sa\mathtt{]}\ (e) & & \text{if } a \in \mathrm{dom}(S) \\
\mathtt{pos}^S_t(e) = e & & \text{otherwise} \\
 & & \\
\mathtt{neg}^S_{t_1 \to t_2}(e) = \lambda x.\ \mathtt{neg}^S_{t_2}(e\ \mathtt{pos}^S_{t_1}(x)) & & x \notin fv(e) \\
\mathtt{neg}^S_a(e) = \mathsf{return}\ (e) & & \text{if } a \in \mathrm{dom}(S) \text{ and } Sa \text{ is not an atomic type} \\
\mathtt{neg}^S_t(e) = e & & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{flatten[}t\mathtt{]}(e) = e & & \text{if } t \text{ is atomic} \\
\mathtt{flatten[}t_1 \to t_2 \cdots t_n \to t'\mathtt{]}(e) = (\lambda x_1 \cdots x_n.\ e \gg\!= \lambda e'.\ e'\ x_1 \cdots x_n) & & \text{otherwise}
\end{array}
$$

**Fig. 2** Type-Directed Monadification Rules and Auxiliary Functions

The rule for variables, (Var), is the most complicated one. Its complexity arises due to the need to support polymorphic higher-order functions. Indeed, if it is not the case, the rewriting rule for a variable simply leaves it intact: $[\![x]\!]^t_\Gamma = x$. Before proceeding to explain the details of how the (Var) rule works, let us see an example of the monadification transformation that does not involve any higher-order functions.

The following code shows the monadified version of the A-normalized `fib` function presented earlier.

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

In the following sections, we will often adopt the practice in Haskell community that uses `do`-notation with a fold over the `do`-bindings to express monadic computation. In particular, the following Haskell code is the sugared version for the above monadified `fib` function.

```
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
```

Now, let us resume the discussion of the (Var) rule. As can be seen from its right-hand side, the rewriting of this rule is driven by the type of the underlying variable in the type environment and the type assigned to it in a context. The result of rewriting may be the same variable or an expanded expression with some boilerplate code inserted by the `pos` and the `neg` functions. Specifically, the `neg` function may insert calls to the `return` of the underlying monad to add an additional level of monadic structure; and, the `pos` function may insert invocations of the `flatten` combinators to remove one level of monadic structure. The `flatten` combinators are synthesized according to the type context. They act like the conventional monad join operator ($join :: M\ (M\ a) \to M\ a$), but work on higher-order functions with types such as $M\ (M\ a \to M\ b)$. Essentially, the purpose of inserting such boilerplate code is to make the monadified expressions that involve higher-order functions type check correctly. This is better illustrated by an example.

Consider the expression, $(id_1\ id_2)$, which applies the identity function, $id$, to itself. (The subscripts are employed for ease of references.) The monadic type scheme for $id$ is $\forall a.M\ a \to M\ a$. Now, suppose that we specialize the type of $id_2$ to $Int \to Int$ with type substitution $S = [a \mapsto (Int \to Int)]$. Then, by (App),

$$[\![(id_1\ id_2)]\!]_{\Gamma}^{Int \to Int} = [\![id_1]\!]_{\Gamma}^{(Int \to Int) \to (Int \to Int)}\ [\![id_2]\!]_{\Gamma}^{Int \to Int}$$

Now, as the monadic type of $id_2$ is $M\ Int \to M\ Int$, in order for $[\![(id_1\ id_2)]\!]_{\Gamma}^{Int \to Int}$ to be type correct, the monadic type of $id_1$ should be $(M\ Int \to M\ Int) \to (M\ Int \to$

$M\ Int$). Therefore, the result of monadifying $id_1$ cannot simply be $id_1$; otherwise, according to the type scheme of $id$, the monadic type assigned to $id_1$ would be $M\ (M\ Int \to M\ Int) \to M\ (M\ Int \to M\ Int)$, which will lead to a type error. On the other hand, applying (VAR) to monadify $id_1$ would reconcile the type mismatch and produce a type-correct result:

$$\begin{aligned}
&[\![\mathrm{id}_1]\!]_{\Gamma}^{(Int\to Int)\to(Int\to Int)} \\
&= \mathtt{pos}^{S}_{a\to a}(\mathrm{id}_1) \\
&= \lambda x.\ \mathtt{pos}^{S}_{a}(\mathrm{id}_1\ \mathtt{neg}^{S}_{a}(x)) \\
&= \lambda x.\ \mathtt{flatten[}Sa\mathtt{]}\ (\mathrm{id}_1\ (\mathsf{return}\ x)) \\
&= \lambda x.\ \mathrm{flatten}\ (\mathrm{id}_1\ (\mathsf{return}\ x)) \\
&\quad \text{where } S = [a \mapsto (Int \to Int)] \\
&\qquad\quad \mathrm{flatten} = \lambda v.\lambda x.\ v \gg\!= \lambda v'.\ v'\ x
\end{aligned}$$

We shall give a formal account of the correctness of such enhancements in the following section.

It is worth further discussing the need of inserting calls to `return` and `flatten` combinators when monadifying higher-order functions. Essentially, the reason for doing so can be traced back to the definition of our monadification operator, $\mathcal{M}(\cdot)$. Recall its first equation:

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

This equation embodies the key features as well as the limitations of our monadification scheme. In particular, as pointed out in [8], in this scheme, "the effect of monadification on a function is to produce a function, rather than a computation of a function." Consequently, monadification of higher-order functions requires the insertion of some boilerplate code to make the resulting expression type check.

An alternative equation for monadic types we had considered is the following one:

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

Although this alternative equation simplifies the monadification of higher-order functions, it leads to more complicated monadic types and monadified expressions with much more boilerplate code that simply acts to add or remove additional monadic structure. Hence we decide to retain the original equation.

### 3.2 Semantics and Correctness

This section gives a formal account of the static and dynamic semantics of expressions and presents the properties of our monadification scheme with respect to the semantics. There are two major theorems. First, the type of a monadified expression is the same as the monadic type assigned to the original expression. Second, the semantic value of a pure expression is preserved by the monadification transformation. The technical lemmas and their proofs can be found in the appendix.

*3.2.1 Static Semantics and Type Preservation*

Let us begin with the static semantics of expressions, as specified by the typing rules in Figure 3. Pure expressions, effectual expressions, and monadic expressions are all included so that we can reason about their types in the same framework. Hence there are three groups of rules. The first group follows the typical Hindley-Milner style to type check pure expressions. In particular, the application, $inst(\sigma)$ instantiates the given type scheme, $\sigma$ to a type $t$; and the application, $gen(\Gamma, t)$, generalizes the type $t$ to a type scheme $\sigma$ with respect to the type environment $\Gamma$.

The second group of rules declares and specifies the types of monadic primitives in a standard manner. Finally, the third group prescribes how the components of an effectual expression should be typed to get a type correct expression. The restriction of the components to atomic types is a consequence of our monadification scheme and the typing rules for monadic primitives.

1. Common expressions:

$$\frac{x : \sigma \in \Gamma \qquad t = \text{inst}(\sigma)}{\Gamma \vdash x : t} \qquad \frac{\Gamma.x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \frac{\Gamma \vdash e_1 : t_2 \to t_1 \qquad \Gamma \vdash a : t_2}{\Gamma \vdash e_1\ a : t_1}$$

$$\frac{\Gamma \vdash a : \mathit{Bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \sigma = \text{gen}(\Gamma, t_1) \qquad \Gamma.x : \sigma \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2}$$

2. Monadic expressions:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{return}\ e : M\ t} \qquad \frac{\Gamma \vdash e_1 : M\ t_1 \qquad \Gamma \vdash e_2 : t_1 \to M\ t_2}{\Gamma \vdash e_1 \gg\!= e_2 : M\ t_2}$$

3. Effectual expressions: ($t_1$ and $t_2$ below must be atomic types)

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1;\ e_2 : t_2} \qquad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma.x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathsf{set!}\ x = e_1;\ e_2 : t_2}$$

**Fig. 3** Typing Rules for Expressions

As our monadification scheme lifts the computations in an expression to the designated monad, we have to ensure that the monadified expression has the proper monadic type. Formally speaking, we require that the following statement hold for every expression.

$$\text{if}\ \ \Gamma \vdash e : t \quad \text{then}\ \ \mathcal{M}(\Gamma) \vdash [\![e]\!]^t_\Gamma : \mathcal{M}(t)$$

We shall refer to the above statement as type preservation of our monadification scheme. Before proceeding to establish it, we need the following definitions to specify how the monadification operator works on type environments and type substitutions:

– For a type environment $\Gamma$, $\mathcal{M}(\Gamma)$ is the pointwise application of $\mathcal{M}(\cdot)$ to the type part of all bindings in $\Gamma$, i.e., $\mathcal{M}(\Gamma)(x) = \mathcal{M}(\Gamma(x))$.

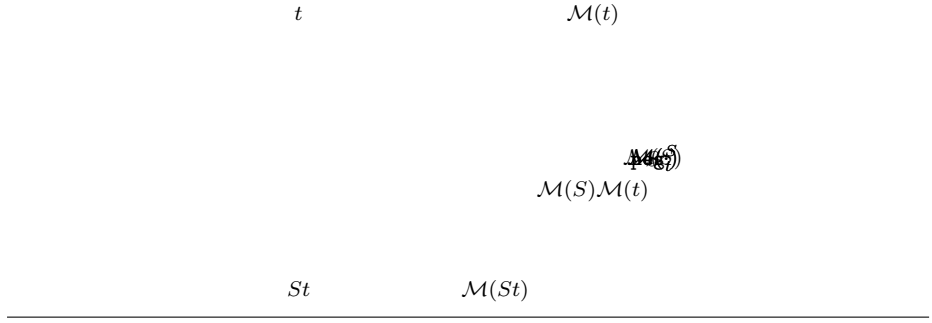


**Fig. 4** The Non-distributivity Between Substitution and Monadification Operator

– For a type substitution $S$ from type variables to types, $\mathcal{M}(S)$ is also a type substitution with $\mathrm{dom}(\mathcal{M}(S)) = \mathrm{dom}(S)$ and

$$\mathcal{M}(S)(a) = \begin{cases} Sa & \text{if } Sa \text{ is an atomic type} \\ \mathcal{M}(Sa) & \text{otherwise} \end{cases}$$

The definition of $\mathcal{M}(\Gamma)$ is straightforward, but the definition of $\mathcal{M}(S)$ needs some extra attention. Specifically, as a type substitution may turn a type variable into a functional type, the monadification operator has been pushed to monadify the resulting functional type in such cases. However, although both $\mathcal{M}(\cdot)$ and $S$ are distributive over the functional type operator ($\rightarrow$), they do not distribute with each other when applying to a functional type. In other words, $\mathcal{M}(t_1 \rightarrow t_2) = \mathcal{M}(t_1) \rightarrow \mathcal{M}(t_2)$, $S(t_1 \rightarrow t_2) = (St_1 \rightarrow St_2)$, but, in general, $\mathcal{M}(St) \neq \mathcal{M}(S)\mathcal{M}(t)$.

This can be illustrated by the $(id_1\ id_2)$ example presented above. As the type scheme for $id$ is $\forall a.a \rightarrow a$, a valid type instance for $id$ is $t = b \rightarrow b$. Now, given type substitution $S = [b \mapsto (Int \rightarrow Int)]$, we get $\mathcal{M}(St) = (M\ Int \rightarrow M\ Int) \rightarrow (M\ Int \rightarrow M\ Int)$. But, on the other hand, $\mathcal{M}(S)\mathcal{M}(t) = M\ (M\ Int \rightarrow M\ Int) \rightarrow M\ (M\ Int \rightarrow M\ Int)$. Due to this non-distributive result, we have to insert `flatten` and `return` operators when monadifying a higher-order function in the (Var) rule via the `pos` and `neg` functions. Figure 4 highlights the general idea. Basically, the `neg` function maps an expression with type $\mathcal{M}(St)$ to one with type $\mathcal{M}(S)\mathcal{M}(t)$; the `pos` function works for the other direction of mapping.

Given the above formal definitions, we can derive the first key property of our monadification scheme which ensures that the type of a monadified expression is the same as the monadic type assigned to the original expression.

**Theorem 1 (Type Preservation)** *Given an expression $e$ and a type environment $\Gamma$, if $\Gamma \vdash e : t$ then $\mathcal{M}(\Gamma) \vdash [\![e]\!]^t_\Gamma : \mathcal{M}(t)$,*

*3.2.2 Dynamic Semantics and Value Preservation*

This section presents a small-step operational semantics for evaluating expressions in a lazy way. Similar to the static semantics, pure expressions, effectual expressions, and monadic expressions are all included so that we can reason about their evaluation in the same framework. Hence we shall simply refer to them as expressions if the context does not require distinguishing them.

By the nature of our aspect programs, there are three kinds of observable entity in our semantics, namely value, store, and output stream. A side-effecting aspect usually works by altering the store or the output stream but leaves the value untouched. Such aspects are considered semantically non-interfering or harmless [2]. Obviously, our monadification transformation should not turn a non-interfering expression into an interfering one. Thus we shall prove that the value of a pure expression is preserved by our monadification transformation. This is another major result about the correctness of our monadification scheme.

Figure 5 shows the semantic domains. There is a heap domain mapping variables to their defining expressions. It is used for modeling sharing in lazy evaluation. The expression deposited into a heap cell will be evaluated to a *heap value* when the associated variable is referenced. The store keeps the values of mutable variables in a record, and the output stream is a list of pairs of strings generated by invocations of putMsg. The set of values an expression evaluates to is standard, yet we need another set of heap values, which comprises standard values as well as computation encapsulated in the monad in the form of $e \gg\!= e'$. Such encapsulated computations are considered a form of functional (heap) values because their evaluation requires a monadic context to be supplied.

| | | |
|---|---|---|
| Heap | $h$ | $::= \text{Var} \mapsto \text{Expressions}$ |
| Store | $S$ | $::= \{var_1 = v_1, \cdots, var_n = v_n\}$ |
| Output Stream | $\mathcal{O}$ | $::= [(\text{String}, \text{String})]$ |
| Value | $v$ | $::= c \mid \lambda x.e \mid \mathsf{return}\ v$ |
| Heap Value | $v_h$ | $::= c \mid \lambda x.e \mid \mathsf{return}\ v_h \mid e \gg\!= e'$ |

**Fig. 5** Semantics Domains for the Operational Semantics

The operational semantics specifies the evaluation of an expression in terms of the following two kinds of transition relations over the configurations $(h, S, \mathcal{O}, e)$.

$$\text{Global step (reduces to } v\text{): } (h, S, \mathcal{O}, e_1) \mapsto (h', S', \mathcal{O}', e_2)$$

$$\text{Heap cell step (reduces to } v_h\text{): } (h, S, \mathcal{O}, e_1) \underset{h}{\mapsto} (h', S', \mathcal{O}', e_2)$$

The reason to have an additional heap cell step is to accommodate the difference between values and heap values, as mentioned above. Figure 6 and Figure 7 display the axioms and rules for defining these two transition relations. Since the axioms and rules for defining them are identical except for monadic expressions, we shall use $\underset{?}{\mapsto}$ to stand for both $\mapsto$ and $\underset{h}{\mapsto}$ in those identical cases. Given an empty heap, an initial store, $S$, of all user-defined mutable variables, an empty output stream, and an expression, $e$, these rules specify the individual steps of the evaluation of $e$. Obviously, we are interested in the case when $e$ is reduced to a value, $v$, with output stream, $\mathcal{O}$, and store, $S'$, after a finite number of steps of evaluation as follows.

$$e \overset{\mathcal{O}}{\longmapsto} v \quad \text{iff} \quad (\emptyset, S, [\,], e) \mapsto^* (h, S', \mathcal{O}, v)$$

Note that the above sequence is a sequence of global steps, but it may employ heap cell steps to reduce expressions kept in the heap. We do not include the stores on the

left-hand side because the contents of the store are not observable when the evaluation is done.

Rules for common expressions ($\underset{?}{\mapsto}$ stands for both $\mapsto$ and $\underset{h}{\mapsto}$)

$$(\text{OS:App1})(h, S, \mathcal{O}, (\lambda x.e)\ e_1) \underset{?}{\mapsto} (h[x \mapsto e_1], S, \mathcal{O}, e)$$

$$(\text{OS:App2})\frac{(h, S, \mathcal{O}, e_1) \underset{?}{\mapsto} (h', S', \mathcal{O}', e_1')}{(h, S, \mathcal{O}, e_1\ e_2) \underset{?}{\mapsto} (h', S', \mathcal{O}', e_1'\ e_2)}$$

$$(\text{OS:If1})\frac{b = True \text{ or } b = False}{(h, S, \mathcal{O}, \mathsf{if}\ b\ \mathsf{then}\ e_{True}\ \mathsf{else}\ e_{False}) \underset{?}{\mapsto} (h, S, \mathcal{O}, e_b)}$$

$$(\text{OS:If2})\frac{(h, S, \mathcal{O}, a) \underset{?}{\mapsto} (h', S', \mathcal{O}', a')}{(h, S, \mathcal{O}, \mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \underset{?}{\mapsto} (h', S', \mathcal{O}', \mathsf{if}\ a'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)}$$

$$(\text{OS:Prim})\frac{(h, S, \mathcal{O}, e_1) \mapsto (h', S', \mathcal{O}', e_2)}{(h, S, \mathcal{O}, p\ e_1) \underset{?}{\mapsto} (h', S', \mathcal{O}', p\ e_2)} \quad \text{for primitive } p$$

$$(\text{OS:hval})\frac{h(x) = v_h}{(h, S, \mathcal{O}, x) \underset{?}{\mapsto} (h, S, \mathcal{O}, v_h)} \qquad (\text{OS:heval})\frac{(h[x \mapsto \bot], S, \mathcal{O}, h(x)) \underset{h}{\mapsto} (h', S', \mathcal{O}', e)}{(h, S, \mathcal{O}, x) \underset{?}{\mapsto} (h'[x \mapsto e], S', \mathcal{O}', x)}$$

$$(\text{OS:Let})(h, S, \mathcal{O}, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \underset{?}{\mapsto} (h[x \mapsto e_1], S, \mathcal{O}, e_2)$$

**Fig. 6** Semantic Rules for Common Expressions

The rules in Figure 6 for common expressions are pretty standard. The (OS: App1) and (OS:App2) rules are the congruence rule and computation rule for reducing an application, respectively. Similar rules exist for reducing an if-expression. For primitives, we list only a template congruence rule. The remaining three rules are inter-related. The (OS:Let) rule deposits the expression of a let-binding into a new cell in the heap. Later, when the variable is referenced, if the expression associated with it is already a heap value, then (OS:hval) will simply return the heap value. Otherwise, the (OS:heval) rule will employ the heap step transition rules to evaluate the expression repeatedly until a heap value is reached, and then update the underlying heap cell using the value, thus achieving the sharing required for future references to the variable.

There are two groups of rules in Figure 7. The first group provides the evaluation rules for the source-level effectual constructs, including those of the state API. The *Cons* operator in the (OS:Put) rule is the list constructor operator. The setter and getter for accessing a user-defined mutable variable $F$ are denoted by primitives $\mathsf{set}F$ and $\mathsf{get}F$, respectively. We write $(S[F \mapsto v])$ for updating the variable $F$ in the store, and $(proj_F\ S)$ for retrieving its value from the store. The second group specifies the evaluation rules for monadic expressions produced by the monadification transformation, following the standard monadic semantics.

Rules for effectual expressions, including the state API:

$$(\text{OS:SEQ1})(h, S, \mathcal{O}, v;\ e) \underset{?}{\mapsto} (h, S, \mathcal{O}, e) \qquad (\text{OS:SEQ2})\frac{(h, S, \mathcal{O}, e_1) \underset{?}{\mapsto} (h', S', \mathcal{O}', e_3)}{(h, S, \mathcal{O}, e_1;\ e_2) \underset{?}{\mapsto} (h', S', \mathcal{O}', e_3;\ e_2)}$$

$$(\text{OS:SET1})(h, S, \mathcal{O}, \mathsf{set!}\ x = v;\ e) \underset{?}{\mapsto} (h, S, \mathcal{O}, [v/x]e)$$

$$(\text{OS:SET2})\frac{(h, S, \mathcal{O}, e_1) \mapsto (h', S', \mathcal{O}', e_3)}{(h, S, \mathcal{O}, \mathsf{set!}\ x = e_1;\ e_2) \underset{?}{\mapsto} (h', S', \mathcal{O}', \mathsf{set!}\ x = e_3;\ e_2)}$$

$$(\text{OS:PUT})(h, S, \mathcal{O}, \mathsf{putMsg}\ v_1\ v_2) \underset{?}{\mapsto} (h, S, Cons\ (v_1, v_2)\ \mathcal{O}, ())$$

$$(\text{OS:SETTER})(h, S, \mathcal{O}, \mathsf{set}F\ v_1) \underset{?}{\mapsto} (h, S[F \mapsto v_1], \mathcal{O}, ())$$

$$(\text{OS:GETTER})(h, S, \mathcal{O}, \mathsf{get}F) \underset{?}{\mapsto} (h, S, \mathcal{O}, proj_F\ S)$$

Rules for monadic expressions (global step only):

$$(\text{OS:RET})(h, S, \mathcal{O}, \mathsf{return}\ v \gg\!= e_2) \mapsto (h, S, \mathcal{O}, e_2\ v)$$

$$(\text{OS:BIND})\frac{(h, S, \mathcal{O}, e_1) \mapsto (h', S', \mathcal{O}', e_3)}{(h, S, \mathcal{O}, e_1 \gg\!= e_2) \mapsto (h', S', \mathcal{O}', e_3 \gg\!= e_2)}$$

**Fig. 7** Semantic Rules for State-related and Monadic Expressions

Let us use the following A-normalized expression and its monadified version to illustrate the operational semantics. This expression is a miniature version of applying the profiler aspect to the function, $d$, using a mutable variable $c$.

$$\begin{array}{l} e \equiv \mathsf{let}\ d = \lambda x.(\mathsf{let}\ v_1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v_1\ ; x + x) \\ \quad \mathsf{in}\ \mathsf{let}\ v_2 = d\ 2 \\ \quad\quad \mathsf{in}\ \mathsf{let}\ v_3 = d\ 3 \\ \quad\quad\quad \mathsf{in\ set!}\ v = v_2 * v_3\ ;\ \mathsf{let}\ v_4 = (\mathsf{show\ getC})\ \mathsf{in}\ (\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)\ ; v \end{array}$$

As the complete evaluation of $e$ is long and tedious, we take the following approach to simplify its presentation. First, we specify in detail the evaluation of the expression, $\mathsf{let}\ d = \lambda x.(\mathsf{let}\ v_1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v_1\ ; x + x)\ \mathsf{in}\ (d\ 2)$, which is a key part of the expression $e$. Then we outline the major steps of the complete evaluation of $e$. Both parts assume an initial store $\bot[c \mapsto 0]$, which will be abbreviated as $\{c = 0\}$. The first part is as follows.

$$\begin{array}{ll} & (\emptyset, \{c = 0\}, [\,], \mathsf{let}\ d = \lambda x.(\mathsf{let}\ v_1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v_1\ ; x + x)\ \mathsf{in}\ d\ 2) \\ & (\text{OS:LET}) \\ \mapsto & (h, \{c = 0\}, [\,], d\ 2) \\ & \quad \mathsf{where}\ h = [d \mapsto \lambda x.(\mathsf{let}\ v1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v1\ ; x + x)] \\ & (\text{OS:APP2}), (\text{OS:HVAL}) \\ \mapsto & (h, \{c = 0\}, [\,], (\lambda x.(\mathsf{let}\ v_1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v_1\ ; x + x))\ 2) \\ & (\text{OS:APP1}) \\ \mapsto & (h[x \mapsto 2], \{c = 0\}, [\,], \mathsf{let}\ v_1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v_1\ ; x + x) \end{array}$$

$$
\begin{array}{l}
\quad (\text{OS:LET}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto \mathsf{getC} + 1], \{c = 0\}, [\,], \mathsf{setC}\ v_1\ ; x + x) \\
\quad (\text{OS:SEQ2}), (\text{OS:PRIM}), (\text{OS:HEVAL}), (\text{OS:GETTER}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 0 + 1], \{c = 0\}, [\,], \mathsf{setC}\ v_1\ ; x + x) \\
\quad (\text{OS:SEQ2}), (\text{OS:PRIM}), (\text{OS:HEVAL}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 0\}, [\,], \mathsf{setC}\ v_1\ ; x + x) \\
\quad (\text{OS:SEQ2}), (\text{OS:HVAL}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 0\}, [\,], \mathsf{setC}\ 1\ ; x + x) \\
\quad (\text{OS:SEQ2}), (\text{OS:SETTER}), \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 1\}, [\,], (); x + x) \\
\quad (\text{OS:SEQ1}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 1\}, [\,], x + x) \\
\quad (\text{OS:PRIM}), (\text{OS:HVAL}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 1\}, [\,], 2 + x) \\
\quad (\text{OS:PRIM}), (\text{OS:HVAL}) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 1\}, [\,], 2 + 2) \\
\mapsto (h[x \mapsto 2][v_1 \mapsto 1], \{c = 1\}, [\,], 4)
\end{array}
$$

Next, we show the major steps for evaluating $e$. The sub-expression to be evaluated at each major step is underlined to help the reader find the points quickly.

$$
\begin{array}{l}
\quad (\emptyset, \{c = 0\}, [\,], e) \\
\mapsto^* (h, \{c = 0\}, [\,], \mathsf{set!}\ v = \underline{v_2} * v_3\ ; \mathsf{let}\ v_4 = (\mathsf{show}\ \mathsf{getC})\ \mathsf{in}\ (\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)\ ; v) \\
\quad \text{where } h = [d \mapsto \lambda x.(\mathsf{let}\ v1 = \mathsf{getC} + 1\ \mathsf{in}\ \mathsf{setC}\ v1\ ; x + x), v_2 \mapsto (d\ 2), v_3 \mapsto (d\ 3)] \\
\mapsto^* (h[v_1 \mapsto 1][v_2 \mapsto 4], \{c = 1\}, [\,], \\
\qquad\qquad \mathsf{set!}\ v = 4 * \underline{v_3}\ ; \mathsf{let}\ v_4 = (\mathsf{show}\ \mathsf{getC})\ \mathsf{in}\ (\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)\ ; v) \\
\mapsto^* (h[v_1 \mapsto 2][v_2 \mapsto 4][v_3 \mapsto 6], \{c = 2\}, [\,], \\
\qquad\qquad \underline{\mathsf{set!}\ v = 4 * 6}; \mathsf{let}\ v_4 = (\mathsf{show}\ \mathsf{getC})\ \mathsf{in}\ (\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)\ ; v) \\
\mapsto^* (h[v_1 \mapsto 2][v_2 \mapsto 4][v_3 \mapsto 6], \{c = 2\}, [\,], \\
\qquad\qquad \underline{\mathsf{let}\ v_4 = (\mathsf{show}\ \mathsf{getC})\ \mathsf{in}\ (\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)}; \ 24) \\
\mapsto^* (h[v_1 \mapsto 2][v_2 \mapsto 4][v_3 \mapsto 6][v_4 \mapsto \text{“2”}], \{c = 2\}, [\,], \underline{(\mathsf{putMsg}\ \text{“}main\text{”}\ v_4)}; 24) \\
\mapsto^* (h[v_1 \mapsto 2][v_2 \mapsto 4][v_3 \mapsto 6][v_4 \mapsto \text{“2”}], \{c = 2\}, [(\text{“}main\text{”}, \text{“2”})], 24)
\end{array}
$$

Hence, the value of $e$ is 24, and the output stream is $[(\text{“}main\text{”}, \text{“2”})]$:

$$e \overset{[(\text{“}main\text{”}, \text{“2”})]}{\longmapsto} 24$$

Now, consider the monadified version of $e$:

$$
\begin{array}{l}
[\![e]\!]^{Int}_{\Gamma} = \mathsf{let}\ d = \lambda x.\mathsf{let}\ v_1 = \mathsf{liftM2}\ (+)\ \mathsf{getC}\ (\mathsf{return}\ 1) \\
\qquad\qquad\qquad\qquad \mathsf{in}\ (\mathsf{setC}\ v_1 \gg\!= \lambda\_.(\mathsf{liftM2}\ (+)\ x\ x)) \\
\qquad\quad \mathsf{in}\ \mathsf{let}\ v_2 = (d\ (\mathsf{return}\ 2))\ \mathsf{in}\ \mathsf{let}\ v_3 = (d\ (\mathsf{return}\ 3))\ \mathsf{in}\ (\mathsf{liftM2}\ (*)\ v_2\ v_3) \gg\!= E \\
\qquad\quad \text{where } E = \lambda v'.\ \mathsf{let}\ v = \mathsf{return}\ v' \\
\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{let}\ v_4 = (\mathsf{liftM}\ \mathsf{show}\ \mathsf{getC}) \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\mathsf{liftM2}\ \mathsf{putMsg}\ (\mathsf{return}\ \text{“}main\text{”})\ v_4) \\
\qquad\qquad\qquad \gg\!= \lambda\_.v
\end{array}
$$

The major evaluation steps of the monadified expression are as follows.

$$
\begin{array}{ll}
 & (\emptyset, \{c = 0\}, [\,], [\![e]\!]_{\Gamma}^{Int}) \\
\mapsto^3 & (h, \{c = 0\}, [\,], \mathsf{liftM2}\ (*)\ v_2\ v_3 \gg\!= E) \\
 & \quad \mathsf{where}\ h = [d \mapsto \lambda x.\, \mathsf{let}\ v_1 = \mathsf{liftM2}\ (+)\ \mathsf{getC}\ (\mathsf{return}\ 1) \\
 & \qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{setC}\ v_1 \gg\!= \lambda\_.(\mathsf{liftM2}\ (+)\ x\ x), \\
 & \qquad\qquad v_2 \mapsto d\ (\mathsf{return}\ 2), v_3 \mapsto d\ (\mathsf{return}\ 3)] \\
\mapsto^* & (h, \{c = 0\}, [\,], \mathsf{liftM2}\ (*)\ ((\lambda x.\, \mathsf{let}\ v_1 = \mathsf{liftM2}\ (+)\ \mathsf{getC}\ (\mathsf{return}\ 1) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ (\mathsf{setC}\ v_1 \gg\!= \lambda\_.(\mathsf{liftM2}\ (+)\ x\ x)))\ (\mathsf{return}\ 2))\ v_3 \\
 & \qquad \gg\!= E) \\
\mapsto^2 & (h[v_1 \mapsto \mathsf{liftM2}\ (+)\ \mathsf{getC}\ (\mathsf{return}\ 1)], \{c = 0\}, [\,], \\
 & \qquad \mathsf{liftM2}\ (*)\ (\mathsf{setC}\ v_1 \gg\!= \lambda\_.(\mathsf{liftM2}\ (+)\ (\mathsf{return}\ 2)\ (\mathsf{return}\ 2)))\ v_3 \gg\!= E) \\
\mapsto^* & (h[v_1 \mapsto (\mathsf{return}\ 1)], \{c = 1\}, [\,], \mathsf{liftM2}\ (*)\ (\mathsf{return}\ 4)\ v_3 \gg\!= E) \\
\mapsto^* & (h[v_1 \mapsto (\mathsf{return}\ 1)], \{c = 2\}, [\,], \\
 & \quad \mathsf{liftM2}\ (*)\ (\mathsf{return}\ 4)\ (\mathsf{return}\ 6) \gg\!= \\
 & \quad \lambda v'.\, \mathsf{let}\ v = \mathsf{return}\ v'\ \mathsf{in} \\
 & \qquad \mathsf{let}\ v_4 = (\mathsf{liftM}\ \mathsf{show}\ \mathsf{getC})\ \mathsf{in}\ \mathsf{liftM2}\ \mathsf{putMsg}\ (\mathsf{return}\ \text{“}main\text{”})\ v_4 \gg\!= \lambda\_.v) \\
\mapsto^* & (h', \{c = 2\}, [\,], \mathsf{liftM2}\ \mathsf{putMsg}\ (\mathsf{return}\ \text{“}main\text{”})\ v_4) \gg\!= \lambda\_.v) \\
 & \quad \mathsf{where}\ h' = h[v_1 \mapsto (\mathsf{return}\ 1)][v \mapsto (\mathsf{return}\ 24)][v_4 \mapsto (\mathsf{liftM}\ \mathsf{show}\ \mathsf{getC})] \\
\mapsto^* & (h'[v_4 \mapsto (\mathsf{return}\ \text{“2”})], \{c = 2\}, [\,], \mathsf{liftM2}\ \mathsf{putMsg}\ (\mathsf{return}\ \text{“}main\text{”})\ v_4) \gg\!= \lambda\_.v) \\
\mapsto^* & (h'[v_4 \mapsto (\mathsf{return}\ \text{“2”})], \{c = 2\}, [(\text{“}main\text{”}, \text{“2”})], v) \\
\mapsto^* & (h'[v_4 \mapsto (\mathsf{return}\ \text{“2”})], \{c = 2\}, [(\text{“}main\text{”}, \text{“2”})], \mathsf{return}\ 24)
\end{array}
$$

Therefore, the value of $[\![e]\!]_{\Gamma}^{Int}$ is return 24, and the output stream is [(“*main*”, “2”)]:

$$[\![e]\!]_{\Gamma}^{Int}\ \overset{[(\text{“}main\text{”}, \text{“2”})]}{\longmapsto}\ \mathsf{return}\ 24$$

The resulting value is the monadic version of the value of the original expression. In other words, the miniature profiling aspect achieves its effects via the store and the output stream without altering the value of its target expression. In general, for pure expressions without any side-effecting components, their value will not be altered by monadification transformation, as shown by the following theorem.

**Theorem 2 (Value Preservation)** *Given a pure expression $e$ and a type environment $\Gamma$, if $\Gamma \vdash e : t$ and $e \overset{[]}{\longmapsto} v$ then $[\![e]\!]_{\Gamma}^{t} \overset{[]}{\longmapsto} \mathsf{return}\ v$*

## 4 State Monads for Side-Effecting Aspects

Given the monadification framework presented above, we now proceed to specialize it by introducing state monads in Haskell to support side-effecting aspects. We shall first present a basic state monad for illustrating our approach followed by a state monad enhanced with caching facility for preserving laziness of expression evaluation. The full Haskell code of our implementation is included in Appendix B.

### 4.1 Basic State Monad

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 refer to them as the *aspect state* and the state monad encapsulating them as the *aspect monad*.

Since the specific content of the user variable record depends on the individual program, we provide the following generic state monad, `GM v`, 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[4]. 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. Their Haskell code is as follows.

```
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 mutable 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 mutable 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 that 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 set!-expressions and sequencing expressions to enable the user to override the default lazy evaluation semantics when applying such state-aware functions.

In the previous section, the monadification of set!-expressions and sequencing expressions was presented in terms of the monad's $\gg=$ operation. From now on, we

[4] The code uses "cons", but we reverse the buffer when it is dumped at the end of program execution.

switch to Haskell's do-notation to present them as follows.

$$[\![\mathsf{set!}\ x\ =\ e_1\ ;\ e_2]\!]^t_\Gamma = \ \mathsf{do}\ \{x' \leftarrow [\![e_1]\!]^{t_1}_\Gamma;\ \mathsf{let}\ x = \mathsf{return}\ x';\ [\![e_2]\!]^t_\Gamma\}$$
$$\textit{where } x' \text{ is a fresh identifier}$$
$$[\![e_1;\ e_2]\!]^t_\Gamma = \ \mathsf{do}\ \{[\![e_1]\!]^{t_1}_\Gamma\ ;\ [\![e_2]\!]^t_\Gamma\}$$

Take the profiler aspect defined previously as an example. After monadification, the `profiler` aspect and its helper function `incProfile` are transformed into the following Haskell code.

```
profilerM :: (M Int -> M Int) -> (M Int -> M Int)
profilerM proceed arg = do incProfileM (return "fib")
                           proceed arg
incProfileM fname =
   do pMap'  <- getProfileMapM  --set! for getting profileMap's value
      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
```

The body of `profilerM` employs a sequencing expression. Hence its body becomes a do-expression after monadification. The `incProfile` uses a set!-expression on mutable variable `profileMap`, so its monadified version has a do-binding with `getProfileMapM`.

Another example is the eager tracing of function `fac`. The program in Example 2 is monadified into the following Haskell code.[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
     setIndentM ind'
     v_1'  <- arg             --set! v1 arg
     let v_1 = return  v_1'
     v_2'  <- arg2            --set! v1 arg
     let v_2 = return  v_2'
     let show_arg2 = (liftM show) v_2
     let str_1 = (liftM2 (++)) show_arg2 (return "]")
     let str_2 = (liftM2 (++)) (return ",") str_1
     let show_arg = (liftM show) v_1
     let str_3 = (liftM2 (++)) show_arg str_2
     let str_4 = (liftM2 (++)) (return "fac receives [") str_3
     let str_5 = (liftM2 (++)) ind str_4
```

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

```
     putMsgM (return "tracerFacM") str_5
     proceedResult  <- proceed v_1 v_2
     let result = return proceedResult
     setIndentM 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)
```

The use of set!-expression allows explicit control of evaluation order.

### 4.2 Cached State Monad for Preserving Laziness

We have seen that in addition to sequencing the desired order of evaluation within side-effecting aspects, explicit use of set!-expressions is able to influence the base program by evaluating the arguments of proceed prior to the call. At the same time, we also want the option of being able to write side-effecting aspects that do not interfere with the lazy semantics of their base program. This preservation of laziness turns out to be non-trivial to enforce because any reference to the arguments of an advice in a sequenced expression may force the evaluation of them. Consider a variant of Example 2.

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

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

We have removed the set!-expressions that evaluate the arguments eagerly, hoping to obtain a trace reflecting the lazy evaluation of `fac`.

As shown in [12], according to the lazy semantics, the tracing result of `(fac 3 1)` should be[6]

```
fac receives [3, 1]
| fac receives [2, 3*1]
| | fac receives [1, 2*(3*1)]
| | | fac receives [0, 1*(2*(3*1))]
| | | | times receives [1, 2*(3*1)]
| | | | | times receives [2, 3*1]
| | | | | | 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.

```
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
     setIndentM 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
     setIndentM 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 =
```

[6] To help the readers understand the lazy trace, we intentionally leave the accumulating parameter of `fac` not fully evaluated.

```
  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 the wrong order. 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 the tracing messages in the wrong order. This result is disturbing because the sole purpose of tracing is to track the evaluation steps of the underlying program and record them in the output stream.

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` (the accumulating parameter), which in turn 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 the `proceed` call, the multiplication call will be triggered and traced again. Hence the problem is how to preserve the lazy evaluation of the base program while monadifying aspects which are perceived to be non-interfering, such as tracing. Unfortunately, existing monadification schemes such as [14,5,7,9] 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 aggravates 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 the `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 in 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 8.

```
data Cell = forall s a. Cell Bool (CState s a)           -- Cells: thunks or values
type Cache = Map.Map Int (Maybe Cell)

newtype CState s a = CState{
     realrunCState :: (s, Cache) -> (Either a Int, (s, Cache))
}
type M a = CState (UserVar, OutputBuf) a

runCState :: CState s a -> (s, Cache) -> (a, (s, Cache)) --helper function for aspect monad
types runCState a (s, cs) = uncurry fromCacheEither $ realrunCState a (s, cs)

instance Monad (CState s) where                          -- Standard State monad impl.
  return t = CState $ \(s, cs) -> (Left t, (s, cs))
  ma >>= k = CState $ \(s, cs) -> let (a, (s', cs')) = runCState ma (s, cs)
                                  in realrunCState (k a) (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 :: 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
   --shown in the Appendix B
fromCell :: Cell -> (s, Cache) -> (Either a Int, (s, Cache))
fromCell (Cell _ c) = realrunCState (unsafeCoerce# c)

--Functions for manipulating the cache
getNewCacheLoc :: CState s Int                -- get a new cell loc from cache
setCache :: Int -> CState s a -> CState s a    -- 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
```

**Fig. 8** 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 a state monad extended with a cache of cells. When feeding an extended state, $(s, cache)$, 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. Because of the cache wrapper, we define a special "unpacker" function, `runCState`, to assist in realizing state processing for the extended aspect monad. Specifically, it first activates the state processing function via the field accessor, `realrunCState`, to obtain an either-object, and then passes it to the `fromCacheEither` function, which may look up the cell in the cache and trigger the monadic computation stored therein via the `fromCell` function. The definition of the bind operator ($\gg=$) of the new aspect monad is almost identical to the standard state monad except the call to `realrunCState`. Note that, due to the use of the `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 8 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 the two functions 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 can be resolved. Recall that, after A-normalization, all non-atomic function arguments will become let-bound expressions. Hence, we can enhance the monadification rewriting rule for let-expressions by applying the `add2Cache` function to fully applied function calls as follows.

$$
\begin{aligned}
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^t_\Gamma = \ & \mathit{if}\ e_1 \text{ is of functional type or a constant} \\
& \mathit{then}\ \mathsf{do}\ \{\mathsf{let}\ x = [\![e_1]\!]^{t_x}_\Gamma;\ [\![e_2]\!]^t_\Gamma\} \\
& \mathit{else}\ \mathsf{do}\ \{x \leftarrow \mathit{add2cache}\ \$\ [\![e_1]\!]^{t_x}_\Gamma;\ [\![e_2]\!]^t_\Gamma\}
\end{aligned}
$$

Following this enhancement, the revised monadification of the tracing program is as follows.

```
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 $ (liftM show) arg2
     str_1 <- add2Cache $ (liftM2 (++)) s_arg2 (return "]")
     str_2 <- add2Cache $ (liftM2 (++)) (return ",") str_1
     s_arg <- add2Cache $ (liftM show) 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 $ (liftM show) 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
```

Running the above code with `facM (return 3) (return 1)` will produce the following result, exactly the same as the eager trace.

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

Now the duplicated evaluations are eliminated, but the lifted `show` function still makes the tracing messages out of order. As mentioned above, 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, $n$ 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.

Now we can adapt our monadification scheme by treating the `show` function as a special primitive function and use this `showM` function as its monadified version. As

a result, the monadified tracing program will produce the same result as described in [12] when run with `facM (return 3) (return 1)`. On the other hand, on certain occasions, such as debugging as mentioned in the introduction, one may prefer an eager tracing of the base programs. Thus, we could also offer both options of monadifying `show`, namely `(liftM show)` and `showM`, and let the user decide which one to use.

## 5 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. We illustrate this by refactoring the monadic version of the "display update" example presented by Hofer and Ostermann [10].

The context of this "display update" example [11] 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 the `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. Example 3 displays 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

aspect DisplayUpdate where
  -- user variable
  var displayObject ::DisplayObject = DisplayObject EmptyDisplay

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

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

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

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

Here the mutable 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.

### 5.1 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:

$$\mathcal{M}(t_1 \to t_2) \Rightarrow \mathcal{M}(t_1) \to \mathcal{M}(t_2) \tag{4}$$

$$\mathcal{M}(a) \Rightarrow MT\ N\ a \tag{5}$$

$$\mathcal{M}(N\ (t_1 \to t_2)) \Rightarrow MT\ N\ (\mathcal{M}(t_1) \to \mathcal{M}(t_2)) \tag{6}$$

$$\mathcal{M}(N\ a) \Rightarrow MT\ N\ a \tag{7}$$

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, some of the monadification rewriting rules also need to be adjusted. There are three 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 the `liftM` operator to lift operations on the base monad before applying them, and use the `liftN` operator to lift results of computations in the base monad, `N`. The following enhanced versions of (Prim) and (App) illustrate the ideas.

$$
\begin{array}{ll}
\text{(Prim)} & [\![p]\!]^t_\Gamma \;=\; \texttt{liftMn}\ p \\
& \quad \textit{where}\ \text{n is the arity of the primitive function or} \\
& \qquad \text{the base monad operation } p \\
\text{(App)} & [\![e\ a]\!]^t_\Gamma \;= \\
& \quad \textit{if}\ isFullAppBaseMonadOP(e_1) \\
& \quad \textit{then}\ \mathsf{do}\ \{x \leftarrow [\![e]\!]^{t_a \to t}_\Gamma\ [\![a]\!]^{t_a}_\Gamma;\ \texttt{liftN}\ x\} \\
& \quad \textit{else}\ [\![e]\!]^{t_a \to t}_\Gamma\ [\![a]\!]^{t_a}_\Gamma
\end{array}
$$

Secondly, the revised rule for let-expressions presented in Section 4.2 needs yet another adjustment. Specifically, if the binding of a let-expression is a monadic expression, then we should not apply the `add2Cache` function, as the computation encapsulated in a monad should be evaluated whenever it is referenced.

Thirdly, we need to extend the rewriting rules to handle the bind ($\gg=$) and the return operations of the base monad.

$$
\begin{array}{lll}
\text{(Bind)} & [\![\mathsf{do}\ \{x \leftarrow e_1;\ e_2\}]\!]^t_\Gamma = & \mathsf{do}\ \{x' \leftarrow [\![e_1]\!]^{t_1}_\Gamma;\ \mathsf{let}\ x = \mathsf{return}\ x';\ [\![e_2]\!]^t_\Gamma\} \\
\text{(Return)} & [\![\mathsf{return}\ e]\!]^t_\Gamma = & [\![e]\!]^t_\Gamma
\end{array}
$$

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 returnof 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
```

## 5.2 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 generalize the state monad along another direction using monad transformers. 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 the previous subsections.

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

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

2. The aspect monad of Section 4.2 can be replaced with the following one:

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

3. The aspect monad of Section 5.1 can be replaced with the following one:

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

Figure 9 shows a summary of the monads we developed along the way towards our goal.

## 6 Related Work

Research about monadification can be traced back to work on continuation passing style conversion [7,9], where compiler-based transformation rules were developed to convert all functions and intermediate results in a program into monadic form. The A-normalization technique was introduced in [7]. 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 [14], in which a set of type-directed transformation rules were devised to

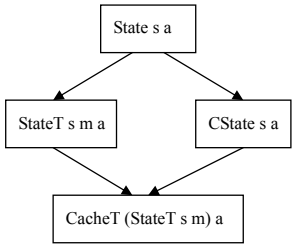


**Fig. 9** Summary of the Monad Transformations

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. In [5], Erwig and Ren 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 these approaches addressed the issues related to lazy semantics.

In this work, the monadification transformation is performed after type inference and after 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 previous approaches by also lifting function parameters to the monadic space. 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.

Fischer et al. [6] presented an efficient implementation of non-strictness, sharing and non-determinism embedded in a purely functional language, such as Haskell. They devised some customized monadic data types to support non-determinism in non-strict context. To enable explicit sharing, a combinator, `share`, is supplied to introduce variables for non-deterministic computations that represent values rather than computations. Not surprisingly, their `share` combinator plays the same role as our `add2Cache` function, as manifested by their type signature, `m a -> m (m a)`, where `m` is instantiated to `CState s` in our case. Indeed, there is a close correspondence between their monadic implementation of the sharing facility and our cache-extended state monad: Both have an implementation of thunk stores with respect to a monad.

The Functional Programming Group at Kent University maintains a web page titled, Monadification as Refactoring, which collects five different styles of monadification and uses a simple interpreter to illustrate these styles. Our monadification scheme presented in Section 3.1 is referred to as restricted call-by-name monadification, and the other so-called full call-by-name monadification is the one that we had considered but not adopted.

Kishon's thesis work [12,13] 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 at the source level. Hence it is easier for his framework to utilize semantic entities such as the 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 [17]. The recent work of Hofer and Ostermann [10] explored this subject in further depth and presented a detailed 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.

## 7 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 the implementation of monitoring by translating the woven code to monadic style purely 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 that employs a cached state monad transformer 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. The generated monadic code reveals further opportunities for optimizations. For example, 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. At the moment, 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 optimizations of the monadic code via some static analysis techniques. In particular, we speculate that the type-based usage analysis developed in [22] can be adapted to serve our purpose.

At the moment, all mutable variables are strictly private to aspects, which spare us from checking non-interference between aspects accessing the same state at the cost of compromised modularity. We plan to relax this restriction in a controlled manner. One plausible direction is to allow explicit inheritance of states between aspects. In this case, non-interference can be reasoned about with the framework in [18].

## References

1. Chen, K., S.-C. Weng, M. Wang, S.-C. Khoo, and C.-H. Chen: 2007, 'A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages'. In: *Static Analysis, 14th International Symposium, SAS 2007*, Vol. 4634 of *LNCS*. pp. 34–51.
2. Dantas, D. S. and D. Walker: 2006, 'Harmless Advice'. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. pp. 383–396.
3. Dantas, D. S., D. Walker, G. Washburn, and S. Weirich: 2008, 'AspectML: A polymorphic aspect-oriented functional programming language'. *ACM Trans. Program. Lang. Syst.* **30**(3), 1–60.

4. Ennals, R. and S. P. Jones: 2003, 'HsDebug : Debugging Lazy Programs by Not Being Lazy'. In: *In Haskell 03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. pp. 84–87.
5. Erwig, M. and D. Ren: 2004, 'Monadification of Functional Programs'. *Science of Computer Programming* **52**(1-3), 101–129.
6. Fischer, S., O. Kiselyov, and C.-C. Shan: 2009, 'Purely functional lazy non-deterministic programming'. In: *ICFP '09: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA, pp. 11–22.
7. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen: 1993, 'The essence of compiling with continuations'. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. pp. 237–247.
8. Functional Programming Group, K. U.: 2005, 'Monadification as a refactoring'.
9. Hatcliff, J. and O. Danvy: 1993, 'A generic account of continuation-passing styles'. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 237–247.
10. Hofer, C. and K. Ostermann: 2007, 'On the Relation of Aspects and Monads'. In: *Foundations of Aspect-Oriented Languages Workshop at AOSD*. pp. 37–46.
11. Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold: 2001, 'An Overview of AspectJ'. In: *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, Vol. 2072 of *LNCS*. pp. 327–353.
12. Kishon, A.: 1992, 'Theory and Art of Semantics-Directed Program Execution Monitoring'. Ph.D. thesis, Yale University.
13. Kishon, A. and P. Hudak: 1995, 'Semantics Directed Program Execution Monitoring'. *Journal of Functional Programming* **5**(4), 501–547.
14. Lämmel, R.: 1999, 'Reuse by Program Transformation'. In: *Trends in Functional Programming: Vol. 1, Selected papers from the 1st Scottish Functional Programming Workshop*. pp. 144–153.
15. Launchbury, J.: 1993, 'A Natural Semantics for Lazy Evaluation'. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. pp. 144–154.
16. Masuhara, H., H. Tatsuzawa, and A. Yonezawa: 2005, 'Aspectual Caml: an aspect-oriented functional language'. In: *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA, pp. 320–330.
17. Meuter, W. D.: 1997, 'Monads as a theoretical foundation for AOP'. In: *International Workshop on Aspect-Oriented Programming at ECOOP*.
18. Oliveira, B. C. d. S., T. Schrijvers, and W. R. Cook: 2010, 'EffectiveAdvice: Disciplined Advice with Explicit Effects'. In: *ACM SIG Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*.
19. Wadler, P.: 1992, 'The Essence of Functional Programming'. In: *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 1–14.
20. Wang, M., K. Chen, and S.-C. Khoo: 2006, 'Type-Directed Weaving of Aspects for Higher-order Functional Languages'. In: *PEPM '06: Workshop on Partial Evaluation and Program Manipulation*. pp. 78–87.
21. Wang, M. and B. C. d. S. Oliveira: 2009, 'What does aspect-oriented programming mean for functional programmers?'. In: *WGP '09: Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*. New York, NY, USA, pp. 37–48.
22. Wansbrough, K. and S. Peyton Jones: 1999, 'Once Upon a Polymorphic Type'. In: *Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 15–28.

## A Proofs of Monadification Properties

In this appendix, we give in detail the proofs of type preservation and value preservation for our monadification scheme, respectively.

First, a key step towards proving the type preservation of our monadification scheme is to prove the type preservation of the monadified expression produced by the (Var) rule. We begin with the following lemma which shows that application of the type-specific `flatten` combinator inserted by the `pos` function can reconcile the type mismatch described in the main text for the base case of type substitution on a type variable.

**Lemma 1** *Given an expression $e$, a type environment $\Gamma$, a type substitution $S$, and a type variable $a \in dom(S)$, if $\mathcal{M}(\Gamma) \vdash e : \mathcal{M}(S)\mathcal{M}(a)$, then $\mathcal{M}(\Gamma) \vdash \texttt{flatten[}Sa\texttt{]}(e) : \mathcal{M}(Sa)$.*

*Proof* If $Sa$ is an atomic type, then the result follows trivially since $\mathcal{M}(S)\mathcal{M}(a) = M\ (Sa) = \mathcal{M}(Sa)$. Otherwise, assume that $Sa = t_1 \to t_2 \cdots t_n \to t$. Then it follows from

$$\mathcal{M}(\Gamma), x_1 : \mathcal{M}(t_1), \cdots, x_n : \mathcal{M}(t_n) \vdash e \gg\!= \lambda e'.\ e'\ x_1 \cdots x_n : \mathcal{M}(t).$$

□

By replacing $Sa$ in Lemma 1 with type $t$, we obtain the following corollary which shows that, when necessary, the type-specific combinator `flatten` can remove one level of monadic structure from the monadic type of the given expression.

**Corollary 1 (Type of `flatten[`$t$`]`)** *If $t$ is an atomic type, then $\mathcal{M}(\Gamma) \vdash e : M\ t$ implies $\mathcal{M}(\Gamma) \vdash \texttt{flatten[}t\texttt{]}(e) : M\ t$; otherwise, $\mathcal{M}(\Gamma) \vdash e : M\ (\mathcal{M}(t))$ implies $\mathcal{M}(\Gamma) \vdash \texttt{flatten[}t\texttt{]}(e) : \mathcal{M}(t)$.*

The second lemma shows the type preservation of our rule for monadifying variables.

**Lemma 2 (Type preservation for** (Var)**)** *Given a variable $x$, a type environment $\Gamma$, and a type $t$, if $x \in dom(\Gamma)$ and $t$ is an instance of $\Gamma(x)$, then*

$$\mathcal{M}(\Gamma) \vdash [\![x]\!]^t_\Gamma : \mathcal{M}(t)$$

*Proof* We prove the following two properties of $\texttt{pos}^S_t$ and $\texttt{neg}^S_t$ simultaneously by a structural induction on $t$. The lemma is a consequence of (a).

Given any type environment $\Gamma$, expression $e$, type $t$, and type substitution $S$,

(a) if $\mathcal{M}(\Gamma) \vdash e : \mathcal{M}(S)\mathcal{M}(t)$ then $\mathcal{M}(\Gamma) \vdash \texttt{pos}^S_t(e) : \mathcal{M}(St)$, and
(b) if $\mathcal{M}(\Gamma) \vdash e : \mathcal{M}(St)$ then $\mathcal{M}(\Gamma) \vdash \texttt{neg}^S_t(e) : \mathcal{M}(S)\mathcal{M}(t)$.

1. $t$ is a type variable $a$ in the domain of $S$:
   (a) As $\texttt{pos}^S_a(e) = \texttt{flatten[}Sa\texttt{]}\ (e)$, by lemma 1, $\mathcal{M}(\Gamma) \vdash \texttt{pos}^S_a(e) : \mathcal{M}(Sa)$.
   (b) If $Sa$ is an atomic type, then $\texttt{neg}^S_a(e) = e$, $\mathcal{M}(S)a = Sa$, $\mathcal{M}(S)\mathcal{M}(a) = M\ Sa = \mathcal{M}(Sa)$. Hence (b) is true. Otherwise, $\texttt{neg}^S_a(e) = \mathsf{return}\ e$, $\mathcal{M}(S)a = \mathcal{M}(Sa)$, and $\mathcal{M}(\Gamma) \vdash \mathsf{return}\ e : M\ (\mathcal{M}(Sa)) = \mathcal{M}(S)\mathcal{M}(a)$ also holds.
2. $t$ is an atomic type and $St = t$: then $\mathcal{M}(S)\mathcal{M}(t) = \mathcal{M}(t)$, and both (a) and (b) follow trivially.
3. $t = t_1 \to t_2$:
   (a) $\texttt{pos}^S_{t_1 \to t_2}(e) = \lambda x.\ \texttt{pos}^S_{t_2}(e\ \texttt{neg}^S_{t_1}(x))$:

$$\begin{array}{rrcll}
 & \mathcal{M}(\Gamma) \vdash & \lambda x.\ \texttt{pos}^S_{t_2}(e\ \texttt{neg}^S_{t_1}(x)) & : \mathcal{M}(St_1) \to \mathcal{M}(St_2) & \\
\Leftarrow & \mathcal{M}(\Gamma).x : \mathcal{M}(St_1) \vdash & \texttt{pos}^S_{t_2}(e\ \texttt{neg}^S_{t_1}(x)) & : \mathcal{M}(St_2) & \\
\Leftarrow & \mathcal{M}(\Gamma).x : \mathcal{M}(St_1) \vdash & e\ \texttt{neg}^S_{t_1}(x) & : \mathcal{M}(S)\mathcal{M}(t_2) & \text{by I.H.(a) on } t_2 \\
\Leftarrow & (\mathcal{M}(\Gamma).x : \mathcal{M}(St_1) \vdash & e & : \mathcal{M}(S)\mathcal{M}(t_1) \to \mathcal{M}(S)\mathcal{M}(t_2) & \\
\wedge & \mathcal{M}(\Gamma).x : \mathcal{M}(St_1) \vdash & \texttt{neg}^S_{t_1}(x) & : \mathcal{M}(S)\mathcal{M}(t_1)) & \text{by I.H.(b) on } t_1
\end{array}$$

   (b) $\texttt{neg}^S_{t_1 \to t_2}(e) = \lambda x.\ \texttt{neg}^S_{t_2}(e\ \texttt{pos}^S_{t_1}(x))$:

$$\begin{array}{rrcll}
 & \mathcal{M}(\Gamma) \vdash & \lambda x.\ \texttt{neg}^S_{t_2}(e\ \texttt{pos}^S_{t_1}(x)) & : \mathcal{M}(S)\mathcal{M}(t_1) \to \mathcal{M}(S)\mathcal{M}(t_2) & \\
\Leftarrow & \mathcal{M}(\Gamma).x : \mathcal{M}(S)\mathcal{M}(t_1) \vdash & \texttt{neg}^S_{t_2}(e\ \texttt{pos}^S_{t_1}(x)) & : \mathcal{M}(S)\mathcal{M}(t_2) & \\
\Leftarrow & \mathcal{M}(\Gamma).x : \mathcal{M}(S)\mathcal{M}(t_1) \vdash & e\ \texttt{pos}^S_{t_1}(x) & : \mathcal{M}(St_2) & \text{by I.H.(b) on } t_2 \\
\Leftarrow & (\mathcal{M}(\Gamma).x : \mathcal{M}(S)\mathcal{M}(t_1) \vdash & e & : \mathcal{M}(St_1) \to \mathcal{M}(St_2) & \\
\wedge & \mathcal{M}(\Gamma).x : \mathcal{M}(St_1) \vdash & \texttt{pos}^S_{t_1}(x) & : \mathcal{M}(St_1)) & \text{by I.H.(a) on } t_1
\end{array}$$

□

Given the two lemmas above, we can prove the first theorem, which ensures that the type of a monadified expression is the same as the monadic type assigned to the original expression.

**Theorem 1 (Type Preservation of Monadification)** *Given an expression $e$ and a type environment $\Gamma$, if $\Gamma \vdash e : t$ then $\mathcal{M}(\Gamma) \vdash [\![e]\!]^t_\Gamma : \mathcal{M}(t)$,*

*Proof* By induction on the derivation of $\Gamma \vdash e : t$.

(Var) Lemma 2

(Lam) By induction, $\mathcal{M}(\Gamma).x : \mathcal{M}(t_1) \vdash [\![e]\!]^{t_2}_\Gamma : \mathcal{M}(t_2)$. Hence, $\mathcal{M}(\Gamma) \vdash \lambda x.[\![e]\!]^{t_2}_\Gamma : \mathcal{M}(t_1) \to \mathcal{M}(t_2)$. The case follows since $\lambda x.[\![e]\!]^{t_2}_\Gamma = [\![\lambda x.e]\!]^{t_1 \to t_2}_\Gamma$.

(App) Trivial.

(If) Trivial if the rule (If-C) is applied. Otherwise, by induction hypothesis, $\mathcal{M}(\Gamma) \vdash [\![a]\!]^t_\Gamma : \mathcal{M}(\mathit{Bool})$. Applying the rule (Bind) with $\mathcal{M}(\Gamma) \vdash \lambda a'.\mathsf{if}\ a'\ \mathsf{then}\ \mathcal{M}(e_1)\ \mathsf{else}\ \mathcal{M}(e_2) : \mathit{Bool} \to \mathcal{M}(t)$, we get $\mathcal{M}(\Gamma) \vdash [\![\mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]^t_\Gamma : \mathcal{M}(t)$.

(Let) Trivial since $\mathcal{M}(\mathrm{gen}(\Gamma, t_1)) = \mathrm{gen}(\mathcal{M}(\Gamma), \mathcal{M}(t_1))$.

(Seq) Trivial since $\mathcal{M}(\Gamma) \vdash \lambda\_.[\![e_2]\!]^t_\Gamma : t_1 \to \mathcal{M}(t_2) = t_1 \to M\ t_2$.

(Set) Similar to (Seq).

□

Next, we proceed to develop the technical machinery for proving the value preservation property of our monadification scheme. We first extend the monadification function $[\![\cdot]\!]_\Gamma$ to work on heap $h$ such that $[\![h]\!]_\Gamma(x) = [\![h(x)]\!]^t_\Gamma$ for $x \in \mathrm{dom}(h)$ and $t$ is $\Gamma(x)$ with quantified type variables instantiated to fresh variables. Then, as the heap is essential to the evaluation of an expression, monadic or not, we define the notions of respect and preservation by a heap for an expression with respect to monadification. First, a monadified expression $e_M$ of type $\mathcal{M}(t)$ is said to *respect* a non-monadic expression $e$ of type $t$ under a heap $h$ according to the structure of $t$ and the semantic evaluation of $e_M$ as follows.

- if $t$ is atomic, then $(h, S, \epsilon, e) \mapsto^* (h', S', \epsilon, v)$ implies $([\![h]\!]_\Gamma, S, \epsilon, e_M) \mapsto^* (h'', S'', \epsilon, \mathsf{return}\ v)$.
- if $t = \mathcal{M}(t_1) \to \mathcal{M}(t_2)$, then, for every $e_N$ of type $\mathcal{M}(t_1)$ respecting $e_1$, application $(e_M\ e_N)$ respects $(e\ e_1)$ under $h$.

Second, an expression $e$ of type $t$ is said to be *preserved* by a heap $h$ if and only if $[\![e]\!]^t_\Gamma$ respects $e$ under $h$. Since the functional case of expression preservation will be used very often, for ease of discussion, we shall adopt the following alternative yet equivalent definition.

- if $t = t_1 \to t_2$, then, for every $e_1$ of type $t_1$ preserved by $h$, application $(e\ x)$ is preserved by $h[x \mapsto e_1]$ where $x$ is a fresh variable.

Besides, a heap should be consistent with the type environment such that $\Gamma \vdash h$ if and only if $\Gamma \vdash h(x) : \Gamma(x)$ for every $x \in \mathrm{dom}(h)$. Finally, if $h(x)$ is preserved by $h[x \mapsto \bot]$ for every $x \in \mathrm{dom}(h)$, we simply say that $h$ is preserved.

The first lemma shows that the respect relation between expressions is invariant under the application of the `pos` function.

**Lemma 3** *Given a monadified expression $e_M$ of type $\mathcal{M}(t)$, if it respects another expression $e$ of type $t$ under a heap $h$, then for any type substitution $S$, $\mathtt{pos}^S_t(e_M)$ has type $\mathcal{M}(St)$ and respects the same expression $e$ of type $St$ under $h$.*

*Proof* We prove it with a similar property of `neg` simultaneously by a structural induction on $t$.

- For an expression $e_M$ of type $\mathcal{M}(St)$, if it respects another expression $e$ of type $St$ under a heap $h$, then $\mathtt{neg}^S_t(e_M)$ can be seen as if having type $\mathcal{M}(t)$ and respects the same expression $e$ of type $t$ under $h$.

The typing parts in both propositions follows directly from the proof of Lemma 2 by adding or removing the monadic type substitution $\mathcal{M}(S)$.

To prove the respect part, we assume that $e$ evaluates to some value $v$ under $h$. Consider the following cases.

1. $t$ is a type variable $a$ in the domain of $S$:
   - If $Sa$ is not a functional type, $\mathtt{pos}^S_t(e_M) = \mathtt{flatten[}Sa\mathtt{]}(e_M) = e_M \equiv \mathsf{return}\ v$, which respects $e$ of type $St$.
     Otherwise, $Sa = t_1 \to t_2 \cdots \to t_a$, $\mathtt{flatten[}Sa\mathtt{]}(e_M) = \lambda x_1 \cdots x_n.\, e_M \gg\!= \lambda y.\, y\ x_1 \cdots x_n$ where each $x_i$ has type $t_i$. Suppose each of $e^M_1 \cdots e^M_n$ respects one of $e_1 \cdots e_n$, respectively, by applying them to the result of `flatten` and replacing $e_M$ by $\mathsf{return}\ v$, $e_M \gg\!= \lambda y.\, y\ x^M_1 \cdots x^M_n$ evaluates to $v\ e^M_1 \cdots e^M_n$, which respects $v\ e_1 \cdots e_n$.

- If $Sa$ is not a functional type, $\mathtt{neg}_t^S(e_M) = e_M$, which respects $e$. Otherwise, $\mathtt{neg}_t^S(e_M) = \mathsf{return}\ e_M$. Since $St$ is a functional type, $e$ is some expression that, after some reductions, can form a lambda expression. Since $e_M$ respects it, $e_M$ cannot evaluate to the form of $\mathsf{return}\ v_M$. The only possibility is something identical to $v$. Hence $\mathsf{return}\ e_M$ evaluates to $\mathsf{return}\ v$, which shows that it respects $e$.

2. $t$ is an atomic type and $St = t$, then both $\mathtt{pos}$ and $\mathtt{neg}$ act like the identity function. The case is trivial.
3. $t = t_1 \to t_2$.
   - $\mathtt{pos}^S_{t_1 \to t_2}(e_M) = \lambda x.\ \mathtt{pos}^S_{t_2}(e_M\ \mathtt{neg}^S_{t_1}(x))$. We can apply it to an expression $e_N$ of type $\mathcal{M}(St_1)$ that respects $e_1$ of type $St_1$, and prove the resulting expression respects $e\ e_1$. By induction on $\mathtt{neg}$, $\mathtt{neg}^S_{t_1}(e_N)$ respects $e_1$. Since $e_M$, with a functional type, respects $e$, $e_M\ \mathtt{neg}^S_{t_1}(e_N)$ has type $\mathcal{M}(t_2)$ and respects $e\ e_1$. Finally by induction on $\mathtt{pos}$, $\mathtt{pos}^S_{t_2}(e_M\ \mathtt{neg}^S_{t_1}(e_N))$ has type $\mathcal{M}(St_2)$ and respects $e\ e_1$.
   - $\mathtt{neg}^S_{t_1 \to t_2}(e_M) = \lambda x.\ \mathtt{neg}^S_{t_2}(e_M\ \mathtt{pos}^S_{t_1}(x))$. Similar to the previous case, by induction on both $\mathtt{pos}$ and $\mathtt{neg}$, and the assumption that $e_M$ respects $e$, for $e_N$ of type $\mathcal{M}(t_1)$ which respects $e_1$ of type $t_1$, $\mathtt{neg}^S_{t_2}(e_M\ \mathtt{pos}^S_{t_1}(e_N))$ has type $\mathcal{M}(t_2)$ and respects $e_1\ e_N$.

The second lemma concerns the base case for proving value preservation by a heap for an expression.

**Lemma 4** *If $\Gamma \vdash x : t$, then for all preserved $h$ such that $\Gamma \vdash h$, $x$ is preserved by $h$.*

*Proof* By a case analysis on the structure of $t$:

- $t$ is atomic: Given the evaluation sequence $(h, S, \epsilon, x) \mapsto^* (h', S', \epsilon, v)$, we need to show $(\llbracket h \rrbracket_\Gamma, S, \epsilon, x) \mapsto^* (h'', S'', \epsilon, \mathsf{return}\ v)$, as $\llbracket x \rrbracket^t_\Gamma = x$.
  According to (OS:HVAL) and (OS:HEVAL), there must be some point in the given evaluation sequence at which $x$ is replaced with a heap value $v_h$:

  $$(h, S, \epsilon, x) \mapsto^* (h'', S'', \epsilon, x) \mapsto (h'', S'', \epsilon, v_h) \mapsto^* (h', S', \epsilon, v)$$

  where $h''(x) = v_h$.
  By (OS:HEVAL), the above sequence is entailed by another evaluation sequence:

  $$(h[x \mapsto \bot], S, \epsilon, h(x)) \underset{h}{\mapsto}^* (h''[x \mapsto \bot], S, \epsilon, v_h).$$

  Since $\underset{h}{\mapsto}$ is a subset of $\mapsto$, we can combine the two sequences together to get $(h, S, \epsilon, h(x)) \mapsto^* (h', S', \epsilon, v)$. Now, as $h$ is preserved, $h(x)$ is preserved, too. Hence

  $$(\llbracket h \rrbracket_\Gamma, S, \epsilon, \llbracket h(x) \rrbracket^t_\Gamma) \mapsto^* (h_M, S_M, \epsilon, \mathsf{return}\ v).$$

  In this sequence, let $(h'_M, S'_M, \epsilon, v'_h)$ be the first configuration in which the expression part is a heap value. Obviously, all of the $\mapsto$ before this configuration can be replaced by $\underset{h}{\mapsto}$. Otherwise the left hand side is of the form $e_1 \gg\!= e_2$, which is also a heap value, contradicting the assumption of this being the first such configuration.
  So, by (OS:HVAL) and (OS:HEVAL), we can compose the following evaluation sequence

  $$(\llbracket h \rrbracket_\Gamma, S, \epsilon, x) \mapsto^* (h'_M, S'_M, \epsilon, x) \mapsto (h'_M, S'_M, \epsilon, v'_h) \mapsto^* (h_M, S_M, \epsilon, \mathsf{return}\ v),$$

  which shows that $x$ is preserved by $h$.
- $t = t_1 \to t_2$: Given the evaluation sequence $(h[y \mapsto e_1], S, \epsilon, x\ y) \mapsto^* (h', S', \epsilon, v)$ for a fresh $y$ and some preserved $e_1$ with type $t_1$, we need to show $(\llbracket h[y \mapsto e_1] \rrbracket_\Gamma, S, \epsilon, \llbracket x\ y \rrbracket^{t_2}_\Gamma) \mapsto^* (h'', S'', \epsilon, \mathsf{return}\ v)$, as $\Gamma.y : t_1 \vdash h[y \mapsto e_1]$.
  By a reasoning similar to the above case, we can divide the given sequence to

  $$(h[y \mapsto e_1], S, \epsilon, x\ y) \mapsto^* (h'', S'', \epsilon, x\ y) \mapsto (h'', S'', \epsilon, v_h\ y) \mapsto^* (h''', S''', \epsilon, v_x\ y) \mapsto^* (h', S', \epsilon, v)$$

  where $h''(x) = v_h$.
  We can construct the monadified evaluation sequence as for the above case; the main difference is that $\llbracket x \rrbracket^t_\Gamma = \mathtt{pos}^{S_x}_{t_x}(x)$, where $\Gamma(x) = \forall \bar{a}.t_x$ and $S_x t_x = t_1 \to t_2$, may not be

equal to $x$. However, the monadification of $y$, $\mathtt{pos}_{t_y}^{S_y}(y)$ where $S_y t_y = t_1$, is also the result of applying the pos function.
By Lemma 3, $\mathtt{pos}_{t_x}^{S_x}(x)$ is an expression of type $S_x t_x$ which respects $v_h$, and $\mathtt{pos}_{t_y}^{S_y}(y)$ has type $S_y t_y = t_1$ and respects $h(y)$. Then $[\![x\ y]\!]_\Gamma^{t_2} = \mathtt{pos}_{t_x}^{S_x}(x)\ \mathtt{pos}_{t_y}^{S_y}(y)$ respects $v_h\ h(y)$, and will evaluate to some $v_M$ that respects $v$. Hence this case is proved.

□

The third lemma shows value preservation by a heap for a well-typed expression.

**Lemma 5** *If $\Gamma \vdash e : t$, then for all preserved $h$ such that $\Gamma \vdash h$, $e$ is preserved by $h$.*

*Proof* By structural induction on $e$.

- $e \equiv x$. By Lemma 4.
- $e \equiv \lambda x.\ e_2$. Let $t = t_1 \to t_2$. Given the evaluation sequence $(h[y \mapsto e_1], S, \epsilon, (\lambda x.\ e_2)\ y) \mapsto^* (h', S', \epsilon, v)$ for a fresh $y$ and some preserved $e_1$ with type $t_1$, we need to show

$$([\![h[y \mapsto e_1]]\!]_\Gamma, S, \epsilon, [\![(\lambda x.\ e_2)\ y]\!]_\Gamma^{t_2}) \mapsto^* (h'', S'', \epsilon, \mathsf{return}\ v).$$

  Moreover, we have

$$(h[y \mapsto e_1], S, \epsilon, (\lambda x.\ e_2)\ y) \mapsto (h[y \mapsto e_1][x \mapsto y], S, \epsilon, e_2).$$

  and

$$([\![h]\!]_\Gamma[y \mapsto [\![e_1]\!]_\Gamma^t], S, \epsilon, (\lambda x.\ [\![e_2]\!]_\Gamma^t)\ [\![y]\!]_\Gamma^{t_1}) \mapsto ([\![h]\!]_\Gamma[y \mapsto [\![e_1]\!]_\Gamma^{t_2}][x \mapsto [\![y]\!]_\Gamma^{t_1}], S, \epsilon, [\![e_2]\!]_\Gamma^{t_2})$$

  Now, as $\Gamma.x : t_1.y : t_1 \vdash e_2 : t_2$, we have $\Gamma.x : t_1.y : t_1 \vdash h[y \mapsto e_1][x \mapsto y]$, by induction on $e_2$ and $[\![h]\!]_\Gamma[y \mapsto [\![e_1]\!]_\Gamma^t][x \mapsto [\![y]\!]_\Gamma^{t_1}] = [\![h[y \mapsto e_1][x \mapsto y]]\!]_\Gamma$, we get

$$([\![h[y \mapsto e_1]]\!]_\Gamma, S, \epsilon, [\![(\lambda x.\ e_2)\ y]\!]_\Gamma^{t_2}) \mapsto ([\![h[y \mapsto e_1][x \mapsto y]]\!]_\Gamma, S, \epsilon, [\![e_2]\!]_\Gamma^{t_2}) \mapsto^* (h'', S'', \epsilon, \mathsf{return}\ v).$$

  Hence $e$ is preserved by $h$.
- $e \equiv e_1\ e_2$. Trivial.
- $e \equiv \mathsf{if}\ a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By induction on $a$:

$$[\![e]\!]_\Gamma^t = [\![a]\!]_\Gamma^{Bool} \gg\!= \lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ [\![e_1]\!]_\Gamma^t\ \mathsf{else}\ [\![e_2]\!]_\Gamma^t \mapsto^*\ \mathsf{return}\ b \gg\!= \lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ [\![e_1]\!]_\Gamma^t\ \mathsf{else}\ [\![e_2]\!]_\Gamma^t$$

  Then $e$ is preserved by $h$ following from induction on $e_1$ and $e_2$.
- $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Similar to the case of $\lambda x.\ e_2$.

□

As a consequence of Lemma 5, we obtain the main theorem of value preservation.

**Theorem 2 (Value Preservation)** *Given a pure expression $e$ and an atomic type $t$, if $\emptyset \vdash e : t$ and $e \overset{\epsilon}{\longmapsto} v$ then $[\![e]\!]_\Gamma^t \overset{\epsilon}{\longmapsto} \mathsf{return}\ v$*

*Proof* A special case of Lemma 5 by letting $\Gamma$ an empty set and $h$ an empty heap. □

## B Full implementation of the aspect monad and state accessors

Code for State Accessors

```
{-# LANGUAGE ScopedTypeVariables #-}
module SideEffects where
import CState
import qualified Data.Map as Map
import Data.Char
import Data.Maybe
```

```
type OutputBuf = [(String,String)] emptyOutputBuf = []
type M a = CState (UserVar, OutputBuf) a

putMsg :: M String -> M String -> M () putMsg a m =
  do a' <- a; m' <- m
     modify $ \(u, ms) -> (u, (a', m'):ms)

getUserVar :: M UserVar getUserVar = do (uv,_) <- get
                return uv

modifyUserVar :: (UserVar -> UserVar) -> M () modifyUserVar trans =
modify $ \(u, s) -> (trans u, s)

deserialize :: (UserVar, OutputBuf) -> Cache -> String -> String
deserialize emptyState cs str = find 0 str
  where find 0 ('<':x) = find 1 x -- DFA
        find 1 ('M':x) = find 2 x
        find 2 ('\'':x) = find 3 x
        find 3 ('M':x) = find 4 x
        find 4 (':':x) = replace 0 x -- Matched
        find 0 (x:xs)  = x : find 0 xs -- Unmatched
        find n a@(x:xs) = (take n "<M'M:") ++ find 0 a
        find n [] = take n "<M'M:"
        replace n (x:xs) | isDigit x = replace (n * 10 + digitToInt x) xs
                         | x == '|'  = getStr n (fromJust $ Map.lookup n cs) ++ find 0 xs
                         | otherwise = "<M'M:" ++ show n ++ find 0 (x:xs)
        getStr _ Nothing = "<thunk>"
        getStr n (Just (Cell False _))  = "<thunk " ++ show n ++ ">"
        getStr _ (Just c@(Cell True _)) =
             let (Left (v :: Int),_) = fromCell c (emptyState, emptyCache)
             in show v

deserializeMsgs:: (UserVar, OutputBuf) -> Cache -> [(String,String)]
-> [(String,String)] deserializeMsgs emptyState cs msgs =
   map (\(a,m) -> (a, deserialize emptyState cs m)) msgs
```

Code for Cache-extended State Monad

```
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, ScopedTypeVariables,
             ExistentialQuantification, MagicHash #-}
module CState (module Control.Monad.State,
               CState(..), Cell(..), Cache,
               evalCState, runCState, emptyCache,
               getNewCacheLoc, setCache, fromCell, add2Cache) where

import Control.Arrow
import Control.Monad.State
import qualified Data.Map as M
import GHC.Prim( unsafeCoerce# )
fromJust' s Nothing = error s
fromJust' _ (Just a) = a

data Cell = forall s a. Cell Bool (CState s a) -- Cell Ever_used Thunk
type Cache = M.Map Int (Maybe Cell)

emptyCache = M.empty

newtype CState s a = CState{ realrunCState :: (s, Cache) -> (Either a Int, (s, Cache)) }
```

```
runCState :: CState s a -> (s, Cache) -> (a, (s, Cache))--Helper function for aspect monad

runCState a (s, cs) = uncurry fromCacheEither $ realrunCState a (s, cs)

evalCState :: CState s a -> s -> (a, s)
evalCState a s = second fst $ uncurry fromCacheEither $ realrunCState a (s, emptyCache)

instance Monad (CState s) where     -- Standard State monad impl.
    return t = CState $ \(s, cs) -> (Left t, (s, cs))
    ma >>= k = CState $ \(s, cs) -> let (a, (s', cs')) = runCState ma (s, cs)
                                    in realrunCState (k a) (s', cs')

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

putCache cs' = CState $ \(s, cs) -> (Left (), (s, cs'))
getCache     = CState $ \(s, cs) -> (Left cs, (s, cs))

getNewCacheLoc :: CState s Int
getNewCacheLoc  = CState $ \(s, cs) -> let n = M.size cs
                                           cs' = M.insert n Nothing cs
                                       in (Left n, (s, cs'))

cached :: Int -> CState s a
cached n = CState $ \(s, cs) -> (Right n, (s, cs))

setCache :: Int -> CState s a -> CState s a
setCache n t = do cs <- getCache
                  case M.lookup n cs of
                    Nothing -> cached n -- for showM, shouldn't happen otherwise
                    Just Nothing -> let cs' = M.insert n (Just $ Cell False t) cs
                                    in putCache cs' >> cached n
                    Just (Just _)  -> cached n

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) =
  let (t, (s', cs'))   = fromCell (fromJust' "a" $
                                   fromJust' (show n ++ show (M.keys cs)) $
                                   M.lookup n cs) (s, cs)
      (a, (s'', cs'')) = fromCacheEither t (s', cs')
  in (a, (s'', M.insert n (Just $ Cell True ((return a) :: CState s a)) cs''))

fromCell :: Cell -> (s, Cache) -> (Either a Int, (s, Cache))
fromCell (Cell _ c) = realrunCState (unsafeCoerce# c)

add2Cache :: CState s a -> CState s (CState s a)
add2Cache v = do n <- getNewCacheLoc; return $ setCache n v
```