A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages

Kung Chen[1], Shu-Chun Weng[2], Meng Wang[3],
Siau-Cheng Khoo[4], and Chung-Hsin Chen[1]

[1] National Chengchi University
[2] National Taiwan University
[3] Oxford University
[4] National University of Singapore

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

1 Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system[8]. In AOP, a program consists of many functional modules and some *aspects* that encapsulate the crosscutting concerns. An aspect provides two specifications: A *pointcut*, comprising a set of functions, designates when and where to crosscut other modules; and an *advice*, which is a piece of code, that will be executed when a pointcut is reached. The complete program behaviour is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. This is called *weaving* in AOP. Weaving results in the behaviour of those functional modules impacted by aspects being modified accordingly.

The effect of an aspect on a group of functions can be controlled by introducing *bounded scope* to the aspect. Specifically, when the AOP paradigm is supported by a strongly-type polymorphic functional language, such as Haskell or ML, it is natural to limit the effect of an aspect on a function through declaration of the *argument type*. For instance, the code shown in Figure 1 defines

three aspects named `n3`, `n4`, and `n5` respectively; it also defines a main/base program consisting of declarations of `f` and `h` and a main expression returning a triplet. These advices designate `h` as *pointcut*. They differ in the type constraints of their first arguments. While `n3` is triggered at all invocations of `h`, `n4` limits the scope of its impact through type scoping on its first argument; this is called a *type-scoped* advice. This means that execution of `n4` will be woven into only those invocations of `h` with arguments of list type. Lastly, the type-scoped advice `n5` will only be woven into those invocations of `h` with their arguments being strings.

Example 1.

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

Fig. 1. An Example of Aspect-oriented program written in AspectFun

As with other AOP, we use `proceed` as a special keyword which may be called inside the body of an *around* advice. It is bound to a function that represents "the rest of the computation at the advised function"; specifically, it enables the control to revert to the advised function (ie., `h`).

Using type-scoped aspects enable us to have customized, type-dependent tracing message. Note that *String* (a list of *Char*) is treated differently 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.

In the setting of strongly-type polymorphic functional languages, types are treated as compile-time entities. As their use in controlling advices can usually be determined at compile-time, it is desirable to perform *static weaving* of advices into base program at compile time to produce an integrated code without explicit declaration of aspects. As pointed out by Sereni and de Moor [13], the integrated woven code produced by static weaving can facilitate static analysis of aspect-oriented programs.

Despite its benefits, static weaving is never a trivial task, especially in the presence of type-scoped advices. Specifically, it is not always possible to deter-

mine *locally* at compile time if a particular advice should be woven. 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 inconsistent behavior of `h` at runtime, 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 do not address this issue of static and yet coherent weaving. In AspectML [4] (*a.k.a* PolyAML [3]), dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [10] takes a lexical approach which sacrifices coherence[1] for static weaving.

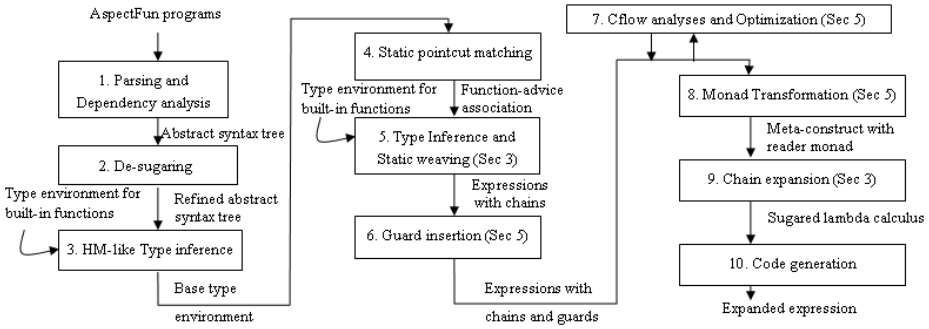

Fig. 2. Compilation Model for AspectFun

In this paper, we present a compilation model for AspectFun that ensures static and coherent weaving. AspectFun is an aspect-oriented polymorphically typed functional language with lazy semantics. The overall compilation process is illustrated in Figure 2. Briefly, the model comprises the following three major steps: (1) Static type inference of an aspect-oriented program; (2) Type-directed static weaving to convert advices to functions and produce a piece of woven code; (3) Type-directed optimization of the woven code. In contrast with our earlier work [15], this compilation model extends our research in three dimensions:

1. Language features: We have included a suite of features to our aspect-oriented functional language, AspectFun. Presented in this paper are: *second-*

[1] Our notion of coherence admits semantic equivalence among different invocations of a function with the same argument type. This is different from the coherence concept defined in qualified types [6] which states that different translations of an expression are semantically equivalent.

order advices, complex pointcuts such as `cflowbelow`, and an operational semantics for AspectFun.

2. Algorithms: We have extended our type inference and static weaving strategy to handle the language extension.[2] We have formulated the correctness of static weaving *wrt.* the operational semantics of AspectFun, and provided a strategy for analysing and optimizing the use of `cflowbelow` pointcuts.
3. Systems: We have provided a complete implementation of our compilation model turning aspect-oriented functional programs into executable Haskell code. [3]

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, the following result of static weaving is expressed using some meta-constructs:

```
n3 = \arg -> (proceed arg ; println "exiting from h") in
n4 = \arg -> (print "entering h with a list" ; proceed arg) in
n5 = \arg -> (print "entering h with " ; println arg; proceed arg) in
h x = x in
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. A *meta*-construct $\langle \_ , \{\ldots\} \rangle$, called *chain expression*, is used to express the chaining of advices and advised functions. For instance, $\langle \texttt{h} , \{\texttt{n3}, \texttt{n4}\} \rangle$ denotes the chaining of advices `n3` and `n4` 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.

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 highlights various Aspect-oriented features through AspectFun and defines its semantics. In Section 3, we describe our type inference system and the corresponding type-directed static weaving process. Next, we formulate the correctness of static weaving with respect to the semantics of AspectFun. In section 4, we provide a detailed description of how control-flow based pointcuts are handled in our compilation model. We discuss related work in Section 5, before concluding in Section 6.

2 AspectFun: The Aspect Language

We introduce an aspect-oriented lazy functional language, AspectFun, for our investigation. Figure 3 presents the language syntax. We write $\bar{o}$ as an abbrevia-

[2] Though not presented in this paper, we have devised a deterministic type-inference algorithm to determine the well-typedness of aspect-oriented programs.

[3] The prototype is available upon request.

tion for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and $\mathrm{fv}(o)$ as the free variables in o. We assume that $\bar{o}$ and o, when used together, denote unrelated objects. We write $t_1 \sim t_2$ to specify unification. We write $t \trianglerighteq t'$ iff there exists a substitution S over type variables in t such that $St = t'$, and we write $t \equiv t'$ iff $t \trianglerighteq t'$ and $t' \trianglerighteq t$. To simplify our presentation, complex syntax, such as `if` expressions and sequencings (`;`), are omitted even though they are used in examples.

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

Fig. 3. Syntax of the AspectFun Language

In AspectFun, top-level definitions include global variable and function definitions, as well as 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 any of the functions designated at the pointcut are about to execute. The act of triggering an advice during a function application is called *weaving*. Pointcuts are denoted by $\{\overline{pc}\}\ (arg)$, where *pc* stands for either a *primitive pointcut*, represented by *ppc*, or a *composite pointcut*. Pointcuts specify certain join points in the program flow for advising. Here, we focus on join points at function invocations. Thus a primitive pointcut, *ppc*, specifies a function or advice name the invocations of which, either directly or indirectly via functional arguments, will be advised.

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

A sequence of pointcuts, $\{\overline{pc}\}$, indicates the union of all the sets of join points selected by the pc_i's. The argument variable *arg* is bound to the actual argument of the named function call and it may contain a type scope. Alpha renaming is applied to local declarations beforehand so as to avoid name clash.

A composite pointcut relates the triggering of advice to the program's control flow. Specifically, we can write pointcuts which identify a subset of function invocations which occur in the dynamic context of other functions. For example,

the pointcut f + `cflowbelow`(g) selects those invocations of f which are made when the function g is still executing (i.e. invoked but not returned yet).[4] As an example, in the following code, there are four invocations of `fac`, and advice `n` will be triggered by all the `fac` invocations, except the first one (`fac 3`) due to the pointcut specification "`fac+cflowbelow(fac)`".

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

Similarly, a *type-scoped* control-flow based pointcut such as `(g+cflowbelow(f(_:t)))` limits the call context to those invocations of `f` with arguments of type `t`.

Composite pointcuts are handled separately in our compilation model through series of code transformation, analyses and optimizations. This is discussed in detail in Section 4.

In AspectFun, advice names can also be primitive pointcuts. As such, we allow advices to be developed to advice other advice. We refer to such advices as *second-order advices*. In contrast, the two-layered design of AspectJ like languages only allow advices to advise other advices in a very restricted way, thus a loss in expressivity [12].

The following code fragment shows a use of second-order advice to compute the total amount of a customer order and apply discount rates according to certain business rules.

Example 2.

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

In addition to the regular discount rules, ad-hoc sale discounts such as holiday-sales, anniversary sales etc., can be introduced through aspect declarations, thus achieving separation of concern. This is shown in the `n1` and `n2` declarations. Furthermore, 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 a business rule can be realized using a second-order aspect, as in `n3`. It calls `proceed` to compute the combined discount rate and ensures that the rate do not exceed 50%.

AspectFun is polymorphic and statically typed. Central to our approach is the construct of *advised types*, ρ in Figure 3, inspired by the *predicated types* [14] used in Haskell's type classes. These advised types augment common type schemes (as found in the Hindley-Milner type system) with *advice predicates*, $(f : t)$, which are used to capture the need of advice weaving based on type context. We shall explain them in detail in Section 3.

[4] The semantics of `cflowbelow` adheres to that provided in AspectJ. Conversion of the popularly `cflow` pointcuts to `cflowbelow` pointcuts is available in [2].

We end our description of the syntax of AspectFun by referring interested readers to the accompanied technical report [2] for detailed discussion of the complete features of AspectFun, which include "catch-all" pointcut any and its variants, a diversity of composite pointcuts, nested advices, as well as advices over curried functions.

Semantics of AspectFun As type information is required at the triggering of advices for weaving, the semantics of AspectFun is best defined in a language that allows dynamic manipulation of types: type abstractions and type applications. Thus, we convert AspectFun into a System-F like intermediate language, FIL.

$$
\begin{array}{lll}
\text{Program} & \pi^I & ::= (\overline{\text{Adv}}, e^I) \\
\text{Advice} & \text{Adv} & ::= (n : \varsigma, \overline{pc}, \tau, e^I) \\
\text{Join points} & jp & ::= f : \tau \mid \epsilon \\
\text{Expressions} & e^I & ::= v^I \mid x \mid \mathit{proceed} \mid e^I\ e^I \mid e^I\{\tau\} \mid \mathcal{LET}\ \ x\ =\ e^I\ \ \mathcal{IN}\ \ e^I \\
\text{Values} & v^I & ::= c \mid \lambda^{jp} x : \tau_x.\ e^I \mid \Lambda\alpha.\ e^I \\
\text{Types} & \tau & ::= \mathit{Int} \mid \mathit{Bool} \mid \alpha \mid \tau \to \tau \mid [\tau] \\
\text{Type schemes} & \varsigma & ::= \forall\overline{\alpha}.\ \tau \mid \tau
\end{array}
$$

Fig. 4. Syntax of FIL

FIL stores all the advices in a separated space leaving only function declarations and the main expression in the program. Expressions in FIL, denoted by e^I, are extensions of those in AspectFun to include annotated lambda ($\lambda^{jp} x : \tau_x.e^I$), type abstraction ($\Lambda\alpha.e^I$) and type application ($e^I\{\tau\}$) as listed in figure 4.

$$
(\overset{\text{PROG}}{\rightarrowtail})\ \frac{\emptyset \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}}{\pi \overset{\text{prog}}{\rightarrowtail} (\mathcal{A}, e^I)} \qquad (\text{Decl:MainExpr})\ \frac{\Delta \vdash e : \tau \rightarrowtail e^I}{\Delta \vdash_D e : \tau \rightarrowtail e^I; \emptyset}
$$

$$
(\text{Decl:Func})\ \frac{\begin{array}{c}\Delta.x : \tau_x \vdash e : \tau_f \rightarrowtail e_f^I \qquad \overline{\alpha} = \text{fv}(\tau_x \to \tau_f) \setminus \text{fv}(\Delta) \\ \Delta.f : \forall\overline{\alpha}.\ \tau_x \to \tau_f \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}\end{array}}{\Delta \vdash_D f\ x\ =\ e\ \text{in}\ \pi : \tau \rightarrowtail \mathcal{LET}\ \ f\ =\ \Lambda\overline{\alpha}.\ \lambda^{f:\tau_x \to \tau_f} x : \tau_x.\ e_f^I\ \ \mathcal{IN}\ \ e^I; \mathcal{A}}
$$

$$
(\text{Decl:Adv-An})\ \frac{\begin{array}{c}\text{fv}(t_x) : \text{fresh}(\text{fv}(t_x)) \vdash t_x \overset{\text{type}}{\rightarrowtail} \tau_x \\ \Delta.x : \tau_x.\mathit{proceed} : \tau_x \to \tau_n \vdash e : \tau_n \rightarrowtail e_n^I \\ \overline{\alpha} = \text{fv}(\tau_x \to \tau_n) \setminus \text{fv}(\Delta) \qquad \Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}\end{array}}{\begin{array}{r}\Delta \vdash_D n@\text{advice around}\ \{\overline{pc}\}\ (x :: t_x) = e\ \text{in}\ \pi : \tau \rightarrowtail e^I; \\ \mathcal{A}.(n : \forall\overline{\alpha}.\tau_x \to \tau_n, \overline{pc}, \tau_x, \Lambda\overline{\alpha}.\ \lambda^{n:\tau_x \to \tau_n} x : \tau_x.\ e_n^I)\end{array}}
$$

$$
(\text{Expr:Var})\ \frac{\tau = \Delta(x)}{\Delta \vdash x : \tau \rightarrowtail x} \qquad (\text{Expr:Ty-App})\ \frac{\forall\overline{\alpha}.\ \tau = \Delta(x) \quad \tau_x = [\overline{\tau'}/\overline{\alpha}]\tau}{\Delta \vdash x : \tau_x \rightarrowtail x\{\overline{\tau'}\}}
$$

$$
(\text{Type:Base})\ \sigma \vdash \mathit{Int} \overset{\text{type}}{\rightarrowtail} \mathit{Int} \qquad \sigma \vdash \mathit{Bool} \overset{\text{type}}{\rightarrowtail} \mathit{Bool} \qquad \sigma.a : \alpha \vdash a \overset{\text{type}}{\rightarrowtail} \alpha
$$

$$
(\text{Type:Inferred})\ \frac{\sigma \vdash t \overset{\text{type}}{\rightarrowtail} \tau}{\sigma \vdash [t] \overset{\text{type}}{\rightarrowtail} [\tau]} \qquad \frac{\sigma \vdash t_1 \overset{\text{type}}{\rightarrowtail} \tau_1 \quad \sigma \vdash t_2 \overset{\text{type}}{\rightarrowtail} \tau_2}{\sigma \vdash t_1 \to t_2 \overset{\text{type}}{\rightarrowtail} \tau_1 \to \tau_2}
$$

Fig. 5. Conversion Rules to FIL (interesting cases)

The conversion is led by rule $\pi \overset{\text{prog}}{\rightarrowtail} (\mathcal{A}, e^I)$. A type environment, also called conversion environment, Δ of the structure $\overline{x : \varsigma}$ is employed. We write the judgement $\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}$ to mean that an AspectFun program having type τ is converted to a FIL program, yielding an advice store $\mathcal{A} \in \overline{Adv}$. The judgement $\Delta \vdash e : \tau \rightarrowtail e^I$ asserts that an AspectFun *expression* e having a type τ under Δ is converted to a FIL expression e^I. The nontrivial conversion rules are listed in Figure 5. The full set of rules is available in [2].

Specifically, the rules (DECL:FUNC) and (DECL:ADV-AN) convert top-level function and advice declarations to ones having annotated lambda $\lambda^{f:\tau} x : \tau_x.e^I$; the annotation $\lambda^{(f:\tau)}$ highlights its jointpoint. The semantics of FIL uses these annotations to find the set of advices to be triggered. The conversion also introduces type abstraction $\Lambda\overline{\alpha}$ into the definition bodies. Rule (EXPR:TY-APP) instantiates type variables to concrete types.

Each advice in AspectFun is converted to a tuple in $\mathcal{A}$. The tuple contains the advice's name (n) with the advice's type (ς), the pointcuts the advice selects ($\overline{pc}$), the type-scope constraint on argument (τ), and the advice body (e^I).

Operational Semantics for FIL We describe the operational semantics for AspectFun in terms of that for FIL. Due to space limitation, we leave the semantics for handling cflow-based pointcut to [2].

Expressions:

$$\text{(OS:VALUE)} \quad c \Downarrow c \quad \lambda^{jp} x : \tau_x.\ e^I \Downarrow \lambda^{jp} x : \tau_x.\ e^I \quad \Lambda\alpha.\ e^I \Downarrow \Lambda\alpha.\ e^I$$

$$\text{(OS:APP)}\ \frac{e_1^I \Downarrow \lambda^{jp} x : \tau_x.\ e_3^I \quad \text{Trigger}(\lambda x : \tau_x.\ e_3^I, jp) = \lambda x : \tau_x.\ e_4^I \quad [e_2^I/x]e_4^I \Downarrow v^I}{e_1^I\ e_2^I \Downarrow v^I}$$

$$\text{(OS:TY-APP)}\ \frac{e_1^I \Downarrow \Lambda\alpha.\ e_2^I \quad [\tau/\alpha]e_2^I \Downarrow v^I}{e_1^I\{\tau\} \Downarrow v^I} \qquad \text{(OS:LET)}\ \frac{[e_1^I/x]e_2^I \Downarrow v^I}{\mathcal{LET}\ \ x\ =\ e_1^I\ \ \mathcal{IN}\ \ e_2^I \Downarrow v^I}$$

Auxiliary Functions:

$$\begin{array}{ll}
\text{Trigger} & :\ e^I \times jp \rightarrow e^I \\
\text{Trigger}(e^I, \epsilon) & = e^I \\
\text{Trigger}(\lambda x : \tau_x.\ e^I, f : \tau_f) & = \text{Weave}(\lambda x : \tau_x.\ e^I, \tau_f, \text{Choose}(f, \tau_x))
\end{array}$$

$$\begin{array}{ll}
\text{Weave} & :\ e^I \times \tau \times \overline{\text{Adv}} \rightarrow e^I \\
\text{Weave}(e^I, \tau_f, []) & = e^I \\
\text{Weave}(e_f^I, \tau_f, a : advs) & = \text{Let } (n : \forall\overline{\alpha}.\ \tau_n, \overline{pc}, \tau, \Lambda\overline{\alpha}.\ e^I) = a \\
& \quad \text{In}\ \ \text{If } \neg(\tau_n \trianglerighteq \tau_f) \text{ Then Weave}(e_f^I, \tau_f, advs) \\
& \qquad \text{Else Let } \overline{\tau} \text{ be types such that } [\overline{\tau}/\overline{\alpha}]\tau_n = \tau_f \\
& \qquad\quad (e_p^I,\ e_a^I) = (\text{Weave}(e_f^I, \tau_f, advs),\ (\Lambda\overline{\alpha}.\ e^I)\{\overline{\tau}\}) \\
& \qquad\quad \lambda^{n:\tau_n} x : \tau_x.\ e_n^I = [e_p^I/proceed]e_a^I \\
& \qquad\quad\ \ \text{In}\ \ \text{Trigger}(\lambda x : \tau_x.\ e_n^I, n : \tau_n)
\end{array}$$

$$\begin{array}{l}
\text{Choose}(f, \tau) = \{(n_i : \varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \mid (n_i : \varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \in \mathcal{A},\ \tau_i \trianglerighteq \tau, \\
\qquad\qquad\qquad\qquad \exists pc \in \overline{pc_i}\ s.t.\ \text{JPMatch}(f, pc)\} \\
\text{JPMatch}(f, pc) = (f \equiv pc)
\end{array}$$

Fig. 6. Operational Semantics for FIL

The reduction-based big-step operational semantics, written as $\Downarrow_{\mathcal{A}}$, is defined in Figure 6. Together with it are definitions of the auxiliary functions used. Note

that the advice store $\mathcal{A}$ is implicitly carried by all the rules, and it is omitted to avoid cluttering of symbols.

Triggering and weaving of advices are performed during function applications, as shown in rule (OS:App). Triggering operation first chooses eligible advices based on argument type, and weaves them into the function invocation – through a series of substitutions of advice bodies – for execution. Note that only those advices the types of which are instantiable to the applied function's type are selected for chaining via the Weave function.

3 Static Weaving

In our compilation model, aspects are woven statically (Step 5 in Figure 2). Specifically, we present in this section a type inference system which guarantees type safety and, at the same time, weaves the aspects through a type-directed translation. Note that, for composite pointcuts such as `f+cflowbelow(g)`, our static weaving system simply ignores the control-flow part and only considers the associated primitive pointcuts (ie., `f`). Treatment of control-flow based pointcuts is presented in Section 4.

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` possesses the advised type $\forall a.(h : a \to a).a \to a$, in which $(h : a \to a)$ 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* $a' \to a'$ *where* a' *is a fresh instantiation of type variable* a. We say a type t is more general than type t' iff $t \trianglerighteq t'$ but $t \not\equiv t'$. 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 $\forall a.a \to a$ since it does not contain any application of advised functions in its definition.

We begin with the following set of auxiliary functions that assists type inference:

$$(\textsc{Gen}) \quad gen(\Gamma, \sigma) = \forall \bar{a}.\sigma \quad \text{where } \bar{a} = \text{fv}(\sigma) \backslash \text{fv}(\Gamma) \qquad (\textsc{Card}) \quad |o_1 ... o_k| = k$$

The main set of type inference rules, as described in Figure 7, 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. Note that locally defined functions are not subject to being advised and not listed in 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}.

Expressions:

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

$$(\textsc{App})\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : t_1 \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : t_2 \rightsquigarrow (e_1'\ e_2')}\ (\textsc{Abs})\ \frac{\Gamma.x : t_1 \rightsquigarrow x \vdash e : t_2 \rightsquigarrow e'}{\Gamma \vdash \lambda x.e : t_1 \to t_2 \rightsquigarrow \lambda x.e'}$$

$$(\textsc{Let})\ \frac{\Gamma \vdash e_1 : \rho \rightsquigarrow e_1' \quad \sigma = gen(\Gamma, \rho) \qquad \Gamma.f : \sigma \rightsquigarrow f \vdash e_2 : t \rightsquigarrow e_2'}{\Gamma \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \rightsquigarrow \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}$$

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

Declarations:

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

$$(\textsc{Adv})\ \frac{\begin{array}{c} \Gamma.proceed : t_1 \to t_2 \vdash \lambda x.e_a : \bar{p}.t_1 \to t_2 \rightsquigarrow e_a' \quad f_i : \forall \bar{a}.t_i \in \Gamma_{base} \\ try(S = t_1 \trianglerighteq t_x) \quad S(t_1 \to t_2) \trianglerighteq t_i \\ \Gamma.n : \sigma \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t' \rightsquigarrow \pi' \quad \sigma = gen(\Gamma, S(\bar{p}.t_1 \to t_2)) \end{array}}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: \forall \bar{b}.t_x) = e_a \mathsf{in}\ \pi : t' \rightsquigarrow n = e_a'\ \mathsf{in}\ \pi'}$$

Fig. 7. Typing rules

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.

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 1. *Given a function f and its type $t_2 \to t_2'$, 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 vacuously 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 (the selection is done by $\{n_i | t_i \trianglerighteq t'\}$). All these selected advices have corresponding non-advised types guaranteed by the weavable condition. This ensures the bodies of the selected advices are correctly woven. Finally, the 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 exist 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, which 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*. This advice parameter enables concrete *advice-chained functions* to be passed in at a later stage, called *releasing*, through application of rule (REL). Specifically, 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.

Handling Advices Declarations define top-level bindings including advices. We use a judgement $\Gamma \vdash \pi : \sigma \rightsquigarrow \pi'$ which reassembles the one for expressions.

Rule (GLOBAL) is very similar to rule (LET) with the tiny difference that rule (GLOBAL) binds id which is not in A with :. It binds id with $:_*$ otherwise.

Rule (ADV) deals with advice declarations. We only consider type-scoped advices, and treat non-type-scoped ones as special cases having the most general type scope $\forall a.a$. We first infer a (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. Thus, as with functions, the advice is parameterized, and an advised type is assigned to it and only released when it is chained in rule (VAR-A).

Next, we check whether the inferred input type is more general than the type-scope: If so, the inferred type is specialized with the substitution S resulted from the matching; otherwise, the type-scope is simply ignored. The function *try* acts as an exception handler. It attempts to match two types: If the matching succeeds, a resulting substitution is assigned to S; otherwise, an empty substitution is returned. As a result, the inferred type t_1 is not strictly required to subsume the type scope t_x. On the other hand, the advice's type $S(t_1 \rightarrow t_2)$ is require to be 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}. Finally, 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.

Correctness of Static Weaving The correctness of static weaving is proven by relating it to the operational semantics of AspectFun. Due to space limitation, we refer readers to [2] for details.

Example We illustrate the application of rules in Figure 7 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:

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

$$\dfrac{\dfrac{\dfrac{h : t \to t \rightsquigarrow dh \in \Gamma_2}{\Gamma_2 \vdash h : t \to t \rightsquigarrow dh}\,(\textsc{Var}) \quad \dfrac{x : t \rightsquigarrow x \in \Gamma_2}{\Gamma_2 \vdash x : t \rightsquigarrow x}\,(\textsc{Var})}{\Gamma_2 = \Gamma_1, x : t \rightsquigarrow x \vdash (h\ x) : t \rightsquigarrow (dh\ x)}\,(\textsc{App})}{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)}\,(\textsc{Abs})}{\Gamma \vdash \lambda x.(h\ x) : (h : t \to t).t \to t \rightsquigarrow \lambda dh.\lambda x.(dh\ x)}\,(\textsc{Pred})$$

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

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

$$\dfrac{\dfrac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : [C] \to [C]).[C] \to [C] \rightsquigarrow f}\,(\textsc{Var}) \quad \dfrac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad \ldots}{\Gamma_3 \vdash h : [C] \to [C] \rightsquigarrow \langle h\ , \{n_3, n_4, n_5\}\rangle}\,(\textsc{Var-A})}{\Gamma_3 \vdash f : [C] \to [C] \rightsquigarrow (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle)}\,(\textsc{Rel}) \quad \ldots$$

$$\dfrac{\Gamma_3 \vdash f : [C] \to [C] \rightsquigarrow (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle)}{\Gamma_3 \vdash (f\ \text{“}c\text{”}) : [Char] \rightsquigarrow (f\ \langle h\ , \{n_3, n_4, n_5\}\rangle\ \text{“}c\text{”})}\,(\textsc{App})$$

We note that rules (Abs),(Let) and (App) are rather standard. Rule (Let) only binds f with : which signalizes locally defined functions are not subject to advising.

Final Translation and Chain Expansions The last step of AspectFun compilation is to expand meta-constructs produced after static weaving, such as chain-expressions, to standard expressions in AspectFun, which are called *expanded expressions*. It is in fact seperated into two steps: *addProceed* and *chain expansion*. AddProceed turns the keyword proceed into a parameter of all advices. Expansion of meta-construct (chains) is defined (partly) below by an expansion operator $[\![\cdot]\!]$. It is applied compositionally on expressions, with the help of an auxiliary function ProceedApply to substitute proper function as the proceed parameter. Moreover, ProceedApply also handles expansion of second-order advices.

e_M : Expressions containing meta-constructs
addProceed : $e_M \longrightarrow e_M$
addProceed(let n *df* $arg\ =\ e_1$ in e_2) = *if* (n is an advice) *then*
    let n *df proceed* $arg\ =\ e_1$
    in addProceed(e_2)
  else let n *df* $arg\ =\ e_1$ in addProceed(e_2)
addProceed(e) = e

$$
\begin{array}{ll}
[\![\cdot]\!] & : e_M \longrightarrow \text{Expanded expression} \\
[\![e_1\ e_2]\!] & = [\![e_1]\!]\ [\![e_2]\!] \quad \textit{... trivial rules omitted} \\
[\![\langle f\ \overline{e}\ , \{\}\rangle]\!] & = [\![f\ \overline{e}]\!] \\
[\![\langle f\ \overline{e}\ , \{e_a, \overline{e_{advs}}\}\rangle]\!] & = \text{ProceedApply}(e_a, \langle f\ \overline{e}\ , \{\overline{e_{advs}}\}\rangle)
\end{array}
$$

$$
\begin{array}{ll}
\text{ProceedApply}(n\ \overline{e}, k) & = [\![n\ \overline{e}\ k]\!] \quad \text{if rank}(n) = 0 \\
\text{ProceedApply}(\langle n\ \overline{e}\ , \{\overline{ns}\}\rangle, k) & = [\![\langle n\ \overline{e}\ k\ , \{\overline{ns}\}\rangle]\!] \quad \text{if rank}(n) > 0
\end{array}
$$

$$
\text{rank}(x) = \begin{cases} 1 + \max_i \text{rank}(e_{ai}) & \text{if } x \equiv \langle f\ \overline{e}\ , \{\overline{e_a}\}\rangle \\ 0 & \text{otherwise} \end{cases}
$$

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. Our strategy is as follows: In the early stage of the compilation process (step 2 in Figure 2), we convert all control-flow based pointcuts in the source to pointcuts involving only `cflowbelow`[2]. For example,

```
m@advice around {h+cflowbelow(d(_::Int))} (arg) = ...
```

will be translated, via introduction of second-order advice, into the following:

```
m'@advice around {d} (arg :: Int) = proceed arg in
m@advice around {h+cflowbelow(m')} (arg) = ...
```

Next, the advice `m` will be further translated to

```
m@advice around {h} (arg) = ...
```

while the association of `h+cflowbelow(m')` and `m` will be remembered for future use.

After the static weaving and *addProceed* step, we reinstall the control-flow based pointcuts in the woven code through guard insertion and monad transformation (steps 6 and 8 in Figure 2), following the semantics of control-flow based pointcuts, and then subject the woven code to control-flow pointcut analysis and code optimization. The description of these steps will be presented after explaining the extension made to the FIL semantics.

Semantics of control-flow based pointcuts The semantics of control-flow based pointcuts is defined by modifying the operational semantics for FIL introduced in section 2.

Specifically, we modify the operational semantics function $\Downarrow_{\mathcal{A}}$, defined in Figure 6, to carry a *stack* $\mathcal{S}$, written as $\Downarrow^{\mathcal{S}}_{\mathcal{A}}$, denoting that the progress is done under a stack environment $\mathcal{S}$. $\mathcal{S}$ is a stack of function names capturing the stack of nested calls that have been invoked but not returned at the point of reduction.

By replacing $\Downarrow$ by $\Downarrow^{\mathcal{S}}$, most of the rules remain unchanged except rules (OS:App) and (OS:Let), which are refined with the introduction of $(\!|e, S|\!)$:

$$\text{(OS:App')}\ \frac{e_1^I \Downarrow^{\mathcal{S}} \lambda^{f:\tau_f} x:\tau_x.\ e_3^I \qquad \text{Trigger}'(\lambda^f x:\tau_x.\ e_3^I, f:\tau_f, \mathcal{S}) = \lambda^g x:\tau_x.\ e_4^I \qquad \mathcal{S}' = cons(g,\mathcal{S}) \qquad [(\!|e_2^I, \mathcal{S}|\!)/x]e_4^I \Downarrow^{\mathcal{S}'} v^I}{e_1^I\ e_2^I \Downarrow_{\mathcal{A}}^{\mathcal{S}} v^I}$$

$$\text{(OS:Let')}\ \frac{[(\!|e_1^I, \mathcal{S}|\!)/x]e_2^I \Downarrow^{\mathcal{S}} v^I}{\mathcal{LET}\ \ x\ =\ e_1^I\ \ \mathcal{IN}\ \ e_2^I \Downarrow^{\mathcal{S}} v^I} \qquad \text{(OS:Clos)}\ \frac{e^I \Downarrow^{\mathcal{S}} v^I}{(\!|e^I, \mathcal{S}|\!) \Downarrow^{\mathcal{S}'} v^I}$$

$(\!|e, \mathcal{S}|\!)$ is a *stack closure*, meaning that e should be evaluated under stack $\mathcal{S}$ ignoring current stack, since we adopt lazy semantics for AspectFun. Detailed discussion of the modification can be found in [2].

State-based implementation As stated above, the only control-flow based pointcut to implement is the `cflowbelow` pointcut. We use an example to illustrate our implementation scheme. The following is part of a woven code after static weaving.

Example 3.
```
// 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 (comment) line in the code above indicates that advice `n` is associated with the pointcut `k+cflowbelow(g)`. Hence, `n` should be triggered at a call to `k` *only if* the `k`-call is made in the context of a `g`'s invocation. We call `g` the `cflowbelow` *advised function*.

In order to support the dynamic nature of the `cflowbelow` pointcut efficiently, our implementation maintains a *global state* of function invocations, and inserts state-update and state-lookup operations at proper places in the woven code. Specifically, the insertion is done at two kinds of locations: At the definitions of `cflowbelow` advised functions, `g` here, and at the uses of `cflowbelow` advices.

For a `cflowbelow` advised function definition, we encode the updating of the global state – to record the entry into and the exit from the function – in the function body. In the spirit of pure functional language, we implement this encoding using a *reader monad* [7]. In pseudo-code format, the encoding of `g` in Example 3 will be as follows:[5]

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

Here, `enter "g"` adds an entry record into the global state, and `restore_state` erases it.

Next, for each use occurrence of `cflowbelow` advices, we wrap it with a state-lookup to determine the presence of the respective pointcuts. The wrapped code is a form of *guarded expression* denoted by `<|`*guard*`,n|>` for advice `n`. It implies

[5] Further mechanism is required when the `cflowbelow` advised function is a built-in function. The detail is omitted here.

that `n` will be executed *only if* the *guard* evaluates to `True`. The Example 3 with wrapped code appears as follows:

Example 3a

```
// 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`.

Control-Flow Pointcut Analysis and Optimization From Example 3a, 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 thus employ two interprocedural analyses to determine the opportunity for optimizing guarded expressions. They are **mayCflow** and **mustCflow** analyses (cf. [1]).

Since the subject language is polymorphically typed and higher-order, we adopt *annotated-type and effect* systems for our analysis (cf. [11]). We define a context φ to be a set of function names. Judgments for both **mayCflow** and **mustCflow** analyses 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 capturing the name 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 resulted from evaluation of e will be invoked.

This type-and-effect approach has been described in detail in [11]. As our analyses follow this approach closely, we omit the detail here for space limitation, and refer readers to [2] for explanation. Applying both **mayCflow** and **mustCflow** analyses over the woven code given in Example 3a, we obtain the following contexts for the body of each of the functions:

$$\begin{array}{lll} \varphi_k^{\texttt{may}} = \{f, g\}, & \varphi_g^{\texttt{may}} = \{f\}, & \varphi_f^{\texttt{may}} = \emptyset \\ \varphi_k^{\texttt{must}} = \emptyset, & \varphi_g^{\texttt{must}} = \{f\}, & \varphi_f^{\texttt{must}} = \emptyset \end{array}$$

The result of these analyses will be used to eliminate guarded expressions in the woven code. The basic principles for optimization are:

Given a guarded expression e_{gd} of the form `<| isIn` f, e `|>`:

1. If the **mayCflow** analysis yields a context $\varphi^{\texttt{may}}$ for e_{gd} *st.* $f \notin \varphi^{\texttt{may}}$, then the guard always fails, and e_{gd} will be eliminated.
2. If the **mustCflow** analysis yields a context $\varphi^{\texttt{must}}$ for e_{gd} *st.* $f \in \varphi^{\texttt{must}}$, then the guard always succeeds, and e_{gd} will be replaced by the subexpression e.

Going back to Example 3a, we are thus able to eliminate all the guarded expressions, yielding the following woven code:

```
// 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; thus `k` will be called as usual.

5 Related Work

AspectML [4, 3] and Aspectual Caml [10] are two other endeavors to support polymorphic pointcuts and advices in a statically typed functional language. While they 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 an aspect-oriented functional language.

AspectML [4, 3] advocates first-class join points and employs the `case-advice` mechanism to support type-scoped pointcuts based on runtime type analysis. It enables programmers to reify calling contexts and change advice behavior based on the context information found therein, thus achieving cflow based advising. Such dynamic mechanisms gives AspectML additional expressiveness not found in other works. However, many optimization opportunities are lost as advice application information is not present during compilation.

Aspectual Caml [10] takes a lexical approach to static weaving. Its weaver traverses 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, thus guaranteeing type safety. Unfortunately, the technique fails to support coherent weaving of polymorphic functions which are invoked indirectly. Moreover, there is no formal description of the type inference rules, static weaving algorithm, or operational semantics.

The implementation and optimization of AspectFun took inspirations from the AspectBench Compiler for AspectJ (ABC) [1]. Despite having a similar aim, the differences between object-oriented and functional paradigms do not allow most existing techniques to be shared. 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.

In [9], Masuhara et al. proposed a compilation and optimization model for aspect-oriented programs. As their approach employs partial evaluation to optimize a dynamic weaver implemented in Scheme, the amount of optimization is restricted by the ability of the partial evaluator. In contrast, our compilation model is built upon a static weaving framework; residues are only inserted when it is absolutely necessary (in case of some control-flow based pointcuts), which keeps the dynamic impact of weaving to a minimum.

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 work, this paper has made the following significant progress. Firstly, while the basic structure of our type system remains the same, the typing and translation 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 proved the soundness of our static weaving with respect to an operational semantics for the underlying language, AspectFun. Thirdly, we seamlessly incorporated a wide range of control-flow based pointcuts into our model and implemented some novel optimization techniques which take advantage of the static nature of our weaver. Lastly, we developed a compiler which follows our model to translate AspectFun programs into executable Haskell code.

Moving ahead, we will investigate additional optimization techniques and conduct empirical experiments of performance gain. Besides, we plan to explore ways of applying our static weaving system to other language paradigms. In particular, Java 1.5 has been extend with parametric polymorphism by the introduction of *generics*. Yet, as mentioned in [5], the type-erasure semantics of Java prohibits the use of dynamic type tests to handle type-scoped advices. We speculate our static weaving scheme could be a key to the solution of the problem.

7 Acknowledgment

We would like to thank the anonymous referees for their insightful comments. This research is partially supported by the National University of Singapore under research grant "R-252-000-250-112", and by the National Science Council, Taiwan, R.O.C. under grant number "NSC 95-2221-E-004-004-MY2".

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. Kung Chen, Shu-Chun Weng, Meng Wang, Siau-Cheng Khoo, and Chung-Hsin Chen. A compilation model for AspectFun. Technical report, TR-03-07, National Chengchi University, Taiwan, March 2007. http://www.cs.nccu.edu.tw/~chenk/AspectFun/AspectFun-TR.pdf.
3. 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.
4. 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.
5. Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006, to appear.
6. M. P. Jones. *Qualified Types: Theory and Practice*. D.phil. thesis, Oxford University, September 1992.
7. Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.
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. Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, pages 46–60, 2003.
10. Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of ICFP'05*. ACM Press, September 2005.
11. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
12. 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.
13. 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.
14. 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.
15. 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.