# What Does Aspect-Oriented Programming Mean for Functional Programmers?

Meng Wang and Bruno C. d. S. Oliveira
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{menw,bruno}@comlab.ox.ac.uk

## Abstract

*Aspect-Oriented Programming* (AOP) aims at modularising crosscutting concerns that show up in software. The success of AOP has been almost viral and nearly all areas in Software Engineering and Programming Languages have become "infected" by the AOP bug in one way or another. Interestingly the functional programming community (and, in particular, the *pure* functional programming community) seems to be resistant to the pandemic. The goal of this paper is to debate the possible causes of the functional programming community's resistance and to raise awareness and interest by showcasing the benefits that could be gained from having a functional AOP language. At the same time, we identify the main challenges and explore the possible design-space.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Multiparadigm languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

***General Terms*** Languages, Design

***Keywords*** Functional Programming, Aspect-Oriented Programming, Program Extensibility and Adaptability, Separation of Concerns

## 1. Introduction

*Aspect-Oriented Programming* (AOP) [30] aims at improving modularity through the separation of orthogonal (also know as crosscutting) concerns that show up in software. The success of AOP has been almost viral and nearly all areas in Software Engineering and Programming Languages (SE&PLs) have become "infected" by the AOP bug in one way or another. In fact, the article describing the original concept is currently rated the second most influential work in all of the SE&PLs areas [36], lagging only behind the seminal '*Gang of Four*' (GoF) design patterns book [18].

As observed by Steimann [47], a possible explanation for the success of AOP is the conception that AOP improves both the modularity and the structure of code, an ultimate appeal to programmers and software engineers. However, interestingly enough, the functional programming community (and, in particular, the *pure* functional programming community) seems to be resistant to the pandemic, with only some sporadic outbreaks happening once in a while. Notably, Aspectual Caml [39] makes a good attempt by centring its design around a classic problem, namely the expression problem. The evaluator developed there allows open extension of new language expressions and new operations at the same time. This two-dimensional extensibility is known to be hard to achieve either in object-oriented languages or functional languages. In another work, Washburn and Weirich demonstrated functional type-directed programming in AspectML [14], and showed that the extensibility of AOP is important for extensible generic programming [54]. AspectFun [52, 7] is another proposal for a functional AOP language, where static resolution of types and advice invocations are emphasised.

It appears that one of the reasons for the lack of interest in functional AOP is the perceived lack of application domains. Indeed, the majority of developments of AOP are based on an Object-Oriented (OO) environment; directly borrowing the results there, usually featuring heavy use of run-time reflection and mutable state, has a counter-effect when promoting AOP in functional programming.

Another notable reason for this lack of interest is the great scepticism that many researchers have towards AOP. As argued by Steimann [47], the (fairly well accepted) defining characteristics of AOP, namely *obliviousness* and *quantification* [16], seem to be fundamentally at odds with the stated goals of improving modularity and structure of code. When it comes to functional programming, fuelled by the first reason above, many (including the authors themselves) very often deem AOP approaches as too invasive and uncontrolled.

In particular, we identify the following main issues:

- *Obliviousness* - AOP languages typically *weave* additional code into existing programs, potentially modifying the behaviour of the original base programs without warning, which hinders reasoning.
- *Parametricity* - Most AOP languages support implicit type-directed programming using either dictionary translation or run-time type representations. This is known to break parametricity, a highly regarded feature of pure functional languages.
- *Explicit effects* - There is a close (perhaps even over-represented) relationship between the concept of AOP and the use of effects as orthogonal concerns, which are challenging in pure languages such as Haskell.

Given all these issues, is it possible to find an AOP-like model that is useful for purely functional languages, without giving up many of the cherished properties of functional programming? Per-


*WGP'09,* August 30, 2009, Edinburgh, Scotland, UK.

haps more importantly and pressingly, will the effort of finding a solution provide a payoff by bringing attractive benefits to functional languages?

The goal of this paper is to start the ball rolling by showcasing the ability of AOP to improve many idiomatic applications of functional programming that were previously thought to be difficult, while preserving the functional flavour of the solutions. The hope is that we can raise awareness and interest in the functional programming community in exploring the design space of AOP languages. We choose the language AspectFun, which will be introduced shortly, as the carrier of discussion in this paper, with the understanding that none of the examples involved is dependent on this choice.

We shall state upfront that this paper is not intended to offer any definitive solution to the issues listed above. However, we do discuss the possibilities and trade-offs that one may face and warmly invite fellow functional programming researchers to venture into this great unknown!

Despite targeting the functional community generally, this paper is particularly relevant to generic programming, which "is about making programs more adaptable by making them more general" [21], in two dimensions: (i) One of the main contributions (Section 3) of AOP in functional programming is the improved extensibility and adaptability (Section 3) (ii) Specific generic programming techniques in functional languages can directly benefit from the additional power that AOP brings in (Section 3.3).

In the sequel of the paper, we firstly give a brief introduction of AspectFun (Section 2), followed by presenting in details the two major strengths of functional AOP namely extensibility (Section 3) and separation of concerns (Section 4). We then discuss the unique characteristics of functional AOP and the design challenges come with them (Section 5) before concluding.

## 2. An Overview of AspectFun

In this section, we introduce the functional AO language, AspectFun [52, 7] , which is the carrier of our discussion. We focus on the Haskell-like syntax of the language and only present the semantics informally with examples. We also choose to omit some of the language features that are language-specific or that we consider less acceptable to functional programming, such as run-time reflection.

| | | |
|---|---|---|
| *Programs* | $\pi$ | $::= \overline{d}$ |
| *Declarations* | $d$ | $::= \mathbf{data}\ T\ \overline{\alpha} = \overline{K\ \overline{\tau}}$ |
| | | $n$@advice around$\{pc\}(arg) = e \mid$ |
| | | $f :: \tau \to \tau$ |
| | | $f\ \overline{x} = e$ |
| *Arguments* | $arg$ | $::= pat \mid pat :: \tau$ |
| *Pointcuts* | $pc$ | $::= ppc \mid pc + pc \mid pc - pc$ |
| *Primitive PCs* | $ppc$ | $::= f\ \overline{x} \mid$ any $\mid n \mid$ *within* $(f)$ |
| *Expressions* | $e$ | $::= c \mid x \mid$ proceed $\mid \lambda x.e \mid e\ e \mid K\ e$ |
| *Patterns* | $pat$ | $::= x \mid T\ \overline{pat} \mid x@pat$ |
| *Types* | $\tau$ | $::= \alpha \mid \tau \to \tau \mid T\ \overline{\tau} \mid Int \mid Bool \mid [\alpha]$ |
| *Predicates* | $p$ | $::= (f : t)$ |
| *Advised Types* | $\rho$ | $::= p.\rho \mid t$ |
| *Type Schemes* | $\sigma$ | $::= \forall\overline{\alpha}.\rho$ |

**Figure 1.** Syntax of the AspectFun Language

Figure 1 presents the language syntax. We write $\overline{o}$ as an abbreviation for a sequence of metavariables $o_1, ..., o_n$.

In AspectFun, top-level definitions include datatype and function definitions, as well as aspects. An *aspect* is an advice declaration which includes a piece of advice and its target *pointcuts*. A piece of *advice* is a function-like expression that executes when any of the functions designated at the pointcut are about to execute. The act of triggering a piece of advice during a function application is called *weaving*.

Pointcuts are denoted by $\{pc\}(arg)$, where *pc* stands for either a *primitive pointcut*, represented by *ppc*, or a *composite pointcut*. Pointcuts specify certain *join points* in the program at which advice is woven when program execution reaches there. Here, we focus on join points at function invocations. Thus a primitive pointcut, *ppc*, specifies a function $f$ or advice name $n$ for the invocations that will be advised. It is worth mentioning that a function pointcut may be *curried* (this is represented by the pattern of function application to variables), which captures executions of partially applied functions. A primitive pointcut can also be a catch-all keyword any. When used, the corresponding advice will be triggered whenever a function is invoked. The pointcut *within* $(f)$ syntactically inspects whether a call occurs within the definition of the function $f$.

Name-based primitive pointcuts can be composed to form composite pointcuts by either adding to or subtracting from the set of names that are captured. As one may well expect, $+$ is commutative and associative, whereas $-$ is neither.

The argument variable *arg* is bound to the actual argument of the named function call, if the pattern matching is successful, and it may contain a *type scope*. A type scope introduces bounded scope to the aspect. Specifically, when the function in the pointcut is polymorphic, it only matches executions with inputs of types that subsume the scope. As a result, it is safe to type check the body of the advice under the strengthened assumption that *arg* is of type $\tau$.

Advice may be executed *before*, *after*, or *around* a join point. An *around* advice is executed in place of the indicated join point, allowing the advised pointcut 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 pointcut. When there are multiple pieces of advice applicable to a join point, their execution follows the reversed textual order: the one declared or imported later gets executed first. As both *before* advice and *after* advice can be straightforwardly simulated by *around* advice that uses proceed, AspectFun only considers *around* advice.

AspectFun is a polymorphic and statically typed language, with full type inference. It introduces the concept of *advised types* that are augmented with type predicates of the form $(f : t)$. Advised types are inspired by the type system of Haskell's type classes, and are used to capture the need of advice weaving based on type context. As a result, AspectFun is able to statically resolve type scopes on pointcut and weave aspects into the base program. In contrast to from type classes, the type predicates are only for the purpose of semantics-preserving weaving and are not reflected in the base program, a direct result of the obliviousness property. Like the host language Haskell that AspectFun is compiled into, AspectFun is a lazy language.

As a first example, consider the function $quicksort :: [Int] \to [Int]$. As the name suggests, quicksort is one of the fastest sorting algorithms in practice. Perhaps ironically, many implementations of quicksort perform badly when the input list is (nearly) sorted. Therefore, in some applications with predominantly nearly sorted lists, we may want to supplement the standard definition with a case that deals with already sorted inputs. We can achieve this using the following advice.

$sort$@advice around$\{quicksort\}(x) =$
  **if** *sorted* $x$ **then** $x$ **else** proceed $x$

Advice *sort* checks 'sortedness' of inputs before execution of *quicksort* (specified by the pointcut here) and resumes the execution if the check fails. Every call to *quicksort* is intercepted,

including recursive ones, which extends its applicability to nearly sorted lists. Note that the use of proceed is important here, since a naive call to *quicksort* will trigger the same advice again and result in non-termination.

## 3. Extensibility and Adaptability

One distinctive strength that AOP may bring into functional programming is modular overriding of existing definitions. In this section we demonstrate how it can help to improve the adaptability and extensibility of functional programs in several different application domains.

### 3.1 Aspects Yield Open Functions

In most functional languages, functions are usually defined by case analysis on the inputs. Once declared, there is no way of introducing new cases without modifying the original definition. Programming language extensions for *open functions* [38, 43] have been proposed in the past to allow such extensions.

In the OO programming paradigm, this problem manifests itself as the TEMPLATE METHOD [18] design pattern, which typically involves defining an abstract class with implementations for most of its methods but leaving some methods abstract. The intention is to defer some application specific steps to the subclasses. Adapting it to the functional setting, instead of defining an abstract class, we define a function by cases while leaving the application specific ones out. Despite being difficult with traditional functional programming, AOP yields open functions naturally. With around advice we can intercept the execution of partially defined functions and direct the control to new cases defined in the advice body.

Consider the example of implementing an evaluator for a small arithmetic language.

```
data Val =  N Int
         |  B Bool
         |  Wrong
data Term = Add Term Term
          |  Minus Term Term
          |  IsZ Term
          |  If Term Term Term
```

The evaluation strategy of most constructs in this language is standard, with the exception of *If*. We can choose either to evaluate eagerly both arms or only evaluate the one that will be picked by the boolean condition. A conventional implementation usually has to make the choice upfront and commit to it. With open functions, we can leave the option open and fill in the context specific missing cases for the specific application domain later.

```
add :: Val → Val → M Val
add (N i) (N j) = return (N (i+j))

substr :: Val → Val → M Val
substr (N i) (N j) = return (N (i−j))

eval :: Term → M Val
eval (Add n1 n2)   = eval n1 >>= (λa →
                     eval n2 >>= (λb →
                     add a b))
eval (Minus n1 n2) = eval n1 >>= (λa →
                     eval n2 >>= (λb →
                     substr a b))
eval (IsZ n)       = eval n >>= (λN a →
                     return (B (a ≡ 0)))
eval _             = error "Unknown Expression!"
```

We leave out the case for *If* in the definition of the open function above. Later, in different modules, we may easily switch between the two evaluation strategies by plugging in one of the aspects.

```
module A where
both@advice around{eval}(If c e1 e2) =
  eval c  >>= (λb  →
  eval e1 >>= (λv1 →
  eval e2 >>= (λv2 →
  applyIf b v1 v2)))
applyIf :: Val → Val → Val → M Val
applyIf (B True) v1 v2  = return v1
applyIf (B False) v1 v2 = return v2
applyIf _               = return Wrong

module B where
one@advice around{eval}(If c e1 e2) =
  eval c >>= (λb → applyIf b e1 e2)
applyIf :: Val → Term → Term → M Val
applyIf (B True) e1 e2  = eval e1
applyIf (B False) e1 e2 = eval e2
applyIf _               = return Wrong
```

Without the aspects, we can still achieve a similar behaviour by parameterising *eval* with the functionality that would handle the case for application. However, this is obviously heavy-weight and involves upfront preparation to account for the additional flexibility.

In a typical OO setting, template methods are usually implemented as abstract; and it is statically enforced that only objects with instantiated template methods can be constructed. This facility does not exist in functional languages: as usual programmers are responsible for making pattern matching exhaustive.

### 3.2 Type-directed Programming

In Haskell, an idiomatic way of achieving modular extensibility is through type classes. Consider the evaluation of the small arithmetic language we had before (we leave out the orthogonal error handling for a simplified presentation). All syntactic constructs must be lifted to the type level so that the different cases can be defined as instances of a class.

```
data Add a b = Add a b
data Minus a b = Minus a b
data IsZ a = IsZ a
data If a b c = If a b c
data Val = Num Int
         | B Bool
```

The function *eval* now can be defined as an overloaded class method.

```
class Eval a where
  eval :: a → Val
instance (Eval a, Eval b) ⇒ Eval (Add a b) where
  eval (Add n1 n2) = let Num v1 = eval n1
                         Num v2 = eval n2
                     in Num (v1 + v2)
instance (Eval a) ⇒ Eval (IsZ a) where
  eval (IsZ n1) = let Num v1 = eval n1
                  in if v1 ≡ 0 then B True else B False
...
```

This solution is much more involved than the one with open functions: a new operation on the language necessarily introduces a new class; and the types of terms become very complicated, for example

*Add* (*Minus* 2 3) 2 :: *Add* (*Minus Int Int*) *Int*. Given terms of different types, it becomes very tricky to manipulate them, for example, putting them into an environment.

This kind of type-directed programming is also supported by most AOP languages. For example, in AspectFun we can start with a default case for the evaluation function and gradually enrich the definition by introducing advice for different types of argument, overriding the default behaviour.

```
eval :: a → Val
eval = error "Unknown expression!"

add@advice around{eval}(Add e1 e2 :: Add a b) =
  let Num v1 = eval n1
      Num v2 = eval n2
  in Num (v1 + v2)
...
```

Given that AspectFun performs static weaving in a similar manner to the dictionary translation of type classes, the two approaches have similar run-time performance. The advantage of the AOP solution is the elimination of the complex class hierarchy, which allows new operations to be more easily defined. This flexibility costs us some static safety: without a type class context, there is no way to guarantee that the recursive calls to *eval*, probably on different input types, are actually defined.

Another benefit of type classes is the explicit qualification of types of overloaded functions, which distinguish them from parametric polymorphism. As a result, the properties of parametricity are preserved and more precise typing can be achieved. We will discuss this in more detail in Section 5.

Other than typing, the more operational difference between aspects and class instances is the possibility of multiple triggering, as we will see next.

### 3.3 Extensible Generic Programming

*Generic programming* [27, 24, 31, 32, 23, 26] is another functional idiom where the extensibility of aspects plays a central role.

Looking back to the *eval* function above, type-directed programming allows us to specify a case for every datatype. This is fine-grained, but not very general: structurally similar but nominally different types have unrelated implementations, which results in large amount of "boiler-plate" code. Consider function *strings* that extracts all the strings from a structure. With a *nominal* approach, we are required to define a case for every datatype, which are mostly non-productive inductive traversals.

In contrast, generic programming exploits structure information of datatypes, and dispatches based on structure representations. Given that overloading is necessary for supporting generic programming [26], type classes have been a popular choice for realising the idea in Haskell [31, 32, 23]. For example, the *Spine* representation of datatypes is defined as follows:

```
data Spine a = Con (Constr a)
             | ∀b.ToSpine b ⇒ App (Spine (b → a)) b

data Constr a = Descr a
```

If a constructor does not take any argument, it is encoded by *Con* together with information about the constructor. Otherwise, a constructor taking arguments is encoded by applying *App* to the representation of the constructor and to its arguments. The function *toSpine*, which converts a datatype to its spine representation, is type-directed, and defined as a type class method below.

```
class ToSpine a where
  toSpine :: a → Spine a

instance ToSpine Int where
  toSpine x = Con (Descr 0)

instance ToSpine Char where
  toSpine x = Con (Descr 'a')

instance ToSpine a ⇒ ToSpine [a] where
  toSpine [] = Con (Descr [])
  toSpine (x : xs) = (App (App (Con (Descr (:))) x) xs)
```

All datatypes are now mapped to a single one, *Spine*. We can easily define functions that work on this representation.

```
strings :: a → String
strings x = strings_ (toSpine x)

strings_ :: Spine a → [String]
strings_ (Con c) = []
strings_ (App f x) = strings_f ++ strings x
```

This version of *strings* behaves uniformly on all datatypes by traversing the structures, but not producing any strings! What we need is a small exception to this generic behaviour that returns a string when the input is a string. Since this is a type-directed operation, we could try to use type classes again.

```
class Strings a where
  strings :: ToSpine a ⇒ a → [String]

instance Strings String where
  strings x = [x]

instance Strings a where
  strings x = strings_ (toSpine x)
```

The two instances above are overlapping. The intention is to match on the more specific one when the input is a string and the general one otherwise. However, this behaviour is not expressible statically: there is no way of knowing the type of the existential component of constructor *App* during compilation, which makes it impossible to decide the call to *strings* inside *strings_*'s body.

Since AspectFun performs aspect weaving statically, it also suffers from the difficulty above. Thus, directly encoding with type-scoped advice as follows does not work.

```
n@advice around{strings}(x :: String) = [x]
```

Because the exact type is not available statically, the resolution of overloading can only be made at run-time by dynamic type casting [31] or special dictionaries [32]. The use of dynamic type casting with type classes is counterproductive here since it precludes extensibility by forcing all relevant cases to be defined in a single instance. The use of special dictionaries could reuse the existing type class mechanism. But a working solution is necessarily complicated and involves not-so-common language extensions.

We believe the fundamental difficulty here is that type classes are designed for overloading, and provide complete functionality for each type case. On the other hand, what is needed in extensible generic programming is the ability to refine or supplement the generic behaviour and AOP seems more suitable for this purpose.

Instead of overloading the generic function, we can simply define the generic behaviour and incrementally include special cases by introducing individual aspects. For example, the *String* case can be defined as the following.

```
n@advice around{strings}(x) =
  case cast x :: Maybe String of Just s → [x]
                                 _      → proceed x
```

This advice intercepts all executions of *strings*. When the input is dynamically verified to be a string, we return that string in the result; otherwise, control is passed back to *strings* (or to some other intercepting advice). As we can see, the *proceed* mechanism plays a central role here: if it is replaced by a call to *strings*, advice *n* will be triggered again, which results in an infinite loop.

In addition to the ability to *proceed*, AOP offers extensibility. Suppose we later implemented a datatype of ASCII-coded of characters and would like to consider a list of ASCII as a string as well. Function *strings* can be easily extended with another special case using advice.

$$
\begin{array}{l}
n1@advice\ around\{strings\}(x) = \\
\quad \textbf{case}\ cast\ x :: Maybe\ [Ascii]\ \textbf{of}\ Just\ s \to [x] \\
\qquad\qquad\qquad\qquad\qquad\quad \_ \quad \to proceed\ x
\end{array}
$$

### 3.4 Inheritance and Overriding

Recursion is the predominant technique in functional programming. For most problems, the recursive pattern is highly structural. This regularity has been well understood and exploited by many to reduce the task of programming into filling in a few blanks. In functional languages, the idiomatic way of achieving such reuse is through higher-order functions that compose behaviours. However, if one chooses to define structural recursion explicitly, reusing the definitions reduces to merely "cut-and-paste". This problem has been discussed in [33], which compared this kind of code reuse to the typical reuse in object-oriented languages and the visitor pattern. Let's consider an example on trees.

$$
\begin{array}{l}
\textbf{data}\ Tree\ a = Leaf\ a \\
\qquad\qquad\quad \mid\ Branch\ (Tree\ a)\ (Tree\ a)
\end{array}
$$

$$
\begin{array}{l}
sum\ (Leaf\ a) = a \\
sum\ (Branch\ t1\ t2) = sum\ t1 + sum\ t2
\end{array}
$$

Function *sum* sums all the leaf values of a tree. Now suppose that we want to define a slight variant of *sum* by summing only the even leaf values. Since this new function is very similar to *sum*, we may consider reusing the original definition. A possible attempt is the following:

$$
\begin{array}{l}
sumEven\ (Leaf\ a) = \textbf{if}\ isEven\ a\ \textbf{then}\ a\ \textbf{else}\ 0 \\
sumEven\ t = sum\ t
\end{array}
$$

Function *sumEven* works as desired for leaves but fails for more complicated trees, since the recursive calls are still bound to *sum*. There is no easy solution to this problem in existing functional languages such as Haskell.

With the introduction of aspects, reuse of *sum* becomes straightforward.

$$
even@advice\ around\{sum\}(Leaf\ x) = \textbf{if}\ isEven\ x\ \textbf{then}\ x\ \textbf{else}\ 0
$$

Advice *even* intercepts every execution of *sum* with leaf input and checks whether the value is even. Additional advice can be subsequently introduced to further change the original function. For example,

$$
positive@advice\ around\{sum\}(Leaf\ x) = \textbf{if}\ x > 0\ \textbf{then}\ x\ \textbf{else}\ 0
$$

the advice *positive* ensures that only positive leaves are added up. Now, programmers can liberally choose different combinations of sum behaviours by bringing different aspects into scope either statically or dynamically, depending on the language's weaving strategy.

The adaptations of *sum* above are "in-place": we lose the original definition of *sum* in the same scope. A solution to this problem is to have another function pointing to *sum*, such as

$$
sumEven = sum
$$

and then we can specify in the pointcuts that only invocations of *sum* before *sumEven* returns will be advised. Through standard in AOP, this kind of *control-flow based* pointcut requires run-time reflection, which does not fit well with (pure) functional programming. We choose to leave this feature out in this paper.

Many definitions share similar cases. We could try to extract the common elements into aspects. Let's consider optimising recursive functions by using accumulator parameters. It is well-known [3] that by using an extra argument (the accumulator parameter) to a function, we can sometimes improve the run-time performance. For example, consider the reverse function on lists.

$$
\begin{array}{l}
reverse :: [a] \to [a] \\
reverse\ [\,] \quad\ = [\,] \\
reverse\ (x : xs) = (reverse\ xs) +\!\!+ x
\end{array}
$$

This straightforward recursive definition has a quadratic run-time performance, due to the expensive operation $+\!\!+$ that is invoked on every step of the recursion.

$$
\begin{array}{l}
reverse' :: [a] \to [a] \to [a] \\
reverse'\ [\,]\ acc \quad\ = acc \\
reverse'\ (x : xs)\ acc = reverse'\ xs\ (x : acc)
\end{array}
$$

By using an accumulator parameter, we can replace the $+\!\!+$ operation by constant time list construction, which achieves linear time performance.

Another example of using accumulator parameter is flattening a binary tree. The straightforward definition has asymptotic performance of $O(n^2)$ whereas the accumulator version

$$
\begin{array}{l}
flatten' :: Btree\ a \to [a] \to [a] \\
flatten'\ Empty\ acc \quad\ = acc \\
flatten'\ (Leaf\ x)\ acc \quad\ = x : acc \\
flatten'\ (Fork\ xt\ yt)\ acc = flatten'\ xt\ (flatten'\ yt\ xs)
\end{array}
$$

has linear time performance.

A very similar story applies for the flattening of rose trees, for which we only show the accumulator definition here.

$$
\begin{array}{l}
flatternRose :: Rose\ a \to [a] \\
flatternRose\ xt = dfcat\ [xt]\ [\,] \\
dfcat\ [\,]\ acc \qquad\qquad\quad = acc \\
dfcat\ (Node\ x\ xts : yts)\ xs = x : dfcat\ xts\ (dfcat\ yts\ xs)
\end{array}
$$

The list goes on to the $showsPrec :: Int \to a \to String \to String$ function in Haskell's class *Show*, where the third parameter is an accumulator parameter.

All the definitions in accumulator style have a common pattern: the accumulator parameter is returned when the input is empty. We could try to capture this common base case with an aspect.

$$
\begin{array}{l}
base@advice\ around\{reverse' + flatten' + \\
\qquad\qquad\qquad\qquad dfcat + showsPrec\ i\}(x) = \\
\quad \textbf{if}\ isEmpty\ x\ \textbf{then}\ id\ \textbf{else}\ proceed\ x
\end{array}
$$

The testing function *isEmpty* has to be a type-directed function that works on multiple types. The way to define this function has been discussed in Section 3.2.

There is another correctness crosscutting concern on accumulator style definitions: the accumulator parameter must be empty when initially called.

$$
\begin{array}{l}
empty@advice\ around \\
\quad \{(reverse'\ x - within\ (reverse')) + \\
\quad\ (flattern'\ x - within\ (flattern')) + \\
\quad\ (dfcat\ x - within\ (dfcat)) + \\
\quad\ (showsPrec\ i\ x - within\ (showsPrec))\}(acc) =
\end{array}
$$

```
  if isEmpty acc then proceed acc
  else error "NonEmpty accm"
```

The negative *within* pointcuts ensure that only the initial calls of the functions are checked. Unfortunately in AspectFun, there is no smarter way of defining lengthy pointcuts like the above, nor any way to extend them easily. However, languages such as AspectML or Aspectual Caml, where pointcuts are defined separately from advices, offers better support for pointcut 'programming'.

This last aspect *empty* encodes an orthogonal concern other than the core functionality of the respective programs, by offering an additional correctness check. This modular separation of concerns is another major feature that AOP may bring to functional programming, as discussed next.

## 4. Separation of Concerns

In this section we will see some classic applications of AOP to the problem of separation of concerns in functional programming.

### 4.1 Contract Enforcement

Very often programmers want to insert assertions at various points in the program to check the validity of values. As a result, such checks are scattered and tangled with other code, which hinders comprehension and complicates maintenance. To solve this problem, a concept of *contract* (a set of pre- and post-conditions) was developed, the value of which in building robust systems has long been recognised [42]. Most contract systems [4, 17, 25] introduce a separate contract specification language into the host language, and compilers are extended to allow interpretation of the specifications either statically or dynamically so that the target program can be checked.

In this section, we show how aspects can be used conveniently and effectively to specify contracts. The modularity of the approach makes it straightforward to introduce contracts and remove them when run-time performance is more critical.

Consider the popular RGB colour model for colour rendering where three parameters representing red, green, blue are added together in various ways to produce a wide spectrum of colours. Suppose we encode the RGB colour in 24 bits per pixel, using three 8-bit unsigned integers (0 through 255) representing the intensities of red, green and blue. Any number beyond this interval is considered an error and cannot be displayed. The following function takes in a triple specifying a colour and displays it. We omit the actual definition of the function.

$$display :: (Int, Int, Int) \to Colour$$

As discussed, *display* can only handle inputs that fall into the interval of [0..255]. We can enforce this precondition by advice.

$$inrange\ x = x \geqslant 0 \wedge x \leqslant 255$$

```
rgb@around advice{display}((r,g,b)) =
  if (inrange r) ∧ (inrange g) ∧ (inrange b)
  then proceed (r,g,b)
  else error "Non-displayable Colour"
```

The advice *rgb* makes use of an auxiliary function *inrange* and only proceeds when the input is valid.

The RGB model can be refined by introducing additional parameters to more accurately specify colours. One such system is HSV, which stands for hue, saturation and value. The HSV values are derived from the RGB values. For example, the saturation value is computed by the following formula:

$$s = \frac{max(r,g,b) - min(r,g,b)}{max(r,g,b)} \qquad (1)$$

```
data Expr where
  Lit      :: Int → Expr
  Var      :: String → Expr
  Plus     :: Expr → Expr → Expr
  Minus    :: Expr → Expr → Expr
  Assign   :: Expr → Expr → Expr
  Sequence :: [Expr] → Expr
  While    :: Expr → Expr → Expr
type Env = [(String, Int)]
type EvalM a = WriterT String (State Env) a
```

**Figure 2.** Datatype and environment type for expressions.

This operation involves division, which gives rise to the divide-by-zero exception. Again, we can use an advice to rule it out.

```
divzero@around advice{div x}(y) =
  if y ≡ 0 then error "Division by Zero"
  else proceed y
```

Since function $div :: Float \to Float \to Float$ takes in two inputs in curried form and has the second as the divisor, the above advice has a *curried pointcut*, which intercepts partial application of function *div* and captures its second input. This pointcut even matches when the partially applied function is not immediately applied.

Postconditions can be specified by advice too. Consider the square root function $sqrt :: Float \to Float$. Given a non-negative input, the output must be non-negative too.

```
sqr@around advice{sqrt}(x) =
  if x ⩾ 0 then let y = proceed x
                in if y ⩾ 0 then y
                   else error "Wrong result for sqrt"
       else error "Invalid sqrt input"
```

Advice *sqr* checks both the precondition and postcondition of *sqrt*. Postconditions can be *dependent* on the input values. For example,

```
sqr1@around advice{sqrt}(x) =
  let y = proceed x
  in if abs (x − y ∗ y) < 0.01 then y
     else error "Wrong result for sqrt"
```

This advice, in addition to *sqr*, checks the accuracy of the result by comparing the square of it with the input.

Dynamic contract checking necessarily incurs run-time overhead. If the functions above are part of a colour representation in the palette of painting software, it is vitally important to display the correct colour perceived by the user.

On the other hand, in some applications, preventing colour distortion caused by having an invalid representation is less important. For example, if we are rendering the display of a LCD panel, having the colour of one pixel out of millions wrong is very unlikely to be observable. In this case, speed becomes more crucial and we may choose to ignore the exceptions; contracts in the form of advice can be easily removed from the system since it is not tangled with the core functionality. Note that this removal does not make the programs less correct; it only eliminates the dynamic contract checking.

### 4.2 Monadic Interpreters

When it comes to orthogonal concerns in the form of side-effects, the conventional approach with pure functional languages is through monads. In Figure 2 we present a datatype representing a simple imperative language that can be used to compute numeric

```
eval :: Expr → EvalM Int
eval exp = case exp of
  Lit x              → return x
  Var s              → do e ← get
                          case lookup s e of
                            Just x → return x
                            _      → error msg
  Plus l r           → do x ← eval l
                          y ← eval r
                          return (x + y)
  Minus l r          → do x ← eval l
                          y ← eval r
                          return (x − y)
  Assign (Var x) r   → do e ← get
                          y ← eval r
                          put ((x, y) : e)
                          return y
  Sequence [ ]       → return 0
  Sequence [x]       → eval x
  Sequence (x : xs)  → eval x ≫ eval (Sequence xs)
  While c b          → do x ← eval c
                          if (x ≡ 0) then return 0
                            else (eval b ≫ eval exp)
  where msg = "Variable not found!"
```

**Figure 3.** A classic monadic evaluator.

expressions—this example is based on an interpreter presented in [11], which in turn is a Haskell translation of an interpreter implemented in ML [33]. Integer literals and variables can be built using, respectively, the *Lit* and *Var* constructors. Simple primitive operations for addition and subtraction are available through the *Plus* and *Minus* constructors. Mutable assignments to variables can be defined using *Assign* and sequential composition and while loops can be constructed with *Sequence* and *While*. A simple environment type for expressions is given by *Env*. We also define a monad *EvalM*, which is the combination of a writer and a state monad, for use with the evaluator.

In Figure 3 we show a classic monadic evaluator for the expressions presented in Figure 2. The state monad transformer is used to pass the environment around and it is also used in the assignment clause to update the value of the variable being assigned. The evaluator is quite standard. Evaluating integer literals returns the integer denoted by the literal. The evaluation of variables looks up the variable from the environment and returns its value; if no value is found, an error is raised. The primitive arithmetic operations are evaluated in a similar way: both arguments of the operations are evaluated and the corresponding arithmetic operations are applied to the result of the evaluations. For assignments we need to evaluate the expression being assigned and update the variable with the new value. Sequential composition of an empty list of expressions returns 0, whereas the sequential composition of a list with a single expression returns the value of that expression. For a non-empty list of expressions we evaluate the expression in the head and then the expressions in the tail. Finally, while loops are evaluated similarly to the *C* programming language, with integers playing the role of booleans: we first evaluate the condition; if that condition is 0 we stop and return 0, otherwise we evaluate the body of the while loop and evaluate the original while loop expression again.

Suppose that, for debugging reasons, we wanted to watch the assignments of some variable and trace the execution of the while loops. Typically, in order to achieve this with the monadic evaluator

```
weval@advice around{eval}(exp@Assign (Var x) r) =
  if x ≡ "y" then
    do n ← proceed exp
       tell (x ++ " = " ++ show n ++ "\n")
       return n
  else proceed exp
```

**Figure 4.** The watching variables aspect.

```
teval@advice around{eval}(exp@While c b) =
  do n ← eval c
     if (n ≡ 0) then (tell "done\n" ≫ return 0)
       else (tell "repeating\n" ≫ eval b ≫ eval exp)
```

**Figure 5.** The tracing loops aspect.

presented in Figure 3, we would need to directly change the original program and adapt it with the extra functionality. Moreover, even if we use approaches such as the mixin-based solution suggested in [11], we would still need to do a little bit of planning for later extensions by writing the base evaluator in a slightly different way.

***Modular Aspects of Interpreters*** In AspectFun, there is no need to touch the base program or plan ahead for possible extensions: we can just write modular aspects that are woven into the base program. In Figure 4 we show how we could modularly define a watching aspect for assignments. This aspect watches a designated variable `"y"`. For all cases other than assignment we *inherit* the functionality by calling proceed. For the *Assign* constructor we do something different by *overriding* the functionality provided by the base interpreter. Since we want to watch what happens in the assignments of `"y"` we have to compare `"y"` with the variable being assigned and, if they represent the same variable, call proceed to execute the assignment code, as well as adding extra watching code using the writer monad. If `"y"` does not match the variable being assigned, then the guard will fail and the execution will fall through the default case, just executing the standard assignment code provided by proceed.

In Figure 5 we show how we could modularly define the code for the tracing while loops using aspects. The idea is that, for the *While* constructor, we make a recursive call directly, which has the effect of *completely overriding* all the code for handling while loops. Consequently, we need to essentially repeat the code that we have in *eval*, but this time decorated by some tracing code using the writer monad.

As we have seen the modularity benefits of using aspects to capture the tracing and watching variables aspects are significant. In order to add a new orthogonal piece of the functionality we do not need to alter the original program. Instead, we can simply create new aspects that decorate the base program and override just the functionality that needs to be changed.

## 5. Discussion

In this section we briefly compare functional AOP with the more established notion of AOP in OO, and discuss the issues that arise in the design of purely functional AOP languages together with possible solutions for them.

### 5.1 Object-Oriented AOP vs Functional AOP

AOP was born as a programming paradigm that improves separation of concerns by offering another dimension of grouping other than the underlying support of encapsulation of a host language [30]. This idea quickly took a strong hold in OO programming, where encapsulation is predominant, and as a result, where

the problem of code dangling and scattering is most severe. Notably the success of AspectJ [29, 2], an AOP language based on Java, is well recognised. Given the complicated control structure of Java, the pointcut language in AspectJ is very rich. A typical aspect in AspectJ *crosscuts* several classes, very often through the use of wildcards in certain fields of the pointcuts. Consequently, despite being oblivious, it is obvious that base programs with better naming disciplines make the aspect development easier.

In most functional AOP languages, the pointcut model based on function invocation is much simpler. In addition to the well known applications of separation of concerns, such as tracing or contract checking, an AOP extension in a functional setting is able to model OO style inheritance and overriding. This is novel, but not surprising, since one of the main strengths of AOP lies in facilitating extensibility and adaptability, which, however, has been shadowed by the powerful inheritance infrastructure in OO.

### 5.2 Parametricity

Most (if not all) functional AOP languages, including AspectML, Aspectual Caml and AspectFun, support type-directed programming. In those languages, it is possible to define a function *eval*, like the one presented in Section 3.2, with the following type:

$$eval :: \forall a.\ a \to Val$$

In languages like Haskell, properties arising from parametricity abound [50], but type-directed programming in the style above breaks these properties. In this example, because of parametricity, we would expect that *eval* would not be able to make any use of its first argument, since nothing is known about the information contained in values of the type $a$. Consequently, in a language where parametricity is preserved, the *eval* function would necessarily need to return a constant. However, with implicit type-directed programming, we can perform a case analysis on the type and discover information about $a$, which allows us to return something other than a constant. This breaks the parametricity properties that we would normally expect from a function of this type. Since parametricity is highly valued in functional programming, it is important to consider possible design alternatives that can be used to restore (or at least partially restore) parametricity. A few alternative designs are discussed next:

***Type-safe cast*** One possible alternative design is to allow type-safe casts [56, 31] as in, for example, the current versions of the Glasgow Haskell Compiler (GHC) [22]. In this design, we would be able to do a (limited) form of type-directed programming but only through the use of a *type-safe cast* function:

$$cast :: (Typeable\ b, Typeable\ a) \Rightarrow a \to Maybe\ b$$

The advantage of this design is that parametricity properties are preserved, since any functions involving *cast* will necessarily give rise to *Typeable* constraints. For example, if we wanted to define an evaluation function using type-directed programming, we would need to write

$$eval :: Typeable\ a \Rightarrow a \to Val$$

In this design a function without any *Typeable* constraints has the usual parametricity properties. However, a disadvantage of this approach is that it typically relies on some built-in compiler machinery (for example, in GHC, we need to rely on the compiler generating the type-class instances for *Typeable*).

***Type Representations*** Another alternative way of doing type-directed programming is through the use of *type representations* [44], which are widely used in lightweight forms of *datatype-generic programming* [8, 23]. In this design the function *eval* would have a type like:

$$eval :: Rep\ a \to a \to Val$$

or, alternatively:

$$eval :: Rep\ a \Rightarrow a \to Val$$

In either case it is possible to discriminate the possible type representations of $a$. Like with the previous solution, we can tell if a function uses type-directed programming because of the *Rep* arguments (or constraints) in the type. The main advantage of this solution when compared to the *Typeable* approach is that it is less reliant on type-classes and some "magic" introduced by the compiler. Unfortunately, a typical disadvantage of this approach is that the type representations are *closed* (that is, it is hard to add new type representations without modifying the original *Rep* type). Some of the latest work in this area has been focused on lifting this limitation [45, 57], giving us some hope that this alternative may be useful in practice.

***Type Labelling*** The restoration of parametricity in the presence of run-time analysis has been studied before [53]. The basic idea of the proposal is simple: distinguishing parametric type variables that are analysable by labels. Applying to the case here, we could mark the types of all overloaded functions, for example *eval* in Section 3.2, so that they are not confused with genuine parametric polymorphic functions. Despite the loss of some obliviousness, we believe in practise this should cause little disturbance because programs only have to make a (usually clear) choice on whether a polymorphic function is intended to be extended with additional type cases.

### 5.3 Reasoning with Aspects

Equational reasoning is a distinctive feature of pure functional languages. Given the absence of (implicit) effects or, more generally, the existence of *referential transparency*, a carefully designed language can support local reasoning about program behaviours and allow replacing programs with equivalent ones without any observable differences in behaviour. The former is clearly beneficial for program comprehension and the latter is important for program optimisation and parallisation. As a simple example, consider the map fusion law:

$$map\ f \circ map\ g = map\ (f \circ g)$$

The left-hand side of the equation above can be safely replaced by the more efficient right-hand side by a compiler regardless of the context it is used. This nice property is directly threatened by the introduction of aspects, especially oblivious aspects. Consider the *sum* example in Section 3.4; an advice such as *even* may override its behaviour. This gives us adaptability and reusability, but at the cost of sound reasoning: the semantics of other programs that make use of *sum* are changed silently. We suggest some possible designs that can help restoring some (and perhaps all) forms of equational reasoning next:

***Noninterference*** To control the possible 'damage' that comes with the use of aspects, a whole theory of non-interference, appearing under different names, has been developed: *observation* [9], *orthogonal, independent and observation* [46], *almost spectative* [28], *strongly independent* [15] and *harmless advice* [13], are just some of the most relevant works in this area. The common goal is to classify aspects and base programs with respect to certain interference properties that they may have. For example, with harmless advice we can ensure that advice will only perform effects and will not change the core functionality of the base program. The tracing aspect in Section 4.2 is an example of harmless advice. Unfortunately, this notion of noninterference is too weak for functional reasoning: two expressions are not equivalent even if they only differ in the effects performed. As a matter of fact in Haskell

all effects are cleanly abstracted into monads (or other mechanisms such as *applicative functors* [41] or *comonads* [49]) and two programs that only differ in their effects will have different types and cannot be substituted for each other. It is perhaps more meaningful to argue whether certain aspects will be capable of breaking existing invariants of the base program instead. For instance, we could consider the *sort* aspect being harmless since it does not change the invariant that any output will be sorted. On the other hand, advice of this kind may change the run-time performance or even the time complexity of the original program. This makes the already complex time complexity analysis of lazy languages even harder.

***Modular Aspects*** Instead of insisting on being completely oblivious, proposals have been made to give the base program some control over the way it can be advised. Commonly hidden within certain module boundaries, base programs may choose to explicitly export join points that are receptive to advising [37, 1]. Applied to the level of functions, it makes sense to syntactically mark functions that are being advised, in the same spirit as the treatment of monads. It is obvious that obliviousness will be affected. However, we believe that this is a reasonable trade-off for proper reasoning. With mixins [5] (which provide a simple model for inheritance) it is possible to program in a style very similar to AOP, but with less obliviousness [11]. In this style a function that is meant to be advised has to have a suitable *Mixin* type. For example, rather than defining an advisable sorting function with a type $sort_1 :: Ord\ a \Rightarrow [a] \rightarrow [a]$ (such as the function discussed in Section 2), we would need to write $sort_2 :: Ord\ a \Rightarrow Mixin\ ([a] \rightarrow [a])$. The advantage of being less oblivious is that it is clear from the types of functions that we can expect $sort_2$ to be more flexible and general than $sort_1$; and $sort_1$ to be easier to reason about (since it is less parametrized).

### 5.4 The Challenge of Effects

A primary goal of AOP is to capture orthogonal concerns that show up in software. For example, we may be interested in capturing *tracing* or *memoisation* concerns separately from the code that implements the core functionality of a program. By their own nature, orthogonal concerns tend to involve side-effects. For instance, in the following toy example,

```
trace@advice around{h}(arg) =
  proceed arg;
  println "exiting from h"
h x = x
```

we can see how to separate tracing from the core functionality of a function $h$. The function $h$ (in this case just the identity function) is advised by *trace*, which prints a message to the console every time that an execution of $h$ finishes.

While orthogonal concerns are one of the primary motivations in traditional renderings of AOP, they pose a fundamental challenge in pure functional languages because effects appear explicitly in the types. In the example above we would expect $h$ to have a type $a \rightarrow a$ but, since the advice executes an *IO* action, $h$ should really have a type $a \rightarrow IO\ a$. In a language with implicit effects (such as AspectML or Aspectual Caml) this would not be a problem because this kind of side-effect would be transparent and would not change the original type. We can think of the following solutions to address the challenge of effects in pure functional languages:

***Anticipate possible effects*** One possible way out of the problem would be to anticipate all possible effects that may occur. This was the option we took in the example presented in Section 4.2. For example, instead of declaring $h$ as above, we could have the following definition:

```
h :: a → IO a
h x = return x
```

The idea here is just that we anticipate the use of *IO* by any possible advice. In this way, advice could freely introduce *IO* computations and no problem would arise in the first place. One problem with this solution is the loss of some obliviousness, since now the program $h$ needs to make some preparation for advice. A more fundamental problem is that this still does not account for the introduction of other kinds of effects. If, for example, we wanted to use exceptions we would fall back into the same problem. Alternatively, if we try to anticipate all possible kinds of effects, then we may as well have used a language with implicit effects.

***Advice can introduce implicit effects*** One very pragmatic solution for the problem would be to allow a language where the base programs are purely functional, but advice can introduce implicit side-effects. The original design of AspectFun can be seen as an example of this design. In AspectFun sequential composition is allowed on advice and the expressions being composed can introduce *IO* computations. However, sequential composition is not allowed on the base programs. The tracing example above is an example of a program written in this style. Nonetheless, even though only advice can introduce side-effects we can still break properties that we would expect from a pure functional program (such as for example parametricity properties). The advantage is that if we ignore all advice, we still have a pure functional program. As discussed earlier, one possible way to recover most (if not all) properties of the original pure functional program, while still allowing advice to introduce side-effect, may be through the use of something similar to Dantas and Walker's harmless advice [13].

***Type refinement in advice*** A more sophisticated alternative would be to allow advice to perform some form of type refinement on the original type of the base program. The idea here would be that the base program could declare a type that specifies that the program may involve some side-effect, but it is unknown which specific effect that is. Although we do not know any AOP language with this design, the library approach based on mixins by Oliveira [11] allows the modular development of pure functional programs in a similar way. For example, in the mixin approach, the type of the base program for the interpreter presented in Section 4.2 would be:

$$eval :: Monad\ m \Rightarrow Mixin\ (Expr \rightarrow m\ Int)$$

and the code corresponding to the tracing aspect would refine that type as follows:

$$teval :: MonadWriter\ String\ m \Rightarrow Mixin\ (Expr \rightarrow m\ Int)$$

The important thing to note here is that *teval* is allowed to make use of the monad writer operations in its definition; but *eval* does not need to prepare for that possibility in advance. Unlike the mixin approach, with an AOP language we would not need to to combine the programs explicitly, which would make it more oblivious. However, we would still need to anticipate the existence of *some* effects and declare a type that can then be refined by the advice. While this would, perhaps, make the approach a bit less oblivious than usual AOP solutions, it would be much more in line with what is expected from a pure functional language.

### 5.5 Static Typing and Type Inference

All the existing functional AO languages are statically typed; but the inference mechanisms are very different. In AspectFun, functions and aspects are typed separately and then connected by the pointcut. The type of the advice is checked to be more general than the type of each function in the pointcut to ensure soundness. In addition, the type system is supplemented with type predicates, sim-

ilar to the ones found in qualified types to facilitate static weaving of advice. Similarly, in Aspectual Caml, functions and aspects are typed separately. However, there is no type compatibility check on pointcuts. Instead, the weaver goes through the type annotated abstract syntax tree and silently drops advice with mismatching types. This missing connection allows aspects to be compiled independently of the functions it advises, at the cost of losing error reporting. AspectML strives to give a concrete type to pointcuts, as they are first-class entities in the language. This proves to be difficult: higher-order unification is needed when more than one function appears in a pointcut. To regain decidability, mandatory type annotations are required with certain constructs of the language.

## 6. Other Related Work

In this section we discuss some other related work.

### 6.1 Type-Directed Programming and Generic Programming

Type-directed programming is an important ideology in strongly typed functional languages. It is essentially about dispatching program behaviour based on the input type. In general, approaches to type-directed programming can be divided into two groups. *Nominal* approaches, such as Haskell's type classes [51], stipulate a separate implementation of a type-directed function for each input type of interest; structurally similar but nominally different types have unrelated implementations. This is more refined – customised behaviours can easily be provided for functions – but less reusable.

On the contrary, *structural* approaches like Generic Haskell [24] map every type into a fixed finite structural *view*, which allows generic functions to be defined once for all types, even those yet to be conceived. This works nicely in most situations, but not all. We need some means of overriding generic behaviour without endangering modularity; this is not possible in most of the current approaches to generic programming.

The line that separates the nominal and structural approaches is not always clear cut. Type classes are a popular tool to encode structural generic programming. There have been some efforts [32, 45, 57], to reconcile genericity and extensibility with a type class based solution for generic programming.

Type-scoped advice is nominal. Compared to type classes, it is more flexible: we can conveniently combine it with structural approaches and deploy it when needed as we see in Section 3.2. In [55], AspectML is used for type-directed programming and extensible generic programming, which inspired our discussion in this paper.

### 6.2 Open Extensibility

Open extensibility, better known by the pun of the *expression problem*, is essentially about supporting modular extension of datatype variants and functions at the same time.

The design of Aspectual Caml is centred around the expression problem. Its static introduction mechanism allows direct injection of new variants into existing datatypes. This solution is more convenient than ours since different expressions have the same type, which makes the defining of certain functions (such as environment lookup) easier. However, it remains a challenge to compile static introduction modularly, a core requirement for the expression problem. A similar problem exists in the proposal of adding open datatypes and open functions into Haskell [38]. In that work, datatype variants as well as clauses of functions can be declared separately and grouped together at link time. This is not truly modular despite the fact that the impact of recompilation can be reduced by the techniques mentioned in the paper.

Our approach only deals with open functions, and lifts variants of datatypes to types for extensibility. The weaving of aspects is at the use site, which is separately compiled from function definitions. Nevertheless, this concept of modularity is different from the traditional one. Since we do not require advanced planning on whether a function will be advised or how many times it may be advised, the behaviour of the function is always subject to changes. Therefore, introduction of advice on this function affects all definitions that depend on it.

Extensible ML (EML) [43] is a much heavier-weight approach to the problem of open extensibility, which basically completely redesigns ML by proposing a very different syntax and semantics. Datatypes and variants are encoded as super- and sub-classes that are modularly extensible.

Polymorphic variants are a language and type system extension implemented in OCaml [35] and proposed for Haskell [20, 34]. Polymorphic variants are declared independently to type definitions and types are formed as collections of such variants. Thus, new variants can be added easily without affecting existing programs. However, polymorphic variants do not induce open functions. It would be interesting to explore how polymorphic variants could be combined with our AOP approach.

Oliveira et al. [45] addressed the problem of *extensible generic functions* with Haskell type classes and noted the connection to the expression problem. In a more recent development, Oliveira [12] proposed a solution to the expression (families) problem inspired by his earlier work and, more generally, showed how to encode *extensible datatypes* in System $F_\omega$-like languages extended with with record subtyping. Swierstra [48] has also proposed a solution to the expression problem using extensible sums (or variants) that has some close similarities to Oliveira et al.'s technique.

The problem of open extensibility is also studied in object-oriented frameworks. In [58], the authors added algebraic datatypes and pattern matching into an objected-oriented language, and argued that through the introduction of defaults they could reverse the subtyping relationship and declare datatypes that extend variants as subtypes of the original datatypes. As a result, standard object-oriented mechanisms such as subtyping extension and overriding can be deployed for extensibility of datatypes.

There is also a folklore encoding of open extensions in Haskell through type classes. Open functions can be declared as class methods, which overload on variants that have been lifted to types. Since each new open function requires a new class, code overhead of this approach is significant. As we have seen in Section 3.2, this highly exclusive approach makes it difficult to combine the approach with other programming methodologies. The power of pattern matching is compromised and there is no easy way of encoding nested patterns.

### 6.3 Inheritance in Functional Programming

Traditionally in functional languages, code reuse is achieved through higher-order functions as combinators, typical examples of which are *fold*, *unfold*, *map* etc. However, it is also possible to achieve reuse using mechanisms akin to inheritance (as usually found in most object-oriented languages). Cook was probably the first to note that inheritance had uses other than object-oriented programming in his work on the denotational semantics of inheritance [10]. In that work, he used several different variations of *mixins* to model different existing kinds of inheritance present at the object-oriented programming languages of the time. McAdam shows how some effects can be simulated (without using monads) using mixins and he presents a type-inference algorithm where the treatment of error messages is modularly defined [40]. Garrigue employs open recursion to emulate *open functions* in his solution to the expression problem with *polymorphic variants* [19]. Läufer shows how to apply mixins to interpreters and how to define mutually-recursive functions using mixins [33]. He also ar-

gues about the relation of his technique with the OO VISITOR pattern [18]. A nice application of inheritance to a problem of separation of concerns is given by Brown and Cook [6], who show how to approach the problem of memoization in purely functional languages using *monadic memoization mixins*. For many problems involving separation of concerns it is possible to use mixin inheritance to provide solutions for these problems. In recent work, Oliveira has shown how to use mixins to solve many problems traditionally solved using AOP-like techniques [11]. A drawback of solutions with mixins is that they are less oblivious than typical AOP approaches and additional parametrization is required.

## 7. Conclusion

As far as we are aware, this paper is the first extensive survey of the impact of AOP in (pure) functional programming. We have identified the main strengths of AOP, namely extensibility and adaptability; and separation of concerns. For each of the two, we demonstrated classical functional applications where modularity was traditionally believed difficult to achieve. The AOP solutions to the problems are lightweight and blend in well in a functional style. At the same time, we also identified the major challenges of a satisfactory functional AOP language – sound reasoning, parametricity, and effects – and discussed possible design options.

It is interesting to observe that, in contrast to the traditional concept of crosscutting in the OO setting where aspects typically crosscut several classes, the majority of the applications of aspects in functional programming only involve a single function in the pointcut. We believe the realisation of this difference as concluded by this paper is important to both the functional and AOP community. There is a pressing need to properly interpret and develop of the concept of 'crosscutting' in the functional setting before functional AOP spreads its wings.


#### Acknowledgements

We are thankful to Jeremy Gibbons who provided valuable feedback and helped in improving the presentation of the paper. The first author would like to thank Siau-Cheng Khoo and Kung Chen for commenting on an earlier draft. We are also thankful to the anonymous reviewers for their constructive and encouraging reviews. This work is supported by the EPSRC grant *Generic and Indexed Programming* (EP/E02128X).


## References


[1] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 144–168, 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):487–504, 1984.

[4] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[6] D. Brown and W. R. Cook. Monadic memoization mixins. Technical Report TR-07-11, The University of Texas, February 2007.

[7] K. Chen, S.-C. Weng, M. Wang, S.-C. Khoo, and C.-H. Chen. A compilation model for aspect-oriented polymorphically typed functional languages. In *Static Analysis, 14th International Symposium, SAS 2007*, volume 4634 of *LNCS*, pages 34–51. Springer-Verlag, 2007.

[8] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, 2002.

[9] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL '02: Proceedings of Foundations of Aspect-Oriented Languages Workshop*, pages 33–44, 2002.

[10] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[11] B. C. d. S. Oliveira. The different aspects of monads and mixins. http://www.comlab.ox.ac.uk/files/2136/Mixins.pdf, 2009.

[12] B. C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In S. Drossopoulou, editor, *23rd European Conference on Object Oriented Programming (ECOOP)*, July 2009.

[13] D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 383–396, New York, NY, USA, 2006. ACM Press.

[14] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007.

[15] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM.

[16] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, Boston, Massachusetts, USA, 2005.

[17] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Int'l Conf. on Functional Programming (ICFP'02)*, pages 48–59, Oct. 2002.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering, Sasaguri, Japan*, 2000.

[20] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report, 1996. Technical report NOTTCS-TR-96-3, University of Nottingham.

[21] J. Gibbons and J. Jeuring, editors. *Generic Programming*. Kluwer Academic Publishers, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloß Dagstuhl, July 2002.

[22] C. V. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler. The glasgow haskell compiler: A retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 62–71, London, UK, 1993. Springer-Verlag.

[23] R. Hinze. Generics for the masses. *J. Functional Programming*, 16(4 & 5):451–483, July & September 2006.

[24] R. Hinze and J. Jeuring. Generic Haskell: practice and theory. In *In Generic Programming, Advanced Lectures, volume 2793 of LNCS*, pages 1–56. Springer-Verlag, 2003.

[25] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In P. Wadler and M. Hagiya, editors, *Eighth International Symposium on Functional and Logic Programming*, April 2006.

[26] R. Hinze and A. Löh. Generic programming in 3d. *Science of Computer Programming*, 2007.

[27] J. Jeuring and P. Jansson. Polytypic programming. In *2nd Int. School on Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.

[28] S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL '04: Proceedings of Foundations of Aspect-Oriented Languages Workshop*, 2004.

[29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[30] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[31] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[32] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, Sept. 2005.

[33] K. Läufer. What functional programmers can learn from the Visitor Pattern. Technical report, Loyola University Chicago, March 2003.

[34] D. Leijen. Extensible records with scoped labels. In *Symposium on Trends in Functional Programming*, pages 297–312, 2005.

[35] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.09 – documentation and users manual, 2007. Institut National de Recherche en Informatique et en Automatique.

[36] Libra academic search website. http://libra.msra.cn/paper_category_4.htm.

[37] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46:2003, 2003.

[38] A. Löh and R. Hinze. Open datatypes and open functions. In *Principles and Practice of Declarative Programming*, 2006.

[39] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 320–330. ACM Press, Sept. 2005.

[40] B. J. McAdam. That about wraps it up — using fix to handle errors without exceptions, and other programming tricks. Technical report, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1997.

[41] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[42] B. Meyer. *Eiffel : The Language*. Prentice Hall PTR, 1991.

[43] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions, 2002.

[44] B. C. d. S. Oliveira and J. Gibbons. TypeCase: a design pattern for type-indexed functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 98–109, New York, NY, USA, 2005. ACM.

[45] B. C. d. S. Oliveira, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In H. Nilsson, editor, *Trends in Functional Programming*, April 2007.

[46] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. *SIGSOFT Softw. Eng. Notes*, 29(6):147–158, 2004.

[47] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM Press.

[48] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[49] T. Uustalu and V. Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263 – 284, 2008.

[50] P. Wadler. Theorems for free! In *Functional Programming and Computer Architecture*, 1989.

[51] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

[52] M. Wang, K. Chen, and S.-C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *Partial Evaluation and Program Manipulation*, pages 78–87. ACM Press, 2006.

[53] G. Washburn. Generalizing parametricity using information-flow. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society.

[54] G. Washburn and S. Weirich. Good advice for type-directed programming. In *Workshop on Generic Programming 2006*. ACM Press, 2006.

[55] G. Washburn and S. Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33–44, New York, NY, USA, 2006. ACM Press.

[56] S. Weirich. Type-safe cast: Functional pearl. In *In Proc. ICFP, Montreal*, pages 58–67. ACM Press, 2000.

[57] S. Weirich. RepLib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12, New York, NY, USA, 2006. ACM Press.

[58] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.