A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages (Technical Report)

Meng Wang
National University of Singapore
wangmeng@comp.nus.edu.sg

Kung Chen
National Chengchi University
chenk@cs.nccu.edu.tw

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

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

Chung-Hsin Chen
National Chengchi University
g9403@cs.nccu.edu.tw

Abstract

Introducing aspect orientation to a polymorphically typed functional language strengthens the importance of *type-scoped advices*; i.e., advices with their effects being harnessed by type constraints. As types are typically treated as compile time entities, it is desirable to be able to perform *static weaving* to determine at compile time the chaining of type-scoped advices to the invocations of their associated join points. In this paper, we describe a compilation model, as well as its implementation, that enables static type inference and static weaving of programs in an aspect-oriented polymorphically typed functional language, AspectFun . We describe a type-directed weaving scheme that successfully, and coherently, weaves type-scoped advices into base programs, in the presence of nested and second-order advices. We also demonstrate how control-flow based pointcuts (such as *cflow* and *cflowbelow*) are compiled away, and describe several type-directed optimization strategies that can improve the efficiency of woven code.

1. Introduction

Aspect-oriented programming (AOP) thrives in facilitating software development through separating and modularising cross-cutting concerns [8]. It provides a new language feature called *aspect* that encapsulates such concern. It also defines an underlying dynamic semantics that enables interaction between method invocations/executions at the base program and aspects through a technique known as *weaving*.

In addition to addressing cross-cutting concerns, aspects and their weaving mechanisms also strengthen the practice of incremental software development. Specifically, functional behaviour of computing objects can be incrementally enhanced through introduction of aspects, including *nested* aspects that further enhance existing aspects. Such behaviour enhancement can be effectively managed by aspects with *bounded scope*, i.e., aspects which are designed to interact with a controlled class of method invocations.

Bounded-scope aspects are particularly useful when AOP paradigm is supported by a strongly-typed polymorphic functional language, such as Haskell or ML. The ability to limit the effect of aspects through types greatly enhances the usability of aspects in functional program development. For instance, the following code declares three aspects labelled by `n3`, `n4` and `n5` respectively, which designate execution of function `h` as their pointcut. They provide advices to the execution of a group of calls to function `h`, which is defined in the base program.

Example 1

```
// Aspects
n3@advice around {h} (arg) =
   proceed arg;
   println "exiting from h"
n4@advice around {h} (arg:[a]) =
   println "entering with a list";
   proceed arg
n5@advice around {h}
   (arg:[Char]) =
   print "entering with ";
   println arg;
   proceed arg
// Base program
h x = x
f x = h x
(f "c", f [1], h [2])
```

```
// Execution trace
 entering with a list
 entering with c
 exiting from h
 entering with a list
 exiting from h
 entering with a list
 exiting from h
```

□

As with other AOP, we use `proceed` in this example as a special function which may be called inside the body of an `around` advice. It is bound to a function the represents "the rest of the computation at the advised function". For easy presentation sake, We use `around` advice throughout the paper, and omit the use of `before` and `after` advices. It is easy to see that both the latter advices can be simulated by `around` advices that always proceed.

In this example, advice `n3` renders advice to all executions of `h`. Advice `n4` limits the scope of its impact through type scoping on its first argument; this is called a *type-scope* advice. This means that `n4` is only triggered when executions of `h` has an argument of list type. Lastly, the type-scoped advice `n5` only provides special treatment to executions of `h` when the arguments are strings. Using type-scoped aspects enable us to have customized, type-dependent tracing message. Note that *String* (a list of *Char*) is treated differ-

ently from ordinary lists. Assuming a textual order of advice triggering, the corresponding trace messages produced by executing the complete program is displayed to the right of the example code.

Type-scoped advice does not only enable finer control of functions' behaviors, it can also be used to guide the development of functions in a type-directed fashion, as advocated by Washburn and Weirich [20]. In line with the spirit of *well-typed programs never go wrong* [12], it is imperative to have a *static type checker* that ensures type-scoped advices do not lead to runtime type errors during program execution.

Furthermore, as types are typically treated as compile-time entities, their use in controlling advices can usually be determined at compile-time. Consequently, it is desirable to perform *static weaving* of advices into based program at compile time to produce an integrated code without explicit declaration of aspects. Static weaving also brings forth another appealing advantage: As pointed out by Sereni and de Moor [16], performing static analysis over aspect-oriented programs has been found to be difficult and non-intuitive, because of the interwound semantics defined by aspects and base program. Such difficulty can be circumvented by performing the corresponding static analysis over the integrated woven code produced by static weaving.

Despite its benefits, static weaving is never a trial task, especially in the presence of type-scoped advices. Specifically, it is not always possible to determine *locally* at compile time if a particular advice should be triggered for weaving. Consider Example 1, from a syntactic viewpoint, function `h` can be called in the body of `f`. If we were to naively infer that the argument `x` to function `h` in the RHS of `f`'s definition is of polymorphic type, we would be tempted to conclude that (1) advice `n3` should be triggered at the call, and (2) advices `n4` and `n5` should not be called as its type-scope is less general than $a \rightarrow a$. As a result, only `n3` would be statically applied to the call to `h`.

Unfortunately, this approach would cause incoherent behavior of `h` at run-time, as only the third trace message "`exiting from h`" would be printed. This would be incoherent because the invocations `(h [1])` (indirectly called from `(f [1])`) and `(h [2])` would exhibit different behaviors even though they would receive arguments of the same type.

Most of the work on aspect-oriented functional languages does not address this issue of static and coherent weaving. In AspectML [3] (*a.k.a* PolyAML [2]), dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [11] takes a syntactic approach which sacrifices coherence[1] for static weaving.

In this paper, we present a compilation model for AspectFun , an aspect-oriented polymorphically typed functional language with lazy semantics. (Example 1 depicts an AspectFun program.) The overall compilation process is illustrated in Figure 1. Briefly, the model comprises the following three major steps: (1) Static type inference of an aspect-oriented program; (2) Type-directed static weaving to produce a single woven code and convert advices to functions; (3) Type-directed optimization of the woven code. In contrast with our earlier work [19], we have extended our research in three dimensions:

1. Language features: We have included a suite of features to our aspect-oriented functional language, AspectFun . They are: *nested advices* (i.e., invocation of advice within the body of another advice), *second-order advices* (i.e., declaration of advices that aim to advise other named advices), and complex pointcuts, including control-flow based pointcuts (a.k.a. `cflow` and `cflowbelow`), and `any`.
2. Algorithms: We have extended our type inference and static weaving strategy to handle the extension of the language features. (Though not presented in this paper, we have devised a deterministic type-inference algorithm to determine the well-typedness of aspect-oriented programs.) We have also provided a strategy for transforming advices with control-flow based pointcuts, and a set of analysis and optimization strategies to enhance the performance of woven codes.
3. Systems: We have provided a complete implementation of our compilation model turning aspect-oriented functional programs into executable Haskell code.

[1] Our notion of coherence admits semantic equivalence among different invocations of a function with the same argument. It should not be confused with the coherence concept defined in qualified types [5] which states that different translations of an expression are semantically equivalent.

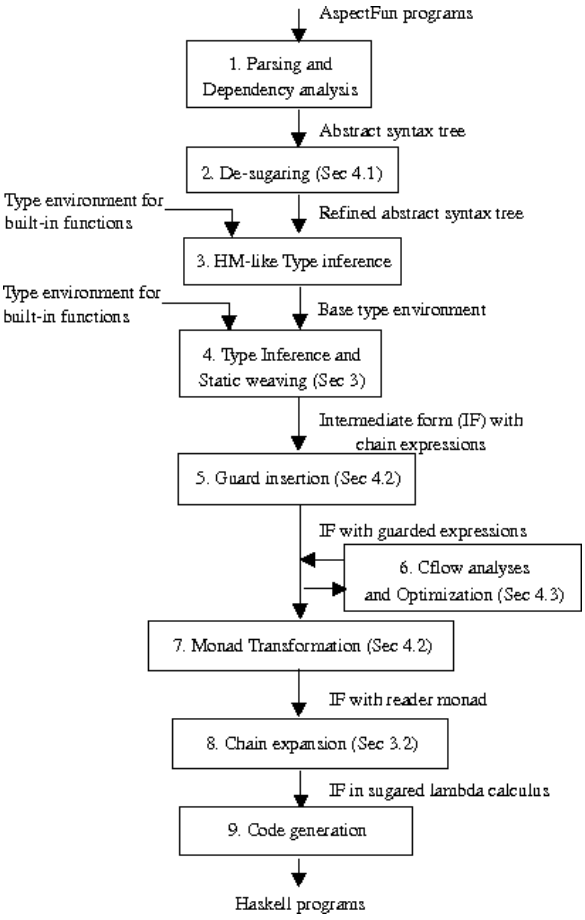

Figure 1. Compilation Model for AspectFun

Under our compilation scheme, the program in example 1 is first translated through static weaving to an expression in lambda-calculus with constants for execution. For presentation sake, we express the result of static weaving in an intermediate form as follows:

```
let n3 = \arg -> (proceed arg ;
                  println 'exiting from h') in
let n4 = \arg -> (print \entering h with a list' ;
                  proceed arg) in
let n5 = \arg -> (print \entering h with ' ;
                  println arg;
```

```
                    proceed arg) in
let h x = x in
let f dh x = dh x in
(f <h,{n3,n4,n5}> ``c'', f <h,{n3,n4}> [1],
                           <h,{n3,n4}> [2])
```

Note that all advice declarations are translated into functions and are woven in. The intermediate form contains two special syntactic constructs: The first is a special keyword `proceed`, which has been retained from the original aspect-oriented language. The second is a special syntax $\langle \_ , \{\ldots\} \rangle$, called *chain expression*, used to chain together advices and advised functions. For instance, $\langle \texttt{h} , \{\texttt{n3}, \texttt{n4}\} \rangle$ denotes the chaining of advices `n4` and `n3` to advised function `h`. In the above example, the two invocations of `h`, with integer-list arguments, in the original aspect program have been translated to invocations of the chain expression $\langle \texttt{h} , \{\texttt{n3}, \texttt{n4}\} \rangle$. This shows that our weaver respects the coherence property.

These two special constructs aim to facilitate our presentation. In actual implementation, the `proceed` keyword is replaced by a parameter named `proceed` which is local to the advice (which has been translated into function). The chain expression is also expanded into series of function applications.

This coherent weaving of advices to `h` entails passing appropriate chain expressions of `h` to those function calls in the program text from which `h` may be called indirectly. This requirement is satisfied by allowing functions of those affected calls to carry extra parameters. In the code above, the translated definition of `f` carries such an additional parameter, `dh`. The original `(f [1])` call is then translated to `(f <h,{n4,n3}> 1)`, in which the chain expression for `h` is passed.

All the technically challenging stages in the compilation process are explained in detail – in their respective sections – in the rest of this paper. For ease of presentation, we gather all compilation processes pertaining to control-flow based pointcuts in Section 4.

The outline of the paper is as follows: Section 2 describes an aspect-oriented language and provides background information and terminologies used. In Section 3, we describe our type inference system and the corresponding type-directed static weaving process. In section 4, we provide a detailed description of how control-flow based pointcuts are handled in our compilation model. The various parts of the compilation process involved include de-sugaring, and `cflow` analyses and optimizations. We discuss related work in Section 5, before concluding in Section 6.

2. AspectFun : The Aspect Language

In this section, we introduce an aspect-oriented functional language, AspectFun , for our investigation. Figure 2 presents the syntax of the language. We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and $fv(o)$ as the free variables in o. Note that we generally assume $\bar{o}$ and o denote non-related objects which should not be confused. We write $t_1 \sim t_2$ to specify unification. We also define the match operation between two types t and t', denoted by $\trianglerighteq$, in the standard manner. Specifically, $t \trianglerighteq t'$ iff there exists a substitution S over type variables in t such that $St = t'$. Besides, we write $t \equiv t'$ iff $t \trianglerighteq t'$ and $t' \trianglerighteq t$. For simplicity, we leave out type annotations, user defined data types, `if` expressions, sequencings (`;`) and pattern matchings but may make use of them in examples.

In AspectFun , a program is a sequence of declarations followed by a main expression. Besides global variables and functions, we can also declare aspects. An *aspect* is an advice declaration which includes a piece of advice and its target *pointcuts*. An *advice* is a function-like expression that executes when the functions designated at the pointcut are about to execute. As stated earlier, we only support *around* advice. Pointcuts are denoted by $\{\overline{pc}\}\ (arg)$, where

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

Figure 2. Syntax of the AspectFun Language

pc stands for either a *primitive pointcut*, represented by ppc, or a *composite pointcut*.

As with other aspect-oriented languages, pointcuts pick out certain join points in the program flow for advising. Since our language is a functional one, we focus on join points of function invocations. Thus the primitive pointcut, ppc, specifies which function invocations will be selected for advising. Furthermore, since functions are first-class values in our langauge, a function can be invoked directly through name-based calls as well as indirectly through aliasing or functional arguments which are passed to a higher-order function. Therefore, in order to catch all potential invocations of a function, our pointcuts behave like the *execution* pointcuts of AspectJ, though after translations advices are chained with function identifiers, which are then executed at call invocations.

The specification of a primitive pointcut can be a function's name (f), a catch-all keyword any, or any with an exclusion list of function names. For example, the pointcut any$\backslash[f, g]$ will select all functions except f and g. Besides, since advices are also named, we allow advices advising other advices. We will see such an example shortly. The sequence of pointcuts, $\{\overline{pc}\}$, indicates the union of all the sets of join points selected by the pc's. The argument variable *arg* is bound to the actual argument of the function call and it may contain a type scope. Note that only global functions and advices are subject to advising; and invocations of anonymous function are not considered as join points, even when any is used. Alpha renaming is applied to local declarations beforehand so that to avoid name capturing. We shall describe the composite pointcuts later.

Our aspect language is polymorphic and statically typed. Basic types such as booleans, integers, characters, tuples, and lists are predefined and their constructors are recorded in some initial environment. We also have the standard constructs of types and type schemes found in the Hindley-Milner type system. In addition, central to our approach is the construct of *advised types*, ρ in Figure 2, inspired by the *predicated types* [18] used in Haskell's type classes. These advised types augment common type schemes with *advice predicates*, $(f : t)$, which are used to capture the need of advice weaving dependent on the type context. We shall explain them in detail in Section 3.1. In the subsequent subsections, we use examples to illustrate the major features of AspectFun .

2.1 Handling Crosscutting Concerns

Lists are the most used data structures in functional programming. For instance, consider the `reverse` function which reverses first input list and stores result in the second.

Example 2

```
reverse [] accum = accum
reverse x accum = reverse (tail x)
                  (cons (head x) accum)
```

□

This code pattern is very common among important list operations such as `append`, `mergeSort` etc, to name a few. The first clause is to be executed when `x` is empty. However, a non-experienced programmer may easily define the two clauses in the wrong order.

```
reverse x accum = reverse (tail x)
                  (cons (head x) accum)
reverse [] accum = accum
```

Calling this function with any lists will end up with a run-time error when `head` is applied to an empty list. To remove the dangling base-case handling, we can use an aspect to crosscut all list functions of this code pattern and provide tests on empty list as follows.

```
n@advice {reverse, append, mergeSort, ...} (arg) =
     \x . if arg == [] then x
                       else (proceed arg) x

reverse x accum = reverse (tail x)
                  (cons (head x) accum)
```

Here the pointcut lists all the applicable list functions. When those functions are invoked, the advice will be triggered to check if the argument is an empty list. If so, it will replace the underlying invocation with the identity function (i.e. returning the `accum` parameter); otherwise, the invocation is resumed by `proceed`.

Enumeration of function names can be troublesome and non-extensible. In this case, we can consider the catch-all pointcut `any` augmented with a type scope. Better still, if we extend our syntax and generalize the `any` pointcut to include module quantifiers, such as `List.any`, we can quantifying `any` over a module of list operations of the above mentioned pattern as follows.

```
n@advice {List.any} (arg::[a]) =
     \x . if arg == [] then x
                       else (proceed arg) x
```

2.2 Control-flow Based Pointcuts

The composite pointcuts of our aspect language are mainly those related to the control flow of a program. Specifically, we can write pointcuts which identify a subset of the invocations of a specific function based on whether they occur in the dynamic context of other functions. For example, the pointcut f + `cflow`(g) selects those invocations of f which are made when the function g is still executing (i.e. invoked but not returned yet). On the other hand, if the operator before the `cflow` designator is a minus sign, it means the opposite, namely only invocations of f which are not under the dynamic context of g will be selected.

Following AspectJ, our aspect langauge also provides two kinds of pointcut designators for specifying control flow restrictions. The first one is expressed as `cflow`(f), and it captures all the join points in the control flow of the specified function f, including the function itself. The second one is expressed as `cflowbelow`(f), and it excludes the specified function. Their difference is best illustrated by the case when the function specified is recursive. For example, in the following simple aspect program, we intend to use advice `n` to advise the recursive `fac` function only once, when it is first executed, via the pointcut "`fac-cflowbelow(fac)`". Had we used "`fac-cflow(fac)`", the advice would not be executed at all.

```
n@advice around {fac - cflowbelow(fac)} (arg) =
      println "Entering fac"; proceed arg in
fac x = if x==0 then 1 else x * fac (x-1) in
fac 3
```

The ability of control-flow based pointcuts to inspect the run-time stack is important to many security applications. Suppose a function `f`'s access to some sensitive code is only enabled by being called from a highly trusted function `g`; Failing in doing so, `f` can only be executed as partially trusted. This policy can be enforced by an aspect.

Example 3

```
n@advice around {f + cflow(g)} (arg)
            = ... // fully trusted execution
f = ... // partially trusted execution
```

□

The above aspect effectively performs a stack walk when `f` is executed and only grants fully trusted execution when `g` is in the dynamic context.

In addition to un-scoped control-flow based pointcuts, AspectFun allows us to specify fine grained ones by augmenting the arguments with type scopes. The advice `n` in the above example can be refined to

```
n@advice around {f + cflow(g(_::[Bool]))} (arg)
        = ... // fully trusted execution
```

In this case, the *cflow* pointcut is only matched when a `g`-call with input of type $[Bool]$ is found inside the dynamic context. Note that we use `_` to indicate no value binding is allowed in control-flow based pointcuts.

However, deriving an efficient static weaving scheme for advices with control-flow based pointcuts is not straightforward, particularly in a statically typed functional language . We employ an implementation scheme similar to that of AspectJ [10], which uses a stack counter to spare the efforts of maintaining a run-time stack with the aspects. Furthermore, we also perform some static analysis to reduce the runtime overhead of executing control-flow based advices. The detailed scheme will be presented in 4.

2.3 Advising Advices and Advice Bodies

In our langauge, aspects are not limited to observing executions of the base program. As shown in Figure 2, the syntactic category of primitive pointcuts, ppc, also includes advice n. In other words, we can develop advice which advises other advices. We refer to such advices as *second-order advice*s. In contrast, the two-layered design of AspectJ like languages only allow advices to advise other advices in a very restricted way. The loss of expressiveness of such an approach has been well argued in [15].

The following program shows an example of using second-order advices. Its purpose is to compute the total amount of a customer order and apply discount rates according to certain business rules.

Example 4

```
calcPrice cart = sum (map discount cart) in
discount item = (getRate item) * (getPrice item)
```

□

In addition to regular discount rules, there are also other ad-hoc sale discounts that may be put into effect in certain occasions, such as special holiday-sales, anniversary-sales, etc. Due to their ad-hoc nature, it is better to separate them from the functional modules and put them in aspects that advise on the discount rate query function.

```
n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg) in
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)
```

Furthermore, it is common to have some business rules that govern all the sales promotions offered to customers. For example, there may be a rule stipulating the maximum discount rate that is applicable to any product item, regardless of the multiple discounts it qualifies. Such business rules can be realized using aspects of second-order in a modular manner.

```
n3@advice around {n1,n2} (arg) =
   let finalRate = proceed arg
   in  if (finalRate < 0.5) then 0.5 else finalRate
```

Here the second-order advice `n3` has meta-control over advices `n1` and `n2`. The call to *proceed* gets the compounded discount rate and the rule that no product can be sold under 50% of its list price is applied.

In addition to direct advising, we can also write advice that advises other advice indirectly. Specifically, inside the body of an advice definition, there may be calls to other functions that are advised by other advices. We call them *nested advices*. This is particularly important in security applications. Consider a different attempt to invoke the restricted function `f` from Example 3.

Example 3a

```
n@advice around {f + cflow(g)} (arg)
                = ... // fully trusted execution
n1@advice around {w} (arg) = f arg in
f x  = ... // partially trusted execution
h x y = x y in
g x = h w x in
g 1
```

□

In the main expression, the application of `g` invokes the execution of `w` which indirectly calls `f` through application of advice `n1`. In a secure system, this silent execution of `f` must be observed and advice `n` is triggered.

There is a special kind of nested advices, which apply to the execution of their own bodies, directly or indirectly. Our system does not allow them for the reason that circular *around* advices together with potential recursive functions that they are advising may form a scenario similar to polymorphic mutual recursion which threatens the decidability of type inference. We leave this to future investigation.

3. Static Weaving

In this section, we present a type inference system which guarantees type safety and, at the same time, weaves the aspect language through a type directed translation. The static weaving system ignore control-flow based pointcuts by treating advices with composite pointcuts such as $f + \texttt{cflow}(g)$ as ones having pointcuts f. The treatment of control-flow based pointcuts such as $\texttt{cflow}(g)$ will be handled in Section 4.

3.1 Type directed weaving

As introduced in Section 2, $advised\ type$ denoted as ρ is used to capture function names and their types that may be required for advice resolution. We further illustrate this concept with our tracing example given in Section 1.

For instance, function `f` in the introductionin Example 1 possesses the advised type $\forall a.(h : a \rightarrow a).a \rightarrow a$, in which $(h : a \rightarrow a)$ is called an *advice predicate*. It signifies that *the execution of any application of* `f` *may require advices of* `h` *applied with a type which should be no more general than* $a' \rightarrow a'$ *where* a' *is a fresh instantiation of type variable* a.

The notion of *more general* is formally defined as:

Definition 1 *We say a type* t *is more general than or equivalent to a type* t'*, if* $t \trianglerighteq t'$*. When* $t \trianglerighteq t'$ *but* $t \not\equiv t'$*, we say* t *is more general than* t'*. Similarly, we say a type* t *is more specific than a type* t' *if* $t' \trianglerighteq t$ *and* $t \not\equiv t'$*.*

Note that advised types are used to indicate the existence of some *indeterminate advices*. If a function contains only applications whose advices are completely determined, then the function will not be associated with an advised type; it will be associated with a normal (and possibly polymorphic) type. As an example, the type of the advised function `h` in Example 1 is $\forall a.a \rightarrow a$ since it does not contain any application of advised functions in its definition.

$$(\textsc{Gen}) \qquad gen(\Gamma, \sigma) = \forall \bar{a}.\sigma \quad \text{where } \bar{a} = fv(\sigma) \backslash fv(\Gamma)$$

$$(\textsc{Card}) \quad |o_1 ... o_k| = k$$

Figure 3. Auxiliary Definitions

Figure 3 defines a set of auxiliary functions/relations that assists type inference. We define a generalization procedure, (GEN), which turns a type into a type scheme by quantifying type variables that do not appear free in the type environment. The (CARD) function, denoted by $|\cdot|$, returns the cardinality of a sequence of objects.

The main set of type inference rules, as described in Figure 4, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \sigma \rightsquigarrow e'$ to denote that expression e has type σ under type environment Γ and it is translated to e'. We assume that the advice declarations are preprocessed and all the names which appear in any of the pointcuts are recorded in an initial global store A. We also assume that the base program is well typed in Hindley-Milner and the type information of all the functions are stored in Γ_{base}.

The typing environment Γ contains not only the usual type bindings (of the form $x : \sigma \rightsquigarrow e$) but also *advice bindings* of the form $n : \sigma \bowtie \bar{x}$. This states that an advice with name n of type σ is defined on a set of functions $\bar{x}$. We may drop the $\bowtie \bar{x}$ part if found irrelevant. When the bound function name is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish from the non-advised case so that it may appear in a predicate as in rule (PRED). We also use the notation $:_{(*)}$ to represent a binding which is either $:$ or $:_*$. When there are multiple bindings of the same variable in a typing environment, the newly added one shadows previous ones.

Note that while it is possible to present the typing rules without the translation detail by simply deleting the '$\rightsquigarrow e$' portion, it is not possible to present the translation rules independently since typing controls the translation.

$$\text{(VAR)}\quad \frac{x : \forall \bar{a}.\bar{p}.t \rightsquigarrow e \in \Gamma}{\Gamma \vdash x : [\bar{t}/\bar{a}]\bar{p}.t \rightsquigarrow e} \qquad \text{(VAR-A)}\quad \frac{\begin{array}{c} x :_* \forall \bar{a}.\bar{p}.t_x \in \Gamma \quad t' = [\bar{t}/\bar{a}]t_x \quad wv(x : t') \quad \Gamma \vdash n_i : t' \rightsquigarrow e_i \\ \bar{n} : \forall \bar{b}.\bar{q}.t_n \bowtie x \rightsquigarrow \bar{n}' \in \Gamma \quad \{n_i \mid t_i \trianglerighteq t'\} \quad |\bar{y}| = |\bar{p}| \end{array}}{\Gamma \vdash x : [\bar{t}/\bar{a}]\bar{p}.t_x \rightsquigarrow \lambda \bar{y}.\langle x\ \bar{y}\ , \{e_i\}\rangle}$$

$$\text{(APP)}\quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : t_1 \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : t_2 \rightsquigarrow (e_1'\ e_2')} \qquad \text{(ABS)}\quad \frac{\Gamma.x : t_1 \rightsquigarrow x \vdash e : t_2 \rightsquigarrow e'}{\Gamma \vdash \lambda x.e : t_1 \to t_2 \rightsquigarrow \lambda x.e'} \qquad \text{(LET)}\quad \frac{\Gamma \vdash e_1 : \rho \rightsquigarrow e_1' \quad \sigma = gen(\Gamma, \rho) \quad \Gamma, f : \sigma \rightsquigarrow f \vdash e_2 : t \rightsquigarrow e_2'}{\Gamma \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \rightsquigarrow \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}$$

$$\text{(PRED)}\quad \frac{\begin{array}{c} x :_* \forall \bar{a}.\bar{p}.t_x \in \Gamma \quad [\bar{t}/\bar{a}]t_x \trianglerighteq t \\ \Gamma, x : t \rightsquigarrow x_t \vdash e : \rho \rightsquigarrow e_t' \quad x \in A \end{array}}{\Gamma \vdash e : (x : t).\rho \rightsquigarrow \lambda x_t.e_t'} \qquad \text{(REL)}\quad \frac{\Gamma \vdash e : (x : t).\rho \rightsquigarrow e' \quad \Gamma \vdash x : t \rightsquigarrow e'' \quad x \in A \quad x \neq e}{\Gamma \vdash e : \rho \rightsquigarrow e'\ e''}$$

Figure 4. Typing rules for expressions

$$\text{(GLOBAL)}\quad \frac{\Gamma \vdash e : \rho \rightsquigarrow e' \quad \sigma = gen(\Gamma, \rho) \quad \Gamma.id :_{(*)} \sigma \rightsquigarrow id \vdash \pi : t \rightsquigarrow \pi'}{\Gamma \vdash\ id = e\ \mathsf{in}\ \pi : t \rightsquigarrow\ id = e'\ \mathsf{in}\ \pi'}$$

$$\text{(ADV)}\quad \frac{\begin{array}{c} \Gamma.proceed : t \vdash \lambda x.e_a : \bar{p}.t \rightsquigarrow e_a' \quad f_i : \forall \bar{a}.t_i \in \Gamma_{base} \\ t \trianglerighteq [\bar{t}/\bar{a}]t_i \quad \Gamma.n : \sigma \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t' \rightsquigarrow \pi' \quad \sigma = gen(\Gamma, \bar{p}.t) \end{array}}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x) = e_a \mathsf{in}\ \pi : t' \rightsquigarrow n = e_a'\ \mathsf{in}\ \pi'}$$

$$\text{(ADV-AN)}\quad \frac{\begin{array}{c} \Gamma.proceed : t \vdash \lambda x.e_a : \bar{p}.t_x \to t \rightsquigarrow e_a' \quad f_i : \forall \bar{a}.t_i \to t_i' \in \Gamma_{base} \\ S = [\bar{t}/\bar{a}]t_i \trianglerighteq t_x \quad t \trianglerighteq S[\bar{t}/\bar{a}]t_i' \quad \Gamma.n : \sigma \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t' \rightsquigarrow \pi' \quad \sigma = gen(\Gamma, \bar{p}.t_x \to t) \end{array}}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: t_x) = e_a \mathsf{in}\ \pi : t' \rightsquigarrow n = e_a'\ \mathsf{in}\ \pi'}$$

Figure 5. Typing rules for declarations

3.1.1 Predicating and Releasing

Before illustrating the main typing rules, we introduce a *weavable* constraint of the form $wv(f : t)$ which indicates that advice application of the f-call of type t can be decided. It is formally defined as:

Definition 2 *Given a function f and its type $f : t_2 \to t_2'$, if $\forall n.n :_{(*)} \forall \bar{a}.\bar{p}.t_1 \to t_1' \bowtie f \in \Gamma \wedge t_1 \sim t_2 \Rightarrow t_1 \trianglerighteq t_2$, then $wv(f : t_2 \to t_2')$.*

This condition basically means that under a given typing environment, a function's type is no more general than any of its advices. For instance, under the environment $\{n : \forall a.[a] \to [a] \bowtie f, n1 : Int \to Int \bowtie f\}$, $wv(f : b \to b)$ is false because the type is not specific enough to determine whether $n1$ and $n2$ should apply whereas $wv(f : Bool \to Bool)$ is vicuously true and, in this case, no advice applies. Note that since unification and matching are defined on types instead of type schemes, quantified variables are freshly instantiated to avoid name capturing.

There are two rules for variable lookups. Rule (VAR) is standard. In the case that variable x is advised, rule (VAR-A) will create a fresh instance t' of the type scheme bound to *x* in the environment. Then we check weavable condition of $(x : t')$. If the check succeeds (*i.e.,* x's input type is more general or equivalent to any of the advice's), x will be chained with the translated forms of all those advices defined on it, having equivalent or more general types than x has. We give all these selected advices a non-advised type in the translation of them $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$. This ensures correct weaving of nested advices advising the bodies of the selected advices. The detail will be elaborated in Section 3.1.3. Finally, the final translated expression is *normalized* by bringing all the advice abstractions of x outside the chain $\langle \ldots \rangle$. This ensures type compatibility between the advised call and its advices.

If the weavable condition check fails, there must exists some advices for x with more specific types, and rule (VAR-A) fails to apply. Since $x \in A$ still holds, rule (PRED) can be applied. This rule introduces an *advice parameter* to the program (through the corresponding translation scheme). This advice parameter enables concrete *advice-chained functions* to be passed in at a later stage, called *releasing*, through the application of rule (REL).

Before we describe rules (PRED) and (REL) in detail, we illustrate the application of these rules by deriving the type and the woven code for the program shown in Example 1. We use C as an abbreviation for $Char$. During the derivation of the definition of f, we have:

$$\Gamma = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_3 : \forall a.a \to a \bowtie h \rightsquigarrow n_3,\ n_4 : \forall a.[a] \to [a] \bowtie h \rightsquigarrow n_4, n_5 : \forall b.[C] \to [C] \bowtie h \rightsquigarrow n_5\}$$

$$\begin{array}{ll} \text{(VAR)} & \dfrac{h : t \to t \rightsquigarrow dh \in \Gamma_2}{\Gamma_2 \vdash h : t \to t \rightsquigarrow dh} \qquad \text{(VAR)}\ \dfrac{x : t \rightsquigarrow x \in \Gamma_2}{\Gamma_2 \vdash x : t \rightsquigarrow x} \\ \text{(APP)} & \dfrac{}{\Gamma_2 = \Gamma_1, x : t \rightsquigarrow x \vdash (h\ x) : t \rightsquigarrow (dh\ x)} \\ \text{(ABS)} & \dfrac{}{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)} \\ \text{(PRED)} & \dfrac{}{\Gamma \vdash \lambda x.(h\ x) : (h : t \to t).t \to t \rightsquigarrow \lambda dh.\lambda x.(dh\ x)} \end{array}$$

Next, for the derivation of the first element of the main expression, we have:

$$\Gamma_3 = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_3 : \forall a.a \to a \bowtie h \rightsquigarrow n_3,\ n_4 : \forall a.[a] \to [a] \bowtie h \rightsquigarrow n_4, n_5 : \forall b.[C] \to [C] \bowtie h \rightsquigarrow n_5,\ f : \forall a.(h : a \to a).a \to a \rightsquigarrow f\}$$

$$\begin{array}{lll} \text{(VAR)} & \dfrac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : [C] \to [C]).[C] \to [C] \rightsquigarrow f} & @ \\ \text{(REL)} & \dfrac{}{\Gamma_3 \vdash f : [C] \to [C] \rightsquigarrow (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle)} & \ldots \\ \text{(APP)} & \dfrac{}{\Gamma_3 \vdash (f\ \text{“}c\text{”}) : [Char] \rightsquigarrow (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle\ \text{“}c\text{”})} & \end{array}$$

$$@ = \quad \text{(VAR-A)} \quad \frac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad ...}{\Gamma_3 \vdash h : [C] \to [C] \rightsquigarrow \langle h\ , \{n_3, n_4, n_5\}\rangle}$$

We note that rules (ABS),(LET) and (APP) are rather standard. Rule (LET) only bind f with : which signalizes locally defined functions are not subject to advising.

Rules (PRED) and (REL) respectively introduces and eliminates advice predicates. Rule (PRED) adds an advice predicate to a type (Note that we only allow sensible choices of t constrained by $t_x \trianglerighteq t$). Correspondingly, its translation yields a lambda abstraction with an advice parameter. At a later stage, rule (REL) is applied to release (*i.e.*,remove) an advice predicate from a type. Its translation generates a function application with an advised expression as argument.

3.1.2 Handling Advices

On top of expressions, declarations define top-level bindings including advices. The typing rules are presented in Figure 5. We use a judgement $\Gamma \vdash \pi : \sigma \rightsquigarrow \pi'$ which closely reassembles the one for expressions.

The rule (GLOBAL) is very similar to (LET) with the tiny difference that (GLOBAL) will bind id with : when it is not in A; and with $:_*$ otherwise.

There are two type-inference rules for handling advices. Rule (ADV) handles non-type-scoped advices, whereas rule (ADV-AN) handles type-scoped advices. In rule (ADV), we firstly infer the (possibly advised) type of the advice as a function $\lambda x.e_a$ under the type environment extended with `proceed`. The advice body is therefore translated. Note that this translation does not necessarily complete all the chaining because the weavable condition may not hold. In this case, just like functions, the advice is parameterized. At the same time, an advised type is assigned to it and only released when it is chained in rule (VAR-A).

After type inference of the advice, we ensure that the advice's type is more general than or equivalent to all functions' in the pointcut. Note that the type information of all the functions is stored in Γ_{base}. Then, this advice is added to the environment. It does not appear in the translated program, however, as it is translated into a function awaiting for participation in advice chaining.

In rule (ADV-AN), variable x can only be bound to a value of type t_x such that t_x is no more general than the input type of those functions in the pointcut. This constraint is similar to the subsumption rule used for type annotations which requires the annotated type to be no more general than the inferred one. For each function in the pointcut, we match a freshly instantiation of the input type t_i to t_x which results in a substitution S. The output type of the advice t is expected to be more general or equivalent to the type of each functions under the substitution S.

In addition, as all the advices are of function types, attempts to advice a non-function type expression will be rejected by the type system.

3.1.3 Advising Advice Bodies

As mentioned in the previous (sub)section, the rules (ADV) and (ADV-AN) make an attempt to translate advice bodies. However, just like the translation of function bodies, the local type contexts may not be specific enough to satisfy the weavable condition. Consider a variant of Example 3a. The control-flow based pointcut `cflow(g)` is removed, and, for illustration purpose, a type scope is added. This is to concentrate on translation of the nested advice and to leave weaving of `cflow` to the next section.

Example 3b

```
n@advice around {f} (arg::Int)
                  = ... // fully trusted execution
n1@advice around {w} (arg) = f arg in
h x y = x y in
h w 1
```

□

Here, advice `n1` calls `f` which is in turn being advised. The goal of our translation is to chain advices which are applicable to the execution of `f` inside an advice. Concretely, when a call to `w` is chained with advice `n1`, the body of `n1` must also be advised. Moreover, the choice of advices must be coherent.

At the time when the declaration of `n1` is translated, the body of the advice is translated. An advised type is given to it since the weavable condition $wv(f : a \to a)$ from the current context is not satisfied.

When the translation attempts to chain an advice in rule (VAR-A), the judgment $\Gamma \vdash n_i : t' \rightsquigarrow e_i$ in the premise forces the advice to have a non-advised type. This is to ensure that all the advice abstractions are fully released so that chaining can take effect.

In the case that this derivation fails, it signifies that the current context is not sufficiently specific for advising some of the calls in this advice's body, and chaining has to be delayed. In example 3, the call to `w` by passing it as an argument to `h` is of type $Int \to Int$. This is sufficiently specific for advising `w`, since `n1` is the only candidate. Consequently, the call to `f` inside the body of `n1` is also of type $Int \to Int$. Now, the weavable condition, $wv(f : Int \to Int)$, is satisfied and the program is translated as follows.

```
n = \arg. ...//fully trusted execution
n1 df = \arg. df arg in
h x y = x y in
h <w,{n1 <f,{n}>}> 1
```

Advice `n` is only chained in the main expression where the context is sufficiently specific for both the calls to `w` and `f`.

The translation of candidate advices $\Gamma \vdash n_i : t' \rightsquigarrow e_i$ in rule (VAR-A)'s premise not only translates bodies of advices but also takes care of chainings of second-order advices.

However, in the premises of rule (ADV) and (ADV-AN), we note that type information of advices is not stored in Γ_{base}. Thus, we replace $f_i : \forall \bar{a}.t' \in \Gamma_{base}$ by $n_i :_* \forall \bar{a}.\bar{q}.t' \in \Gamma$.[2] By doing this, we assume advised advices are translated before the advices defined on them. This is valid because circular cases are precluded.

Thus, example 4 is translated into

```
n1 = \arg.(getHolidayRate arg)*(proceed arg) in
n2 = \arg.(getAnniversaryRate arg)*(proceed arg) in
n3 = \arg.let finalRate = proceed arg
          in  if (finalRate < 0.5)
              then 0.5 else finalRate in
calcPrice cart = sum (map discount cart) in
discount item = (<getRate,{<n1,{n3}>,<n2,{n3}>}>
                    item)
                *(getPrice item)
```

Note that advices `n1` and `n2` are chained with `n3` before the chaining to `getRate`.

3.1.4 Advising Recursive Functions

We have seen our predicating/releasing system working well for non-recursive function. However, if we apply rule (REL) to a call of an advised recursive function, it may end up looping indefinitely.

[2] Advices defined on functions cannot be treated this way because of possible recursiveness of the functions.

Let's illustrate this with our reverse example. The code is reproduced below.

```
n@advice {List.any} (arg::[a]) =
     \x . if arg == [] then x
                       else (proceed arg) x in
reverse x accum = reverse (tail x)
                  (cons (head x) accum) in
reverse [1,2] []
```

After the type inference of advice `n` and function `reverse`, we get the following result (we omit the irrelevant translation part for the moment). We write t_r as an abbreviation of $[a] \rightarrow [a] \rightarrow [a]$.

$\Gamma = \{$ `n` $: \forall ab.[a] \rightarrow b \rightarrow b,$ `reverse` $:_* \forall a.(reverse : t_r).t_r\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{looping}{\Gamma \vdash \texttt{reverse} : [Int] \rightarrow [Int] \rightarrow [Int]}\ (\textsc{Rel}) \quad ...}{\Gamma \vdash \texttt{reverse} : [Int] \rightarrow [Int] \rightarrow [Int]}\ (\textsc{Rel}) \quad ...}{\Gamma \vdash (\texttt{reverse}\ [1,2]) : [Int] \rightarrow [Int]}\ (\textsc{App})}{\Gamma \vdash (\texttt{reverse}\ [1,2]\ []) : [Int]}\ (\textsc{App})$$

The above derivation clearly shows that rule (REL) will repeatedly apply on the same judgement when an advised type has a predicate that is the same as the base type.

Our solution is to break the loop by devising a different releasing rule for recursive functions which predicate on themselves.

$$(\textsc{Rel-F}) \quad \frac{\Gamma \vdash f : (f : t).\rho \rightsquigarrow e' \quad F\ fresh \quad f \in A}{\Gamma \vdash f : \rho \rightsquigarrow \mathsf{let}\ F = (e'\ F)\ \mathsf{in}\ F}$$

Rule (REL-F) uses a fixed point combinator as the translation result. Note that it only releases the recursive predicate $(f : t)$. Should there be any predicates of other functions, rule (REL) is applied. As a result, the main expression in the above program is translated to

```
let F = \y.<reverse y,{n}> F
in F [1,2] []
```

3.2 Translating Chain Expressions

After the process of type inference and type-directed weaving, programs of AspectFun will be transformed into an intermediate form which is essentially a sugared lambda calculus with a special construct of *chain expressions*. In the subsequent step, our compilation model will conduct a syntactic transformation to expand the chain expressions and convert the input program to a Haskell program.

Since the specific execution trace of a chain expression depends mainly on the use of the special keyword, `proceed`, inside the chained advices, the key task of our transformation is to properly handle the occurrences of *proceed*. As stated earlier, any occurrences of proceed inside an advice should be bound to a function that represents the rest of computation (i.e., continuation). Hence the transformation is designed to realize this requirement. In practice, the transformation consists of two steps. The first step concerns the advices in a program. It adds an additional parameter called proceed to all advices, namely lambda abstracting `proceed`. The second step converts any chain expression in a program into a form of function application in continuation passing style. Basically, the conversion it performs can be defined inductively as follows.

```
<f, {}> = f
<f, {n1, n2, ..., nk}> = (n1 <f, {n2, ..., nk}>)
```

For example, the translation of Example 3b presented earlier will be converted to the following form.

```
n = \proceed.\arg. ...//fully trusted execution
n1 df = \proceed.\arg. df arg in
h x y = x y in
h (n1 (n f) w) 1 //was h <w,{n1 <f,{n}>}> 1
```

Admittedly, the chain expansion step is rather straightforward. One may suggest that the step should be integrated into the weaving step, thus eliminating the need of generating programs in the intermediate form. However, we argue that a staged translation process with chain expression as an intermediate form opens a wide scope of opportunities for optimizing the translated code. For instance, it is obvious that some advices will never invoke `proceed`. For these advices, all other advices chained after any of them are considered dead code and should be eliminated. We can therefore prune such chains by performing *dead-code elimination* analysis on the woven code. In the next section, we show yet another optimization of control-flow based pointcuts which take advantage of the explicit intermediate form.

4. Compiling Control-Flow Based Pointcuts

In this section, we present our compilation model for composite pointcuts – control-flow based pointcuts. Despite the fact that control-flow information are only available fully during run-time, we strive to discover as much information as possible during compilation. In particular, we transform type scopes within such pointcuts and then compile these type scopes away using our static type-directed weaver. When a pointcut designator depends on the dynamic state of the join point, we insert dynamic test to capture such dependency. These dynamic tests are implemented in a state-based fashion without the need to maintain call stacks, and is similar to that used in AspectJ as well as that used by Masuhara et al. [10]. We also consider the strategy to eliminate such tests at compile time. Our compilation process for composite pointcuts thus involves three steps:

1. Pre-processing source code to eliminate the use of type-scoped control flow (eg. `cflow(f(_ :: Int))`) and `cflow`.
2. Installing state-based mechanism in woven code.
3. Analyzing and optimizing woven code produced at step 2 to compile away as many dynamic tests as possible.

We shall describe these steps in more detail in the rest of this section.

4.1 De-sugaring

The objective of this step is to transform the source language into one that is amenable to static type inference and weaving. Specifically, type-scoped control flow (eg. `cflow(f(_ :: Int))`) and `cflow` can be considered syntactic sugar in our source language. They are therefore translated away before we conduct static analysis on the source code.

Type-scoped control-flow based pointcuts can be replaced by ones without type scopes. For instance,

```
n@advice around {k + cflow(f(_::Int))} (arg) = ...
```

is translated into

```
n'@advice around {f} (arg :: Int) = proceed arg
n@advice around {k + cflow(n')} (arg) = ...
```

Note that `cflow(f(_::Int))` has been translated into `cflow(n')`, where `n'` is a newly defined type-scoped advice on `f` which simply passes the argument to proceed. As a language design decision, we only allow the introduction of advice name as argument to `cflow` as part of compiler internal; it is not part of the source language.

In addition, we can translate all `cflow`-pointcuts into pointcuts involving `cflowbelow`. Doing so reduces the number of cases to

be considered during compilation. The translation rules for `cflow` translation are listed below. They are applied repetitively on pointcuts until there is no more change. The notation $+o$ refers to *other* pointcuts which are not the target of current iteration of translation.

Original	Translated
$f + \texttt{cflow}(f)+o$	$f+o$
$f + \texttt{cflow}(g)+o$	$f + \texttt{cflowbelow}(g)+o$ when $f \neq g$
$\texttt{any} + \texttt{cflow}(f)+o$	$\texttt{any} + \texttt{cflowbelow}(f)+o$ **and** $f+o$
$f - \texttt{cflow}(f)+o$	FALSE
$f - \texttt{cflow}(g)+o$	$f - \texttt{cflowbelow}(g)+o$ when $f \neq g$
$\texttt{any} - \texttt{cflow}(f)+o$	$\texttt{any}\backslash[f] - \texttt{cflowbelow}(f)+o$

Note that the pointcut $\texttt{any} + \texttt{cflow}(f)+o$ is translated to two pointcuts: $\texttt{any} + \texttt{cflowbelow}(f)+o$ and $f+o$. Also, the pointcut $f - \texttt{cflow}(f)+o$ does not refer to any feasible join points, and will be omitted from the translated code.

4.2 State-based Implementation

Information pertaining to `cflowbelow` pointcuts is ignored during static weaving. It is instead captured in a data structure, called `IFAdvice`, which is then used in the latter stages of compilation. An example of a woven code after static weaving is show here (in pseudo-code format):

Example 5

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1 in
g x = <k, {n}> x in
f x = if x == 0 then g x else <k, {n}> x in
(f 0, f 1)
```

□

This first line in the code above displays a meta-data structure capturing the association of the advice `n` with the `cflowbelow` pointcut `k+cflowbelow(g)`. This implies that dynamic testing is needed at call to function `k` to determine if `n` should be invoked; ie., if `k` is called in the context of a call to `g`. We call `g` the `cflowbelow` *advised function*.

In general, in order to enable matching of `cflowbelow` pointcuts dynamically, we maintain a global state of function invocations, and insert state-update and state-lookup operations at proper places in the woven code. Specifically, the encoding is done at two kind of locations: At the definitions of `cflowbelow` advised functions and at the uses of `cflowbelow` advices.

At the definition of a `cflowbelow` advised function, such as `g` in Example 5, we set up a *global state* to record the entry into and exit from the advised function. These are encoded in the body of the advised function. In the spirit of pure functional language, we implement this encoding using a *reader monad* [6]. In pseudo-code format, the encoding of `g` will be as follows:[3]

```
g x = enter "g";
      <k, n> x;
      restore state
```

Here, `enter` records into the global state the entry into function `g`, and `restore` erases this record from the global state.

Next, uses of `cflowbelow` advices appear in various chain expressions, such as `<k,n>` occurring in two places in Example 5. For these uses, we insert code to encode the lookup for the presence of the respective pointcuts in the global state. The encoding is a form of *guarded expression* denoted by `<|` $guard$, `n |>`. Semantically, the advice `n` will be executed only if $guard$ evaluates to `True`. The translated (pseudo) code for Example 5 is as follows:

Example 5a

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1  in
g x = enter "g";
      <k, { <| isIn "g", n|> } > x;
      restore state  in
f x = if x == 0
        then g x
        else <k, { <| isIn "g", n |> } > x in
(f 0, f 1)
```

□

The guard (`isIn "g"`) determines if `g` has been invoked and not yet returned. If so, advice `n` is executed. In this case, `n` is not triggered when evaluating `f 1`, but it is when evaluating `f 0`.

4.3 Control-Flow Pointcut Analysis and Optimization

From Example 5a, we note that the guard occurring in the definition of `g` is always true, and can thus be eliminated. Similarly, the guard occurring in the definition of `f` is always false, and the associated advice `n` can be removed from the code. Indeed, many of such guards can be eliminated during compile time, thus speeding up the execution of the woven code.

We share the sentiment with Avgustino et al. [1] that such optimization and its associated analysis can be more effectively performed on the woven code. In our system, we employ two interprocedural analysis to determine the opportunity for optimizing guarded expressions. They are **mayCflow** and **mustCflow** analysis (cf. [1]).

Since the subject language is polymorphically typed and higher-order, we adopt *annotated-type and effect* systems for our analysis. This approach has been described in detail in [13]. Judgments for both **mayCflow** and **mustCflow** analysis are of the form

$$\hat{\Gamma} \vdash e : \hat{\tau}_1 \xrightarrow{\varphi'} \hat{\tau}_2 \;\&\; \varphi$$

For **mayCflow** analysis (resp. **mustCflow** analysis), this means that under an annotated-type environment $\hat{\Gamma}$, an expression e has an annotated type $\hat{\tau}_1 \xrightarrow{\varphi'} \hat{\tau}_2$ and a context φ comprising the names of those functions which may be (resp. must be) invoked and not yet returned during the execution of e. The annotation φ' above the arrow $\rightarrow$ is the context in which the function e will be invoked. It is the union (resp. intersection) of all possible invocation contexts of e. Thus, φ and φ' both represent context, but the former captures all contexts in which e is evaluated, whereas the latter captures those in which e is invoked.

4.3.1 Lazy Semantics

The lazy semantics of AspectFun may appear to entail a different analysis than those with strict semantics. A plausible argument for this is that calls are only invoked on demand. Consider the following code:

```
// meta-data: IFAdvice [f+cflowbelow(g)] (n,...)
n proceed arg = ... in
f x = x in
g x = x + 1 in
g (f 3)
```

[3] This technique does not work satisfactorily when the cflow-advised functions are built-in functions, and will require additional function wrapping. We shall omit the detail in this paper.

$$\text{(CONST)}\quad \hat{\Gamma} \vdash_{\texttt{may}} c : \hat{\tau}_c \,\&\, \varphi \qquad \text{(VAR)}\quad \hat{\Gamma} \vdash_{\texttt{may}} x : \Gamma(x) \,\&\, \varphi$$

$$\text{(LAMBDA)}\quad \frac{\hat{\Gamma}[x \mapsto \hat{\tau}_x] \vdash_{\texttt{may}} e : \hat{\tau}_e \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \lambda x.e : \hat{\tau}_x \xrightarrow{\varphi'} \hat{\tau}_e \,\&\, \varphi} \qquad \text{(IF)}\quad \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \mathit{Bool} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e_1 : \hat{\tau} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e_2 : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \hat{\tau} \,\&\, \varphi}$$

$$\text{(LET)}\quad \frac{\hat{\Gamma}[x \mapsto \hat{\tau}_x] \vdash_{\texttt{may}} e_1 : \hat{\tau}_1 \,\&\, \varphi \quad \hat{\Gamma}[x \mapsto \hat{\tau}_x] \vdash_{\texttt{may}} e_2 : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \hat{\tau} \,\&\, \varphi} \qquad \text{(APP)}\quad \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \hat{\tau}_2 \xrightarrow{\varphi_F} \hat{\tau} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e_2 : \hat{\tau}_2 \,\&\, \varphi \quad \varphi \subseteq \varphi_F}{\hat{\Gamma} \vdash_{\texttt{may}} (e_1\ e_2) : \hat{\tau} \,\&\, \varphi}$$

$$\text{(DECL)}\quad \frac{\hat{\Gamma}[f \mapsto \hat{\tau}_x \xrightarrow{\varphi_F} \hat{\tau}][x \mapsto \hat{\tau}_x] \vdash_{\texttt{may}} e_f : \hat{\tau}' \,\&\, \varphi_F \cup \{\mathrm{f}\} \quad \hat{\Gamma}[f \mapsto \hat{\tau}_x \xrightarrow{\varphi_F} \hat{\tau}] \vdash_{\texttt{may}} p : \hat{\tau} \,\&\, \emptyset}{\hat{\Gamma} \vdash_{\texttt{may}} f\ x = e\ \mathsf{in}\ p : \hat{\tau} \,\&\, \emptyset}$$

$$\text{(CHAIN)}\quad \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1(e_2(\ldots(e_n\ e)\ldots)) : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \langle e, \{e_1, \ldots, e_n\}\rangle : \hat{\tau} \,\&\, \varphi} \qquad \text{(GUARDED)}\quad \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \mathit{Bool} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} < |e_1, e| > : \hat{\tau} \,\&\, \varphi}$$

$$\text{(SUB)}\quad \frac{\hat{\Gamma} \vdash_{\texttt{may}} e : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} e : \hat{\tau}' \,\&\, \varphi} \quad \text{if } \hat{\tau} \leq \hat{\tau}'$$

Figure 6. mayCflow inference rules

Under the lazy semantics, `(f 3)` will be executed within the body of `g`. This gives the impression that `f` is called within the calling context of `g`. Thence, advice `n` will be triggered at the `f`-call.

However, upon closer examination, we find this argument fallacious. Specifically, during the evaluation of `g (f 3)`, the subexpression `(f 3)` is first converted into a *thunk*, which captures the current calling context to be used for future evaluation. This calling context, which is the true context in which `f`-call is evaluated, does *not* contain `g`. As such, `n` will not be triggered.

In summary, while lazy semantics delays the execution of a call until it is needed, it does not induce a different calling context for the call from its strict semantics counterpart. Therefore, our control-flow pointcut analysis are oblivious to the call semantics of the language.

4.3.2 The Analysis and Optimization Details

Figure 6 presents our type-and-effect system for **mayCflow** analysis. Subtyping of annotated type is defined as

$$\hat{\tau} \leq \hat{\tau} \qquad \frac{\hat{\tau}_1' \leq \hat{\tau}_1 \quad \hat{\tau}_2 \leq \hat{\tau}_2' \quad \varphi' \subseteq \varphi}{\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \leq \hat{\tau}_1' \xrightarrow{\varphi'} \hat{\tau}_2'}$$

The above rule means that a function f of the LHS type can replace another function f' of the RHS type if:

1. f accepts all arguments that f' can accept ($\hat{\tau}_1' \leq \hat{\tau}_1$),
2. Results produced by f can be used in the context of f' ($\hat{\tau}_2 \leq \hat{\tau}_2'$), and
3. f can be used in all the possible contexts of f', and possibly more ($\varphi' \subseteq \varphi$).

Note that the rules specified in figure 6 together yield a set of constraints over context variables φ. The least solution of the constraints is the one containing the most information.

Applying the analysis over the woven code given in Example 5a, we obtain the following contexts for the body of each of the functions:

$\varphi_k^{\texttt{may}} = \{f, g\}$	May-context for body of `k`
$\varphi_g^{\texttt{may}} = \{f\}$	May-context for body of `g`
$\varphi_f^{\texttt{may}} = \emptyset$	May-context for body of `f`

Since the type-and-effect system for **mustCflow** analysis is similar to that for **mayCflow** analysis, we omit the detail in this paper, but simply to point out the resulting contexts produced by performing **mustCflow** analysis over the same example:

$\varphi_k^{\texttt{must}} = \emptyset$	Must-context for body of `k`
$\varphi_g^{\texttt{must}} = \{f\}$	Must-context for body of `g`
$\varphi_f^{\texttt{must}} = \emptyset$	Must-context for body of `f`

After collecting all the **mayCflow** and **mustCflow** information, we perform optimization over the woven ocde by eliminating guarded expressions. The basic principles for optimization are:

> Given a guarded expression of the form `<| isIn` f, e `|>` occurring in a program:
>
> 1. If the **mayCflow** analysis yields a context $\varphi^{\texttt{may}}$ for the expression such that $f \notin \varphi^{\texttt{may}}$, then the guard always fails, and the guarded expression will be eliminated from the program.
> 2. If the **mustCflow** analysis yields a context $\varphi^{\texttt{must}}$ for the expression such that $f \in \varphi^{\texttt{must}}$, then the guard always succeeds, and the guarded expression will be replaced by the subexpression e.
>
> In both cases, the guarded expression is successfully eliminated.

Going back to Example 5a, we are thus able to eliminate all the guarded expressions, yielding the following woven code:

Example 5b

```
// meta-data: IFAdvice [k+cflowbelow(g)] (n,...)
n proceed arg = arg+123 in
k x = x + 1 in
g x = enter "g";
      <k, {n}> x;
      restore state  in
f x = if x == 0
        then g x
        else <k, {} > x in
(f 0, f 1)
```

□

The expression <k,{}> indicates that no advice is chained, and thus k will be called as usual.

As a final example, consider a program that uses higher order functions:

Example 6

```
// meta-data: IFAdvice [k+cflowbelow(f)] (n,...)
n proceed arg = proceed (arg + 1) in
f x = enter "f"; x 1 ; restore state in
g y = y 2 in
k z = z * 2 in
(f <k, {<| isIn "f", n|>}>,
 g <k, {<| isIn "f",n|>}>)
```

□

The resulting annotated type of k used as an argument to f is $\text{Int} \xrightarrow{\{f\}} \text{Int}$ in **mustCflow** analysis, making n to be statically woven on it. Furthermore, the one used as an argument of g has annotated type $\text{Int} \xrightarrow{\{g\}} \text{Int}$ in **mayCflow** analysis; this results in the full removal of the associated advice. The final code is thus:

```
n proceed arg = proceed (arg + 1) in
f x = enter "f"; x 1 ; restore state in
g y = y 2 in
k z = z * 2 in
(f <k,{n}>, g <k,{}>)
```

5. Related Work

5.1 Aspect-Oriented Languages

Recently, researchers in functional languages have started to study various issues of adding aspects to strongly typed functional languages. Two notable works in this area, AspectML [3, 2] and Aspectual Caml [11], have made many significant results in supporting polymorphic pointcuts and advices in strongly typed functional languages such as ML. While these works have introduced some expressive aspect mechanisms into the underlying functional languages, they have not successfully reconciled coherent and static weaving – two essential features of a compiler for a Aspect-Oriented functional language.

AspectML [3, 2] advocates first-class join points for constructing generic aspect libraries. In order to support non-parametric polymorphic advice, AspectML includes case-advices which are subsumed by our type-scoped advices. Its type system is a conservative extension to the Hindley-Milner type inference algorithm with a form of local type inference based on some required annotations. During execution, advices are looked-up through the labels and runtime type analysis are performed to handle the matching of type-scoped pointcuts. This complete dynamic mechanism gives additional expressiveness by allowing run-time advice introduction. However, many optimization opportunities are lost as advice application information is not present during compilation. Lastly, advices are anonymous in AspectML and apparently not intended to be the targets of advising, *i.e.* no second-order advices.

Aspectual Caml [11], on the other hand, carries out type inference on advices without consulting the types of the functions designated by the pointcuts. Similar to AspectML, it allows a restricted form of type-scoped advices. Static weaving is achieved by traversing type-annotated base program ASTs to insert advices at matched joint points. The types of the applied advices must be more general than those of the joint points, through which, type safety is guaranteed. This design has the advantage of clear separate compilation as aspects can be compiled completely independently from the base program. In our case, we value correctness and understandability of program more than the ease of compilation.

Aspectual Caml's syntactic approach also makes it easy to advise anonymous functions. However, for polymorphic functions invoked indirectly through aliases or functional arguments, this approach cannot achieve coherent weaving results. It is also not clear how to extend the syntactic weaving scheme to handle nested advices, second-order advices or control flow based pointcuts such as cflow.

The current work is a conservative extension of our previous work [19], where we developed a type-directed weaving strategy for functional languages featuring higher-order functions, curried pointcuts and overlapping type-scoped advices. *Around* advices are woven into the base program based on the underlying type context using a Hindley-Milner type inference system extended with advised types and source translation. Coherent translations are achieved without using any dynamic typing mechanisms. However, in that work, advices and functions are still kept in two completely different levels: advices can never invoke advised functions. Moreover, control-flow based pointcuts were absent from the language. All these shortcomings are fully addressed in this paper.

5.2 Type-Scoped Programming

Our type-directed translation was originally inspired by the dictionary translation of Haskell type classes [18]. A number of subsequent applications of it [9, 7] also share some similarities. However, the issues discussed in this paper are unique, which make our translation substantially different from the others.

There has been some recent effort in encoding core features of AO functional languages with Haskell type classes [17]. The encoding is light-weight and allows easy integration with existing advanced language features such as type classes and GADTs [14]. In that work, all candidate advices are piled up at function calls and correct advice chainings are done implicitly by the type class resolution. This approach does not allow AOP specific static optimizers to take advantage of the chaining information, which defies one of the main thrusts of our compilation model. Moreover, there is also no clue on how control-flow based pointcuts and second-order advices can be incorporated.

On the other dimension, Washburn and Weirich demonstrated type-directed programming in AspectML [20]. They showed aspects together with a run-time type check mechanism can be used as an alternative of type classes and even performs better in cases where type classes struggle.

5.3 Static Optimization

The implementation and optimization of AspectFun took inspirations from the AspectBench Compiler for AspectJ (ABC) [1]. ABC implemented a series of optimizations which significantly improved AspectJ's run-time performance. Despite having a similar aim, the differences between object-oriented and functional paradigms do not allow most existing techniques to be shared. For example, the concerns of *closures* and *inlining* can be more straightforwardly encoded with higher-order functions and function calls in AspectFun ; whereas the complex control flow of higher-order functional languages makes the cflow analysis much more challenging. As a result, our typed cflow analysis has little resemblance with the one in ABC which was based on call graphs of an imperative language.

It is also worth mentioning that even though a number of optimizations have been done for AspectFun , the main purpose of this paper is to present a compilation model which supports static weaving and optimization for a polymorphic functional language. We leave further enhancements and empirical results to future investigation.

In [10], Masuhara, Kiczales and Dutchyn proposed a compilation and optimization model for aspect-oriented programs. Their approach was employing partial evaluation to optimize an evaluator for aspect-oriented languages implemented in Scheme. The limited power of the partial evaluator makes their work differ from ours in at least three ways: 1. Dynamic execution pointcuts are not statically determined. 2. The dealing of type scopes relies on dynamic type testing. 3. There is no mention of ways to reduce dynamic cflow checks.

6. Conclusion and Future Work

Static typing, static and coherent weaving are our main concerns in constructing a compilation model for functional languages with higher-order functions and parametric polymorphism. As a sequel to our previous results, this paper has advanced our investigation in a variety of ways. Firstly, while the basic structure of our type system remains the same, the typing rules have been significantly refined and extended beyond the two-layered model of functions and advices. Consequently, advices and advice bodies can also be advised. Secondly, we have devised new typing and translation rules to handle the weaving of advices on recursive functions which are polymorphic. Thirdly, we seamlessly incorporated a wide range of control-flow based pointcuts into our model and implemented a number of novel static optimization techniques which took advantage of the static nature of our weaver.

Moving ahead, we shall continue this line of investigation in a few directions. Currently, the type system bans mutual recursion and circular around advice execution. It will be interesting to see how these limitations can be removed. Since one of the major advantages of static weaving is the ease of static analysis and optimization, we will investigate additional optimization techniques and conduct empirical experiments of performance gain.

On another frontier, we plan to explore applying our static weaving system to other language paradigms. Java 1.5 has been extend with parametric polymorphism by the introduction of *generics*. The following example is taken from [4]

```
class List<T extends Comparable<T>> {
  T[] contents; ...
  List<T> max(List<T> x) {
    // general code for general types
} }
```

This class implements a list with a method `max`. When the input is an Boolean list, we may want to use bit operations for a more efficient implementation. This can be done with a type-scoped aspect.

```
aspect BooleanMax {
  List<Boolean> around(List<Boolean> x): args(x) &&
    execution(List<Boolean>
                List<Boolean>.max(List<Boolean>)) {
    // special code for boolean arguments
} }
```

However, as mentioned in [4], the above aspect cannot be handled by their aspect language because the type-erasure semantics of Java prohibits any dynamic type test here. We speculate our type-directed weaving could be a key to the solution of the problem.

References

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[2] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: a polymorphic aspect-oriented functional programmming language. In *Proc. of ICFP'05*. ACM Press, September 2005.

[3] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006, to appear.

[4] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006, to appear.

[5] M. P. Jones. *Qualified Types: Theory and Practice*. D.phil. thesis, Oxford University, September 1992.

[6] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.

[7] Mark P. Jones. Exploring the design space for type-based implicit parameterization. Technical report, Oregon Graduate Institute of Science and Technology, 1999.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, pages 108–118, 2000.

[10] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, pages 46–60, 2003.

[11] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of ICFP'05*. ACM Press, September 2005.

[12] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[13] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[14] S. Peyton Jones, D.Vytiniotis, G. Washburn, and S. Weirich. Simple unification-based type inference for GADTs, 2005. Submitted to PLDI'06.

[15] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[16] Damien Sereni and Oege de Moor. Static analysis of aspects. In Mehmet Aksit, editor, *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 30–39. ACM Press, 2003.

[17] Martin Sulzmann and Meng Wang. Aspect-oriented programming with type classes. http://www.comp.nus.edu.sg/˜ sulzmann, 2006.

[18] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

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

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