行政院國家科學委員會專題研究計畫 成果報告

以型態導向方法發展多型剖面的織入技術與應用(第 2 年)研究成果報告(完整版)

計 畫 類 別 ：個別型
計 畫 編 號 ：NSC 95-2221-E-004-004-MY2
執 行 期 間 ：96 年 08 月 01 日至 97 年 07 月 31 日
執 行 單 位 ：國立政治大學資訊科學系

計 畫 主 持 人：陳恭

計畫參與人員：碩士班研究生-兼任助理人員：陳忠信
碩士班研究生-兼任助理人員：陳鉅秉
碩士班研究生-兼任助理人員：林佳瑩
大專生-兼任助理人員：翁書鈞

報 告 附 件 ：出席國際會議研究心得報告及發表論文

處 理 方 式 ：本計畫可公開查詢

中 華 民 國 97 年 10 月 30 日

行政院國家科學委員會補助專題研究計畫 ■ 成 果 報 告 □期中進度報告

以型態導向方法發展多型剖面的織入技術與應用

A Type-Directed Approach to Developing Static Weaving Techniques for Polymorphic Aspects and Its Applications

計畫類別：■ 個別型計畫　　□ 整合型計畫
計畫編號：NSC 95－2221－E—004－004－MY2
執行期間： 95 年 8 月 1 日至　97 年 7 月 31 日

計畫主持人：陳 恭
共同主持人：
計畫參與人員： 陳忠信，翁書鈞，林佳瑩，陳鉅秉

成果報告類型(依經費核定清單規定繳交)：□精簡報告　■完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
■出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立政治大學

中　華　民　國　97　年　10　月　30　日

研究計畫中英文摘要：

關鍵詞：剖面導向程式設計，多型剖面，織入，多型型態系統，型態推理

剖面導向程式設計(Aspect-Oriented Programming, AOP)是近年來繼物件導向程式設計(OOP)後，所興起的一種新的程式設計方法。從 AOP 的觀點來看，應用程式除了功能邏輯以外，還有許多像安全需求等的橫跨性關注(crosscutting concerns)；實現這些橫跨性關注的程式碼應該要從功能模組中分離出來，自成一模組並稱之為剖面。剖面與功能模組之間的界接點由所謂橫跨點(pointcut)來定義，並透過稱之為織入(weaving)的機制將剖面程式碼整合入功能模組中，從而合成完整程式，滿足系統整體需求。這樣實現橫跨性關注的程式碼就可以集中封裝於適當的模組中，避免掉程式碼糾結與重複的問題。

近幾年來關於剖面程式設計的各項研究蓬勃發展，從剖面程式語言、開發工具、剖面設計與分析，到剖面的理論基礎，都可見到了許多的研究成果陸續發表出來。這些研究成果大多集中在探討像 AspectJ 這類以物件導向為基礎的剖面語言的各個面向，因這類語言都不具備參數式多型(parametric polymorphism)的功能，所以關於多型剖面(Polymorphic aspects)的研究尚屬起步階段。不過去年推出的 Java 5 已提供多型(或稱泛型 Generics)的機制，所以我們認為實有必要針對多型剖面所帶來的技術與理論的挑戰加以探討，好替下一代的剖面語言奠定基礎。目前多型剖面的研究主要是以具備多型機制的函數式語言為基礎，雖然已經有了一些重要的結果，但仍然有許多可以改善加強的地方。舉例而言，如何適當整合高階函數與多型剖面就還是一個重要的研究課題。

本計劃以型態導向的方法發展一套多型剖面的織入技術，並將其應用到多型程式設計上。目前關於多型剖面的織入處理研究，雖然也會採用型態資訊，但是還是以語法結構為基礎，以致於不能處理像高階函數的間接呼叫，也無法解決多個形態有重疊的剖面的織入問題。我們提出所謂的諮詢型態(advised types)的概念，將剖面織入的需求內入型態之中，透過型態推理(type inference)的過程，選擇出型態相容的剖面在適當的環境(context)下織入功能程式模組。這樣以型態推演程序完成織入的作法，不僅可以避免單就語法結構來決定織入剖面的缺點，並有機會可以替多型剖面奠定一個良好的理論基礎，並進而在多型程式的設計上，導入剖面機制，改善多型程式的模組結構。

研究計畫中英文摘要：

Keywords: Aspect-Oriented Programming, Polymorphic Aspects, Weaving, Type Inference

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system. 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, designate 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.

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 at 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 also introduce many advanced aspect features to *AspectFun*, and show how these features are handled by our compilation model. We present a type-directed weaving scheme that coherently weaves type-scoped advices into the base program at compile time. We provide the detailed proof of 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.

目錄

1. INTRODUCTION

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system[Kiczales et al. 1997]. 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. Moreover, as pointed out by Sereni and de Moor [Sereni and de Moor 2003], 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 determine *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 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 do not address this issue of static and yet coherent weaving. In AspectML [Dantas et al. 2007] (*a.k.a* PolyAML [Dantas et al. 2005]), dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [Masuhara et al. 2005] takes a lexical approach which sacrifices coherence[1] for static weaving.

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.

This paper consolidates our past research in this field [Wang et al. 2006b; 2006a; Chen et al. 2007] and makes significant revisions and extensions to several dimensions of our research. Specifically, in this paper,

[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 [Jones 1992] which states that different translations of an expression are semantically equivalent.

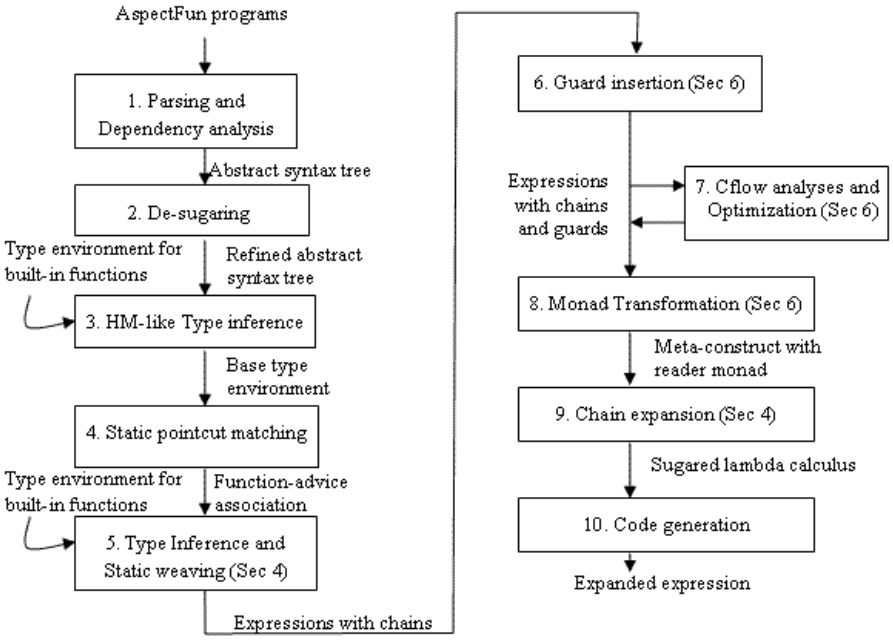

Fig. 2. Compilation Model for AspectFun

(1) Language features: We provide a complete treatment to several advanced features in our aspect-oriented functional language, AspectFun. These include: second-order advices, nested advices, complex pointcuts such as `cflowbelow` and curried pointcuts.

(2) Semantics for AspectFun: We present a lazy operational semantics for the entire AspectFun language, including that of control-flow based pointcuts, which has not been fully published before.

(3) Correctness: For the first time, we provide our full formulation of the correctness of static weaving *wrt.* the lazy operational semantics of AspectFun and its proof.

(4) Systems: We provide a complete implementation of our compilation model turning aspect-oriented functional programs into executable Haskell code, incorporating the analysis and optimisation of `cflowbelow` pointcuts. [2]

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. The result is expressed using some meta-constructs as follows:

```
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
```

[2]The prototype is available upon request.

```
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 then woven. 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. We gather all compilation processes pertaining to control-flow based pointcuts in Section 6.

The outline of the paper is as follows: Section 2 highlights various aspect-oriented features through AspectFun. Section 3 defines a lazy semantics for AspectFun. In Section 4, 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 6, we provide a detailed description of how control-flow based pointcuts are handled in our compilation model. We discuss related work in Section 7 and conclude in Section 8. Appendix A provides the detailed proof of the correctness of static weaving.

2. ASPECTFUN: THE ASPECT LANGUAGE

In this section, we introduce the aspect-oriented lazy functional language, AspectFun, for our investigation. Next, we briefly introduce the concept of advised type to capture the need for advice weaving based on type contexts. We also introduce various features of aspects in our language. These include: curried pointcuts, control-flow based pointcuts, second-order advices and nested advices.

Programs	π	::=	d in $\pi \mid e$
Declarations	d	::=	$x = e \mid f\ \overline{x} = e \mid n$@advice around $\{\overline{pc}\}\ (arg) = e$
Arguments	arg	::=	$x \mid x :: t$
Pointcuts	pc	::=	$ppc \mid pc + cf \mid pc - cf$
Primitive PC's	ppc	::=	$f\ \overline{x} \mid$ any $\mid$ any$\backslash[\overline{f}] \mid n$
Cflows	cf	::=	$\texttt{cflow}(f) \mid \texttt{cflow}(f(\_ :: t)) \mid \texttt{cflowbelow}(f) \mid \texttt{cflowbelow}(f(\_ :: t))$
Expressions	e	::=	$c \mid x \mid \texttt{proceed} \mid \lambda x.e \mid e\ e \mid$ 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	::=	$\forall \bar{a}.\rho$

Fig. 3. Syntax of the AspectFun Language

Figure 3 presents the language syntax. 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 set of 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 ease our presentation, complex syntax, such as `if` expressions and sequencings (`;`), are omitted even though they are used in examples.

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 in which advices are woven when program execution reaches there. 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 join point. An *around* advice is executed in place of the indicated join point, allowing the advised pointcut to be replaced. A special keyword `proceed` may be used inside the body of 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 primitive pointcut can also be a catch-all keyword `any`. When used, the corresponding advice will be triggered whenever a named function is invoked. For example, the pointcut $\mathsf{any}\backslash[f, g]$ will select all named functions except f and g. Besides, since advices are also named, we allow advices to advice other advices. 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.

Composite pointcuts are handled separately in our compilation model through series of code transformation, analyses and optimizations. This is discussed in detail in Section 6. Note that only global functions and advices are subject to advising; and invocations of anonymous functions are not considered as join points, even when `any` is used.

2.1 Advised Types

AspectFun is polymorphically and statically typed. Central to our approach is the construct of *advised types*, ρ in Figure 3, inspired by the *predicated types* [Wadler and Blott 1989] 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 4.1.

In the subsequent subsections, we use examples to illustrate the major features of AspectFun.

Essentially, the introduction of advised types enables us to perform static and coherent weaving of type-scoped advices. For instance, in Example 1, the type

scheme for `f` will be inferred to be $\forall a.(h : a \to a).a \to a$, which indicates that whenever `f` is applied in a specific context, the advices on `h` may also be triggered. Specifically, inside the main expression, all three function calls will carry some proper sets of advices on `h`, even for the calls that are made to `f`. Then, when a `h`-call is eventually invoked, the right advices to `h` will be triggered.

2.2 Curried Pointcuts

Higher-orderness naturally brings forth the notion of partial application of curried functions. In typical applications of AOP such as tracing or profiling, it is important to be able to advise on not only full applications of functions, but also partial applications of curried functions to their arguments other than the first one. In our system, we allow type-scoped advices with curried-function pointcuts, as shown below:

Example 2

```
n1@advice around {f x} (arg::Int) = e1 in
n2@advice around {f} (arg::Int) = e2 in
n3@advice around {f x} (arg) = e3 in
let f x y = x + 1 in
f 1 2
```

We use meta-variables (in italic font) such as e_1, e_2 and so on to represent expressions with omitted details. Through out this paper, without special mentioning, we assume expressions represented by meta-variables are not advised. Note that because of well-typedness, we insist that the advice body of `n2`, e_2, to be of a function type $\cdot \to \cdot$, while the advice bodies e_1 and e_3 are of non-function types.

There have been some recommmendation for defining operational semantics of aspect-oriented programs for higher-order functions and partial function applications, scuh as [Tucker and Krishnamurthi 2003]. Nevertheless, there is still not yet any unanimous and unambiguous agreement on it. In this work, we adopt the viewpoint that functions are identified by names at pointcuts, and advices are triggered whenever their partial applications matches that of the pointcuts [Masuhara et al. 2005]. This differs from the recommendation by [Tucker and Krishnamurthi 2003] which triggers advices based on matching of run-time closures.

Back to Example 2, using a similar technique as in [Masuhara et al. 2005], we simplify curried advices into non-curried ones as follows:

```
n1 = \x.\arg -> let proceed' = proceed x in [proceed'/proceed]e1 in
n2 = \arg -> e2 in
n3 = \x.\arg -> let proceed' = proceed x in [proceed'/proceed]e3 in
f x y = x in
<f,{n1,n2,n3}> 1 2
```

We refer the readers to Section 4.6 for detail.

2.3 Control-flow Based Pointcuts

The composite pointcuts in AspectFun are those related to the control flow of a program. Specifically, we can write a pointcut which identifies a subset of invoca-

tions of a specific function based on whether they occur in the dynamic context of other functions. For example, the pointcut $f + \texttt{cflow}(g)$ 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 (eg. $f - \texttt{cflow}(g)$), 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 $\texttt{cflow}(f)$, and it captures all the join points in the control flow from the the specific application to function f, including that specific f-application. The second one is expressed as $\texttt{cflowbelow}(f)$, and it captures all the join points in the control flow from the specific application to f, but excluding that specific f-application. 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 which `f` would have to be executed only as partially trusted. This policy can be enforced by an aspect.

Example 3

`n@advice around {f + cflow(g)} (arg) =` e `in //fully trusted execution`
`f =` e' `// 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 unscoped 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

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

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 that 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 [Masuhara et al. 2003], 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 Section 6.

2.4 Second-Order Advices and Nested Advices

In AspectFun, advice names can also be primitive pointcuts. As such, we allow advices to be developed to advise other advices. 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. The loss of expressiveness of such an approach has been well argued in [Rajan and Sullivan 2005].

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 4

```
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 e
```

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 concerns. 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 does not exceed 50%.

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 the latter *nested advices*. This is particularly useful in security applications. Consider a different attempt to invoke the restricted function `f` from Example 3.

Example 5

```
n@advice around {f + cflow(g)} (arg) = e1  // fully trusted execution
n1@advice around {w} (arg) = f arg in
f x = e2  // partially trusted execution
w x = e3  in
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 that apply to the execution of their own bodies, directly or indirectly. These are identified as *circular advice execution applications*. Such applications "are very rare, and usually pathological and a symptom of an error in the program" [Avgustinov et al. 2005]. Our system does not allow such nested advices for the reason that circular *around* advices advising potentially recursive functions may form a scenario similar to polymorphic mutual recursion which threatens the decidability of type inference. We leave this to future investigation.

3. THE SEMANTICS OF ASPECTFUN

This section presents an operational semantics for 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. After a brief overview of FIL, we shall present a set of type-directed conversion rules for transforming an AspectFun program to an FIL program, followed by presenting the operational semantics of FIL and an example.

Program	π^I	::=	$(\overline{\text{Adv}}, e^I)$
Advice	Adv	::=	$(n : \varsigma, \overline{pc}, \tau, e^I)$
Join points	jp	::=	$f : \tau \mid \epsilon$
Expressions	e^I	::=	$v^I \mid x \mid proceed \mid e^I\ e^I \mid e^I\{\tau\} \mid \mathcal{LET}\ x = e^I\ \mathcal{IN}\ e^I$
Values	v^I	::=	$c \mid \lambda^{jp} x : \tau_x.\ e^I \mid \Lambda\alpha.\ e^I$
Types	τ	::=	$Int \mid Bool \mid \alpha \mid \tau \rightarrow \tau \mid [\tau]$
Type schemes	ς	::=	$\forall\overline{\alpha}.\ \tau \mid \tau$
Type Substitution	S	::=	$[\overline{\tau}/\overline{\alpha}]$

Fig. 4. Syntax of FIL

Figure 4 displays the syntactic constructs of FIL. An FIL program consists of a separate store of all advices and a main expression with function declarations. Each advice in FIL is represented as a tuple of four elements: (1) the name of the advice (n) with its type (ς); (2) the pointcuts ($\overline{pc}$) the advice selects; (3) the type scope (τ), and (4) the body (e^I) of the advice. Expressions in FIL, denoted by e^I, are extensions of those in AspectFun to include join-point-annotated lambda ($\lambda^{jp} x : \tau_x.e^I$), type abstraction ($\Lambda\alpha.e^I$) and type application ($e^I\{\tau\}$). As to type structures, FIL employs the same standard type scheme of the Hindley-Milner type system. Hence, except for the advice predicate part, FIL and AspectFun share the same type structures.

3.1 FIL conversion

We provide three sets of conversion rules to transform an AspectFun program into an FIL program, namely declaration conversion, expression conversion and type conversion, listed in figures 5, 6 and 7. The conversion process is led by the step $\pi \overset{\text{prog}}{\rightarrowtail} (\mathcal{A}, e^I)$ in the declaration conversion rule, $(\overset{\text{PROG}}{\rightarrowtail})$. The judgement $\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}$ asserts that an AspectFun program, π, under a type environment (also called conversion environment) Δ of structure $\overline{x : \varsigma}$, having type τ is converted to an FIL program, consisting of an expression e^I, and an advice store $\mathcal{A}$ which is a set of advice tuples. Another major conversion is expression conversion. The judgement $\Delta \vdash e : \tau \rightarrowtail e^I$ asserts that an AspectFun *expression* e having a type τ under Δ is converted to an FIL expression e^I. Lastly, the type conversion $\overset{\text{type}}{\rightarrowtail}$ simply strips away the advice predicate part of a type scheme in AspectFun and map the latter to an isomorphic type scheme in FIL. Here, the function fresh$(\overline{a})$ takes a sequence of type variables and returns the same number of new type variables in sequence.

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

$$(\textsc{Decl:Var}) \ \frac{\Delta \vdash e : \tau_x \rightarrowtail e^I_x \qquad \overline{\alpha} = \text{fv}(\tau_x) \setminus \text{fv}(\Delta) \qquad \Delta.x : \forall \overline{\alpha}.\ \tau_x \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}}{\Delta \vdash_D x \ = \ e \ \text{in} \ \pi : \tau \rightarrowtail \mathcal{LET} \ \ x \ = \ \Lambda \overline{\alpha}.\ e^I_x \ \ \mathcal{IN} \ \ e^I; \mathcal{A}}$$

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

$$(\textsc{Decl:Adv}) \ \frac{\Delta \vdash_D n@\text{advice around } \{\overline{pc}\}\ (x :: a) = e \ \text{in} \ \pi : \tau \rightarrowtail e^I; \mathcal{A}}{\Delta \vdash_D n@\text{advice around } \{\overline{pc}\}\ (x) = e \ \text{in} \ \pi : \tau \rightarrowtail e^I; \mathcal{A}}$$

$$(\textsc{Decl:Adv-An}) \ \frac{\begin{array}{c}t_x \overset{\text{type}}{\rightarrowtail} \tau_x \quad \Delta.\mathit{proceed} : \tau_1 \rightarrow \tau_2 \vdash \lambda x.e : \tau_1 \rightarrow \tau_2 \rightarrowtail \lambda x : \tau_1 . e^I_n \\ \overline{\alpha} = \text{fv}(\tau_1 \rightarrow \tau_2) \setminus \text{fv}(\Delta) \qquad \Delta.n : \forall \overline{\alpha}.\tau_1 \rightarrow \tau_2 \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}\end{array}}{\begin{array}{r}\Delta \vdash_D n@\text{advice around } \{\overline{pc}\}\ (x :: t_x) = e \ \text{in} \ \pi : \tau \rightarrowtail e^I; \\ \mathcal{A}.(n : \forall \overline{\alpha}.\tau_1 \rightarrow \tau_2, \overline{pc}, \tau_x \sqcap \tau_1, \Lambda \overline{\alpha}.\ \lambda^{n:\tau_1 \rightarrow \tau_2} x : \tau_1 .\ e^I_n)\end{array}}$$

$$\tau_1 \sqcap \tau_2 = \text{let } \mathsf{S} = mgu(\tau_1, \tau_2) \text{ in } \mathsf{S}\tau_1$$

Fig. 5. FIL declaration conversion rules

Since most of the conversion rules follow the standard Hindley-Milner style, we only highlight the distinguished parts here. First, the (Decl:Func) rule converts top-level functions to let-bound ones having annotated lambda $\lambda^{f:\tau} x : \tau_x . e^I$; the

annotation $\lambda^{(f:\tau)}$ highlights the fact that the function defines a join point of name and type $f : \tau$. The semantics of FIL uses these annotations to select 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 enabling advices with matching types to be triggered in the context of polymorphic functions.

Second, the (Decl:Adv) rule simply delegates the conversion to the (Decl:Adv-An) rule, making non-type-scoped advice a special case of type-scoped advice with the widest polymorphic scope. The (Decl:Adv-An) rule performs the real work of advice conversion and appends the converted advice to the advice store $\mathcal{A}$. Since the user-specific type scope, t_x, may be more specific or general than the inferred type for an advice's parameter, τ_1, the (Decl:Adv-An) rule computes the *greatest lower bound* of τ_1 and τ_x (corresponding to t_x), and makes it the final type-scope of the underlying advice. Moreover, we shall prove later that in a well-typed AspectFun program, either $\tau_1 \trianglerighteq \tau_x$ or $\tau_x \trianglerighteq \tau_1$, and hence $\tau_x \sqcap \tau_1$ is either τ_x or τ_1.

Third, the (Decl:Var) rule works in the same way as the rule (Decl:Func), except that it does not add the join point annotation, thus avoiding unwanted advice triggering.

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

$$\text{(Expr:Abs)}\ \frac{\Delta.x : \tau_x \vdash e : \tau \rightarrowtail e^I}{\Delta \vdash \lambda x.\ e : \tau_x \to \tau \rightarrowtail \lambda x : \tau_x.\ e^I} \qquad \text{(Expr:App)}\ \frac{\Delta \vdash e_1 : \tau' \to \tau \rightarrowtail e_1^I \quad \Delta \vdash e_2 : \tau' \rightarrowtail e_2^I}{\Delta \vdash e_1\ e_2 : \tau \rightarrowtail e_1^I\ e_2^I}$$

$$\text{(Expr:Let)}\ \frac{\Delta.f : \tau_f \vdash e_f : \tau_f \rightarrowtail e_f^I \qquad \overline{\alpha} = \mathrm{fv}(\tau_f) \setminus \mathrm{fv}(\Delta) \qquad \Delta.f : \forall\overline{\alpha}.\ \tau_f \vdash e : \tau \rightarrowtail e^I}{\Delta \vdash \mathsf{let}\ f\ =\ e_f\ \mathsf{in}\ e : \tau \rightarrowtail \mathcal{LET}\ \ f\ =\ \Lambda\overline{\alpha}.\ e_f^I\ \ \mathcal{IN}\ e^I}$$

Fig. 6. FIL expression conversion rules

3.2 Operational Semantics for FIL

We describe the operational semantics for AspectFun in terms of that for FIL. For ease of presentation, we leave the semantics for handling cflow-based pointcut to section 6.1. The reduction-based big-step operational semantics, written as $\Downarrow_{\mathcal{A}}$, is defined in Figure 8. Note that the advice store $\mathcal{A}$ is implicitly carried by all the rules, and it is omitted to avoid cluttering of symbols. Except advice triggering, these reduction rules follow the standard ones for a typed lambda calculus with constants.

Also shown in Figure 8 are the definitions of four auxiliary functions used for advice triggering and weaving. Triggering and weaving of advices are performed

$$(\overset{\text{TYPE}}{\rightarrowtail}) \quad \frac{\overline{b} = \text{fv}(t) \setminus \overline{a} \quad \overline{\alpha} = \text{fresh}(\overline{a}) \quad \overline{\beta} = \text{fresh}(\overline{b}) \quad \overline{a : \alpha}.\overline{b : \beta} \vdash t \overset{\text{type}}{\rightarrowtail} \tau}{\forall \overline{a}.\overline{p}.t \overset{\text{type}}{\rightarrowtail} \forall \overline{\alpha}.\tau}$$

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

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

Fig. 7. FIL type conversion rules between AspectFun and FIL

Expressions:

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

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

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

Auxiliary Functions:

$$\begin{array}{lll}
\mathit{Trigger} & : & e^I \times jp \to e^I \\
\mathit{Trigger}(e^I, \epsilon) & = & e^I \\
\mathit{Trigger}(\lambda x : \tau_x.\ e^I, f : \tau_f) & = & \mathit{Weave}(\lambda x : \tau_x.\ e^I, \tau_f, \mathit{Choose}(f, \tau_x)) \\
 & & \\
\mathit{Weave} & : & e^I \times \tau \times \overline{\text{Adv}} \to e^I \\
\mathit{Weave}(e^I, \tau_f, []) & = & e^I \\
\mathit{Weave}(e_f^I, \tau_f, a : advs) & = & \text{Let} \quad (n : \forall\overline{\alpha}.\ \tau_n, \overline{pc}, \tau, \Lambda\overline{\alpha}.\ e^I) = a \\
 & & \qquad \overline{\tau} \text{ be types such that } [\overline{\tau}/\overline{\alpha}]\tau_n = \tau_f \\
 & & \qquad e_p^I = \mathit{Weave}(e_f^I, \tau_f, advs) \\
 & & \qquad e_a^I = (\Lambda\overline{\alpha}.\ e^I)\{\overline{\tau}\} \\
 & & \qquad \lambda^{n:\tau_n} x : \tau_x.\ e_n^I = [e_p^I/proceed]e_a^I \\
 & & \text{In} \quad \mathit{Trigger}(\lambda x : \tau_x.\ e_n^I, n : \tau_n) \\
\multicolumn{3}{l}{\mathit{Choose}(f, \tau) = \{(n_i : \varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \mid\ (n_i : \varsigma_i, \overline{pc_i}, \tau_i, e_i^I) \in \mathcal{A},\ \tau_i \trianglerighteq \tau,} \\
\multicolumn{3}{r}{\exists pc \in \overline{pc_i}\ s.t.\ \mathit{JPMatch}(f, pc)\}} \\
\multicolumn{3}{l}{\mathit{JPMatch}(f, pc) = (f \equiv pc) \vee (pc \equiv \mathsf{any}) \vee (pc \equiv \mathsf{any} \setminus [\overline{h}] \wedge f \notin \overline{h})}
\end{array}$$

Fig. 8. Operational Semantics for FIL

during function applications, as shown in rule (OS:App). The advice 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. Three points worth mentioning here. First, in the main body of *Weave* function, the function *Trigger* is invoked again to handle any possible triggering of second-order advice. Second, among the advices matched by *JPMatch*, the function *Choose* keeps all the advices whose type scope is more general than the type passed to it, regardless of their return types. Consequently, it is likely that, during the subsequent execution of the woven advice, a runtime type error may occur and the reduction fails (unless, of course, the program has been analyzed to be safe by our type system). Third, advices selected by *Choose* are kept in a set, rather than an ordered sequence. While it is understood that advices are to be chained in a specific order during execution, we believe that the issue of chaining order is orthogonal to our study here. Consequently, we do not explicitly fix the order in our semantics definition, neither do we choose a specific advice chaining order in the static translation of AspectFun programs. Instead, we assume that the order chosen during static translation *is consistent with* that arranged by the *Choose* function in the operational semantics.

3.3 Example

We use a contrived example to demonstrate how the semantics of AspectFun works. The AspectFun program listed in Example 6 includes three kinds of advices, namely type-scoped advice, polymorphic advice and second-order advice. They will be triggered according to the type context at different join points during the execution of the program.

Example 6

```
nscope@advice around {f} (arg::[a]) = proceed (tail arg) in
n @advice around {g} (arg) = proceed arg in
n2nd @advice around {n} (arg) = proceed arg in
f x = x in
g x = (f x, f (x, x), f [x]) in
h x = g [x] in
k x = g x in
(h 1, k 2)
```

After applying the rules in figures 5 and 6 to the above AspectFun program, we get the following FIL-converted expression:

$$
\begin{array}{lll}
\mathcal{LET}\ \mathtt{f} = \Lambda\alpha.\lambda^{\mathtt{f}:\alpha\to\alpha}\mathtt{x}:\alpha.\mathtt{x} & & \mathcal{IN} \\
\mathcal{LET}\ \mathtt{g} = \Lambda\alpha.\lambda^{\mathtt{g}:\tau_g}\mathtt{x}:\alpha.(\mathtt{f}\{\alpha\}\mathtt{x},\mathtt{f}\{(\alpha,\alpha)\}\mathtt{(x,\ x)},\mathtt{f}\{[\alpha]\}\mathtt{[x]}) & & \mathcal{IN} \\
\mathcal{LET}\ \mathtt{h} = \Lambda\alpha.\lambda^{\mathtt{h}:\tau_h}\mathtt{x}:\alpha.\mathtt{g}\{[\alpha]\}\mathtt{x} & & \mathcal{IN} \\
\mathcal{LET}\ \mathtt{k} = \Lambda\alpha.\lambda^{\mathtt{k}:\tau_g}\mathtt{x}:\alpha.\mathtt{g}\{\ \alpha\ \}\mathtt{x} & & \mathcal{IN} \\
(\mathtt{h}\ \{Int\}\ 1, \mathtt{k}\ \{Int\}\ 2) & &
\end{array}
$$

where τ_g and τ_h are abbreviations for $\alpha \to (\alpha, (\alpha, \alpha), [\alpha])$ and $\alpha \to ([\alpha], ([\alpha], [\alpha]), [[\alpha]])$, respectively. Besides, the advice store $\mathcal{A}$ thus produced contains the following three tuples:

$(\texttt{nscope} : \forall\alpha.[\alpha] \to [\alpha],\ \texttt{f},\ [\alpha],\ \Lambda\alpha.\lambda^{\texttt{nscope}:[\alpha]\to[\alpha]}\texttt{arg} : [\alpha].\ \texttt{proceed (tail arg)}),$
$(\texttt{n} : \forall\alpha\beta.\alpha \to \beta,\ \texttt{g},\ \alpha,\ \Lambda\alpha.\Lambda\beta.\lambda^{\texttt{n}:\alpha\to\beta}\texttt{arg} : \alpha.\ \texttt{proceed arg}),$
$(\texttt{n2nd} : \forall\alpha\beta.\alpha \to \beta,\ \texttt{n},\ \alpha,\ \Lambda\alpha.\Lambda\beta.\lambda^{\texttt{n2nd}:\alpha\to\beta}\texttt{arg} : \alpha.\ \texttt{proceed arg})$

When evaluating the converted program, the application of `h` $\{Int\}$ 1 in the main expression will result in the invocation of `g` $\{[Int]\}$ 1, which will then lead to the weaving of advices `n` and `n2nd`. During the evaluation of `g`'s body, `f` will be applied to three different types of arguments: $[Int]$, $([Int], [Int])$, and $[[Int]]$. The advice *nscope* will be triggered, except for the second one, since the call $Choose(\texttt{f}, ([Int], [Int]))$ returns an empty set. The case for the application of (`k` $\{Int\}$ 2) is also similar. The notable difference is that, during the evaluation of the three function calls to `f`, only the last one of `f` $\{[Int]\}$ will trigger the advice `nscope`. Finally, the result of executing the FIL program is

$$((], ([1], [1]), []), (2, (2, 2), []))$$

4. 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 6.

4.1 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` possesses the advised type $\forall a.(\texttt{h} : a \to a).a \to a$, in which $(\texttt{h} : a \to a)$ is called an *advice predicate*. It signifies that *the execution of any application of* `f` *may require triggering of those advices on* `h` *whose types can be instantiated to* $t' \to t'$*, where* t' *is an 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 advices *indeterminable* at compile time. 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: (Gen) is a generalization procedure turning a type into a type scheme by quanti-

(GEN) $gen(\Gamma, \rho) = \forall \bar{a}.\rho$ where $\bar{a} = fv(\rho) \backslash fv(\Gamma)$ (CARD) $|o_1 ... o_k| = k$

Fig. 9. Auxiliary Definitions

fying type variables that do not appear free in the type environment. The function $|\cdot|$ returns the cardinality of a sequence of objects.

The main set of type inference rules, as described in Figure 10, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \rho \leadsto e'$ to denote that expression e has type ρ 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}.

The typing environment Γ contains not only the usual type bindings (of the form $x : \sigma \leadsto e$) 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. This type σ is inferred from the body and type scope of the advice described in rules (ADV) and (ADV-AN); and it is used to guard advice application in rule (VAR-A). When a bound function name is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish it from the non-advised ones so that the former may appear in an advice 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 always shadows previous ones.

4.2 Predicating and Releasing

Before illustrating the main typing rules, we introduce a *weavable* constraint of the form $wv(f : t)$ which indicates that applicable advices to be triggered at the call to f instantiated with type t. It is formally defined as:

Definition 2 *Given a function f and its type $t_1 \to t_2$, the predicate $wv(f : t_1 \to t_2)$ holds iff the following implication holds:*

$$((\forall n.n :_{(*)} \forall \bar{a}.\bar{p}.t'_1 \to t'_2 \bowtie f) \in \Gamma \wedge t_1 \sim t'_1) \Rightarrow (t'_1 \to t'_2 \trianglerighteq t_1 \to t_2).$$

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 $\{\texttt{n1} : \forall a.[a] \to [a] \bowtie \texttt{f}, \texttt{n2} : Int \to Int \bowtie \texttt{f}\}$, $wv(\texttt{f} : b \to b)$ is false because the type is not specific enough to determine whether `n1` and `n2` should apply whereas $wv(\texttt{f} : Bool \to Bool)$ 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

Expressions:

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

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

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

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

Declarations:

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

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

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

Fig. 10. Static Typing and Weaving rules

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 those of the advices with unifiable types), 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 $\{n_i | t_i \trianglerighteq t'\}$). We coerce all these selected advices to have non-advised type during their translation $\Gamma \vdash n_i : t' \rightsquigarrow e_i$. This ensures correct weaving of nested advices advising the bodies of the selected advices. The detail will be elaborated in Section 4.4. 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 the application of rule (Rel).

We illustrate the application of rules (Pred) and (Rel) 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:

$$\begin{array}{l}\Gamma = \{\ \mathtt{h} :_* \forall a.a \to a \rightsquigarrow \mathtt{h}, \mathtt{n}_3 : \forall a.a \to a \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_3, \mathtt{n}_4 : \forall a.[a] \to [a] \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_4,\\ \quad \mathtt{n}_5 : \forall b.[C] \to [C] \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_5\}\end{array}$$

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

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

$$\begin{array}{l}\Gamma_3 = \{\ \mathtt{h} :_* \forall a.a \to a \rightsquigarrow \mathtt{h}, \mathtt{n}_3 : \forall a.a \to a \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_3,\ \mathtt{n}_4 : \forall a.[a] \to [a] \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_4,\\ \quad \mathtt{n}_5 : \forall b.[C] \to [C] \bowtie \mathtt{h} \rightsquigarrow \mathtt{n}_5,\ \mathtt{f} : \forall a.(\mathtt{h} : a \to a).a \to a \rightsquigarrow \mathtt{f}\}\end{array}$$

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \forall a.(\mathtt{h} : a \to a).a \to a \rightsquigarrow \mathtt{f} \in \Gamma_3}{\Gamma_3 \vdash \mathtt{f} : (\mathtt{h} : [C] \to [C]).[C] \to [C] \rightsquigarrow \mathtt{f}}\ (\textsc{Var}) \quad \dfrac{\mathtt{h} :_* \forall a.a \to a \rightsquigarrow \mathtt{h} \in \Gamma_3 \quad ...}{\Gamma_3 \vdash \mathtt{h} : [C] \to [C] \rightsquigarrow \langle \mathtt{h}\ , \{\mathtt{n}_3, \mathtt{n}_4, \mathtt{n}_5\}\rangle}\ (\textsc{Var-A})}{\Gamma_3 \vdash \mathtt{f} : [C] \to [C] \rightsquigarrow (\mathtt{f}\ \langle \mathtt{h}\ , \{\mathtt{n}_3, \mathtt{n}_4, \mathtt{n}_5\}\rangle)}\ (\textsc{Rel}) \quad ...}{\Gamma_3 \vdash (\mathtt{f}\ \text{"}c\text{"}) : [C] \to (\mathtt{f}\ \langle \mathtt{h}\ , \{\mathtt{n}_3, \mathtt{n}_4, \mathtt{n}_5\}\rangle\ \text{"}c\text{"})}\ (\textsc{App})$$

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

Rules (Pred) and (Rel) introduce and eliminate advice predicates respectively. Rule (Pred) adds an advice predicate to a type. 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.

4.3 Handling Advices

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

Rule (Global) is very similar to rule (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 passing, we note that these two rules can be merged but it makes the rule rather complicated. Hence we keep them separated. In addition, as all the advices are of function types, attempts to advise a non-function type expression will be rejected by the type system.

4.4 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 5, in which the control-flow based pointcut is removed and a type scope is added for illustration purpose.

```
n@advice around {f} (arg::Int) = e1 in // fully trusted execution
n1@advice around {w} (arg) = f arg in
f x = e2 in
w x = e3 in
h x y = x y in
h w 1
```

Here, advice `n1` calls `f` which will then be advised. The goal of our translation is to enable correct chaining of advices even within some advice body, such as that of `n1`. 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 $wv(\texttt{f} : a \to a)$ 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 ensures 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. This is the case, in the above program, for the call (`f arg`) in the body of `n1`. Chaining is thus delayed by introducing an advice parameter `df` into the (translated) declaration of `n1`. Then, at the base program, when `w` is called indirectly from `h` with the specific calling context (type of the argument Int), the call to `f` inside the body of `n1` is inferred to have argument of type Int. Thus, the weavable condition, $wv(\texttt{f} : Int \to Int)$, is satisfied, and the program is translated as follows."

```
n = \arg. e1'  in //fully trusted execution
n1 df = \arg. df arg in
f x = e2' in
w x = e3' 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`.

To handle second-order advices, we slightly adapt the rules (Adv) and (Adv-an) by allowing them to introduce $:_*$-binding for advices that are advised and to replace condition $f_i : \forall \bar{a}.t' \in \Gamma_{base}$ by $n_i :_* \forall \bar{a}.\bar{q}.t' \in \Gamma$ to make up the fact that type information of advices is not stored in Γ_{base}. By doing this, we assume advised advices are translated before the advices defined on them. This is valid because circular cases are precluded, as mentioned in Section 2.4.

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, we do not allow advices to be used as advice predicates because it causes complications without adding expressiveness: any predication of advises can be replaced by predication of the function calls that trigger the advices.

With the newly adapted rules, 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`.

4.5 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.

Let's illustrate this with an example of advising tail recursive functions. Many list manipulation functions, such as `reverse`, `append`, and `mergeSort`, can be written in a tail recursive pattern in which their accumulating parameter is simply returned

when their input parameter is empty. We can capture this pattern using an advice as follows.

```
n@advice {reverse, append, mergesort} (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] []
```

Here we focus on the `reverse` function to show our scheme. 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 $[a] \to [a] \to [a]$.

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

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

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.

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

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] []
```

4.6 Curried Pointcuts

In [Masuhara et al. 2005], Masuhara *et al.* proposed a technique to simplify a curried pointcut by iteratively removing the last parameter in it. Unfortunately, their weaving strategy does not support type-scoped curried pointcuts.

In our approach, we also simplify curried pointcuts into uncurried pointcuts, but we maintain the type constraints of type-scoped curried advices in the environment. Furthermore, the special function *proceed* is redefined locally to effect the currying of function arguments. Because function calls are not handled syntactically (*i.e.*, they are not handled according to their textual appearances), our approach can deal with type-scoped curried pointcuts straightforwardly: What we need is to introduce

a slight variant of the (Adv-an) rule. For the sake of simplicity, this variant rule only deals with curried functions with two parameters. It can be straightforwardly extended to handle curried functions with arbitrary number of parameters.

$$\text{(Adv-C)}\quad \frac{\begin{array}{c}\Gamma.\mathit{proceed} : t \to t_y \to t_a \vdash \lambda x.\lambda y.e_a : \bar{p}.t \to t_y \to t_a \rightsquigarrow e'_a \\ f_i : \forall \bar{a}.t_1 \to t_2 \to t_3 \in \Gamma_{base} \quad \sigma = gen(\Gamma, \bar{p}.t \to t_y \to t_a) \quad S = [\bar{t}/\bar{a}]t_2 \trianglerighteq t_y \\ t \trianglerighteq S[\bar{t}/\bar{a}]t_1 \quad t_a \trianglerighteq S[\bar{t}/\bar{a}]t_3 \quad \Gamma.n : \sigma \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t' \rightsquigarrow \pi'\end{array}}{\begin{array}{c}\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\ x\}\ (y :: t_y) = e_a \mathsf{in}\ \pi : t' \rightsquigarrow \\ n = \lambda x.\lambda y.\mathsf{let}\ \mathit{proceed}' = \mathit{proceed}\ x\ \mathsf{in}\ [\mathit{proceed}'/\mathit{proceed}]e'_a\ \mathsf{in}\ \pi'\end{array}}$$

Given the code displayed in Example 2 involving partial applications of curried functions, our rule translates it to the following:

```
n1 = \x.\arg -> let proceed' = proceed x in [proceed'/proceed]e1 in
n2 = \arg -> e2 in
n3 = \x.\arg -> let proceed' = proceed x in [proceed'/proceed]e3 in
f x y = x in
<f,{n1,n2,n3}> 1 2
```

As e_2 is ensured by the typing rules to be a function type, the types of the three advices `n1`, `n2` and `n3` are unifiable with `f`'s. Our translation is general enough to handle curried functions even with type annotations on arguments other than the first one.

4.7 Translating Chain Expressions

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 separated into two steps: *addProceed* and *chain expansion*, as shown in Figure 11. Function *addProceed* turns the keyword `proceed` into a parameter of all advices representing the rest of computation (i.e., continuation). Expansion of meta-construct (chains) is defined 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.

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 section 6, we show yet another optimization of control-flow based pointcuts which take advantage of the explicit intermediate form.

To close this section, we apply the static weaving and chain expansion steps to the AspectFun program in Example 6. The intermediate result produced by static weaving is as follows:

```
nscope arg = proceed (tail arg) in
```

$$
\begin{array}{lcl}
e_M & : & \text{Expressions containing meta-constructs} \\
\mathit{addProceed} & : & e_M \longrightarrow e_M \\
\mathit{addProceed}\ (n\ \mathit{df}\ \mathit{arg}\ =\ e_1\ \mathsf{in}\ e_2) & = & \mathit{if}\ (n \text{ is an advice})\ \mathit{then} \\
 & & \quad n\ \mathit{df}\ \texttt{proceed}\ \mathit{arg}\ =\ e_1 \\
 & & \quad \mathsf{in}\ \mathit{addProceed}(e_2) \\
 & & \mathit{else}\ n\ \mathit{df}\ \mathit{arg}\ =\ e_1\ \mathsf{in}\ \mathit{addProceed}(e_2) \\
\mathit{addProceed}\ (e) & = & e
\end{array}
$$

$$
\begin{array}{lcl}
[\![\cdot]\!] & : & e_M \longrightarrow \text{Expanded expression} \\
[\![x\ =\ e_1\ \mathsf{in}\ e_2]\!] & = & x\ =\ [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\
[\![\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ x\ =\ [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\
[\![x]\!] & = & x \\
[\![\texttt{proceed}]\!] & = & \texttt{proceed} \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\
[\![\langle f\ \overline{e}\ , \{\}\rangle]\!] & = & [\![f\ \overline{e}]\!] \\
[\![\langle f\ \overline{e}\ , \{e_a, \overline{e_{advs}}\}\rangle]\!] & = & \mathit{proceedApply}(e_a, \langle f\ \overline{e}\ , \{\overline{e_{advs}}\}\rangle)
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{proceedApply}(n\ \overline{e}, k) & = & [\![n\ \overline{e}\ k]\!] & \text{if } \mathit{rank}(n) = 0 \\
\mathit{proceedApply}(\langle n\ \overline{e}\ , \{\overline{ns}\}\rangle, k) & = & [\![\langle n\ \overline{e}\ k\ , \{\overline{ns}\}\rangle]\!] & \text{otherwise}
\end{array}
$$

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

Fig. 11. Definition of Chain Expansion with `proceed` Lifting

```
n      arg = proceed arg in
n2nd   arg = proceed arg in
f x = x                                                    in
g df x = (df x, f (x, x), <f, {nscope}> [x])               in
h    x = (\df. <g df, {<n, {n2nd}>}>) <f, {nscope}> [x]    in
k df x = (\df. <g df, {<n, {n2nd}>}>) df            x      in
(h 1, k f 2)
```

After applying *addProceed* and chain expansion, the final result is the following expanded expression:

```
n      proceed arg = proceed arg in
nscope proceed arg = proceed (tail arg) in
n2nd   proceed arg = proceed arg in
f x = x                                              in
g df x = (df x, f (x, x), nscope f [x])              in
h    x = (\df. n2nd (n (g df))) (nscope f) [x]       in
k df x = (\df. n2nd (n (g df))) df x                 in
(h 1, k f 2)
```

4.8 Unresolved Advice Predicates

A problem inherent with our advised type approach to static weaving is the possibility of unresolved advice predicates . For example, consider the following AspectFun program:

```
n@advice around {f} (arg::[Char]) = proceed (tail arg) in
f l = length l in
g i = i + f [] in
g 5
```

After static weaving, the function `g` has type scheme $\forall a.(\texttt{f} : [a] \rightarrow Int).Int \rightarrow Int$, and is translated to the following intermediate result:

```
g df i = i + df []
```

As the type-scope of `f`'s advice `n` is more specific than $[a]$, the static weaver cannot resolve the advice predicate $(\texttt{f} : [a] \rightarrow Int)$. Hence, subsequently when `g` is applied (`g 5` above), the static weaver will be forced to resolve this advice predicate arbitrarily. In particular, depending on what the type variable a is instantiated to, advice `n` may or may not be applied.

Obviously this is unacceptable. Thus we should consider such programs as ill-typed and reject them statically. Similar to Haskell's type classes, such an unresolved advice predicate, p, manifest itself in an advised type, $\bar{p}.t$, as there are some type variables in p, but not in the type body t. Hence we can easily detect it during typing a definition. Specificaly, we refine the $gen(\Gamma, \bar{p}.t)$ function used in the typing rules so that if $fv(\bar{p}) \not\subseteq fv(t)$, then gen will return an error to reject the expression under typing.

5. CORRECTNESS OF STATIC WEAVING

The correctness of static weaving is proven by relating it to the operational semantics of AspectFun. Specifically, given an AspectFun program, we prove that if it is well-typed by our static typing and weaving rules, then the resulting woven program, after chain expansion, is equivalent to the FIL-converted program according to the operational semantics of AspectFun. The detail of the correctness proof is available in Appendix A. In this section, we outline and explain the structure of our proof.

Figure 12 depicts the structure of our main proof steps. The basis for our proof is the definition of "equivalence" between terms from AspectFun and FIL. We shall employ three related definitions of equivalence to develop our proof. First, we define an equivalence relation, denoted by $\sim\!\!\sim$, between a closed and expanded expression e'' of AspectFun, and an FIL expression e^I in a program context. This is the ultimate equivalence result we aim to achieve in our proof. Second, as an intermediate step, we extend the equivalence relation to open terms based on a form of *body substitution* and denote this open equivalence relation by $\simeq$. Third, based on open equivalence, we define the notion of respect of expressions, denoted by $\propto$ to relate a woven AspectFun expression, e', to an FIL-expression, e^I, under a specific type context. The main theorems are derived in reverse order. We shall first prove that, any intermediate expression obtained during static weaving of an AspectFun program respects its corresponding FIL-converted expression, and then

go on to prove the closed-equivalence between the whole expanded expression of an AspectFun program and the whole FIL-converted expression of the same program. Note that since an expanded expression e'' is basically an AspectFun expression e without those advice-related constructs, to avoid notation cluttering, we shall use e and e'' interchangeably in the following discussion when the context is clear.

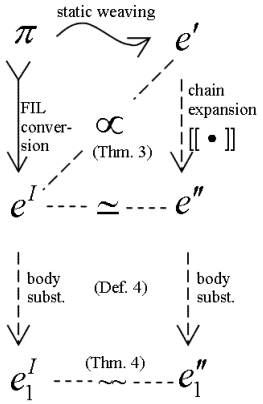

Fig. 12. Relationship between expressions and equivalences

We begin with the definition of the equivalence between closed terms of AspectFun expressions and FIL expressions. As the operational semantics of AspectFun is reduction-based, we define the equivalence relation using the proper reduction rules.

Definition 3 ($\sim\!\sim$) *A closed expanded expression e is said to be equivalent to a closed FIL expression e^I under an advice store $\mathcal{A}$, written as $e \sim\!\sim e^I$, if the following holds:*

$$e \longmapsto_\beta^* v \text{ iff } e^I \Downarrow_{\mathcal{A}} v^I$$

where v^I is not a type abstraction and $v \sim\!\sim_v v^I$.

In the above, $v \sim\!\sim_v v^I$ is defined by:

$$\frac{}{c \sim\!\sim_v c}$$

$$v \sim\!\sim_v \lambda^{jp} x : \tau.e^I \;\; \textit{iff} \;\; \forall e_1 \sim\!\sim e_1^I,\; (v\ e_1) \sim\!\sim ((\lambda^{jp} x : \tau.e^I)\ e_1^I) \quad \textit{when the RHS is well-typed}$$

We shall omit the advice store $\mathcal{A}$ when it is obvious from the context.

Next, we extend the equivalence relation to open terms of expanded AspectFun expressions and FIL expressions. The free variables which may occur in them, directly or indirectly, are those introduced via top-level bindings, local let-bindings and lambda-bound parameters.[3] They are subjected to static weaving and FIL-conversion. (However, these have no effect on lambda parameters.) Hence we

[3]Note that, after static weaving, advice names, indicated by n, may appear as free variables in an expanded AspectFun expression. In addition, *proceed* is treated as a lambda parameter.

introduce some notations for referring to the intermediate results derivable from static weaving and FIL-converting a declaration body, as shown in Figure 10 and Figure 5 respectively. For example, based on the (GLOBAL) rule, we have the following static typing and weaving result for a global variable declaration:

$$\frac{\Gamma \vdash\ e_x : \bar{p}.t \rightsquigarrow \lambda\overline{dp}.e'_x \quad \sigma = gen(\Gamma, \bar{p}.t) \quad \Gamma.x : \sigma \rightsquigarrow x \vdash\ \pi : t \rightsquigarrow \pi'}{\Gamma \vdash\ \ x = e_x \ \mathsf{in}\ \pi : t \rightsquigarrow\ \ x = \lambda\overline{dp}.e'_x \ \mathsf{in}\ \pi'}$$

Note that the woven code may be a lambda abstraction with advice parameters. Notation-wise, given the AspectFun expression $x = e_x$ shown in the above global variable declaration, we write $Body(x)$ to denote the entire woven code resulting from the declaration body; ie., $\lambda\overline{dp}.e'_x$. We also use $body(x)$ to denote the main part – without advice parameters – of the woven code; ie., e'_x. Similarly, we write $Body^I(x)$ for the entire result of FIL-converting a variable declaration body; ie., $\Lambda\overline{\alpha}.e^I_x$. We write $body^I(x)$ for the main part of the conversion; ie., e^I_x.

Moreover, the notations for the body results of a function or an advice declaration will be different from that of a variable declaration. Hence, in the following we categorise these body notations by the forms of the declarations; namely variable, function and advice.

—For a top-level or a let-bound variable declaration of x with body e_x, which will be FIL-converted to $\Lambda\overline{\alpha}.e^I_x$, we have

$$\begin{array}{lcl} Body(x) & = & \lambda\overline{dp}.e'_x \\ body(x) & = & e'_x \\ Body^I(x) & = & \Lambda\overline{\alpha}.e^I_x \\ body^I(x) & = & e^I_x \end{array}$$

—For a top-level or a let-bound function declaration of g with body $\lambda x.e_g$, which will be FIL-converted to $\Lambda\overline{\alpha}.\lambda^{g:\tau_g} x : \tau_x.e^I_g$, we have

$$\begin{array}{lcl} Body(g) & = & \lambda\overline{dp}.\lambda x.e'_g \\ body(g) & = & e'_g \\ Body^I(g) & = & \Lambda\overline{\alpha}.\lambda^{g:\tau_g} x : \tau_x.e^I_g \\ body^I(g) & = & e^I_g \end{array}$$

—For a top-level advice n with body $\lambda x.e_n$, which will be FIL-converted to $\Lambda\overline{\alpha}.\lambda^{n:\tau_x\rightarrow\tau_n} x : \tau_x.e^I_n$, we have

$$\begin{array}{lcl} Body(n) & = & \lambda\overline{dp}.\lambda x.e'_n \\ body(n) & = & e'_n \\ Body^I(n) & = & \Lambda\overline{\alpha}.\lambda^{n:\tau_x\rightarrow\tau_n} x : \tau_x.e^I_n \\ body^I(n) & = & e^I_n \end{array}$$

In subsequent discussion, we also write $Body(\bar{x})$ for $\overline{Body(x)}$, and similarly for other body notations.

As to the lambda-bound parameters, we shall denote them by $\overline{y}$ to distinguish them from the other categories of declared entities. With these notations, we can now define the open equivalence in terms of the closed equivalence, $\sim\!\sim$, and substitutions of *body expressions* as follows.

Definition 4 ($\simeq$) *Let e be an expanded expression typed under environment Γ and e^I an FIL expression to be woven dynamically using the advice store $\mathcal{A}$. We define an open equivalence between e and e^I under Γ and $\mathcal{A}$, written as $e \simeq e^I$, if for all $\overline{e_v} \sim\sim \overline{e}_v^I$ such that $[\overline{e}_v^I/\overline{y}]e^I$ is well-typed, then*

$$[\overline{e_v}/\overline{y}][Body'(\overline{x})/\overline{x}]e \sim\sim [\overline{e}_v^I/\overline{y}][Body^I(\overline{x})/\overline{x}]e^I \quad \text{under } \mathcal{A}$$

where

$$Body'(x) = \begin{cases} [\![Body(x)]\!] & \text{if } x \text{ is not an advice} \\ [\![\lambda\overline{dp}.\lambda proceed.\lambda z.body(x)]\!] & \text{if } x \text{ is an advice} \end{cases}$$

We shall omit Γ and $\mathcal{A}$ when they are obvious from the context.

As an example of the open equivalence relation, consider the definition of `g` in Example 6 (Section 3.3): `g x = (f x, f(x, x), f [x])`. In particular, we focus on the part in which function `f` is applied to a list argument `[x]`. As described in Section 4.7, after static weaving, this occurrence of function `f` is woven with advice `nscope` as `<f, {nscope}>` and is subsequently expanded to `nscope f`. By contrast, on the FIL side, it will be converted to $f\ \{[\alpha]\}$. Then, according to the definition of open equivalence, we apply the body substitutions to (`nscope f`) and (`f` $\{[\alpha]\}$), and get `\arg.((\x.x)(tail arg))` and $\lambda\ \mathtt{x}^{\mathtt{f}:[\alpha]\to[\alpha]} : [\alpha].\mathtt{x}$, respectively. Clearly, when the substituted FIL expression is invoked with an argument of type $[\alpha]$, the advice `nscope` will be triggered and executed. Hence, `\arg.((\x.x)(tail arg))` $\sim\sim$ $\lambda\mathtt{x}^{\mathtt{f}:[\alpha]\to[\alpha]} : [\alpha].\mathtt{x}$. It then follows that (`nscope f`) $\simeq$ `f` $\{[\alpha]\}$.

Note that FIL expressions do not contain advice; advices only take effect when advised functions are invoked. Hence, the open equivalence relation above does not consider the case when e^I involves advice. However, due to nested advice and second-order advice, static weaving will treat such advices as *advised* functions and transform any occurrence of such advice into an intermediate expression that should be related to its corresponding FIL expression, which is simply advice with type application.[4] Hence we need to enhance the definition of open equivalence to handle the cases when the FIL expression under equivalence test is an advice expression.

As an example, consider the function `k` in Example 6 (Section 3.3): `k x = g x`. After static weaving, the function `g` inside `k` is translated to `\df.<g df, {<n, {n2nd}>}>` and then expanded to `\df.(n2nd n (g df))` In other words, the invocation of `g` will trigger advice `n`, which in turn will trigger the second-order advice `n2nd`. Here, although (`n2nd n`) is not a complete expanded expression, we still need to relate it to its FIL-corresponding part: $n\{\alpha\}$, which, when applied, will also trigger advice `n2nd`.

To handle such intermediate expressions produced by static weaving in the presence of second-order advices, the definition of open equivalence needs to consider the case when $e^I \equiv n\{\overline{\tau}\}$. Specifically, the definition of $Body'(x)$ is extended to:

$$Body'(x) = \begin{cases} [\![Body(x)]\!] & \text{if } x \text{ is not an advice or } x \equiv n \\ [\![\lambda\overline{dp}.\lambda\mathtt{proceed}.\lambda z.body(x)]\!] & \text{if } x \text{ is an advice and } x \not\equiv n \end{cases}$$

[4] Recall the premise $\Gamma \vdash n_i : t' \rightsquigarrow e_i$ in the (VAR-A) rule of static weaving.

In other words, when advice `n` is the FIL expression under the equivalence test, we treat it like an advised function and do not wrap its body with *proceed*. Now, applying the enhanced open equivalence with this modified $Body'(x)$ to (`n2nd n`) and $n\{\alpha\}$, we get (`\arg.(\arg'.proceed arg') arg`) and $(\Lambda\alpha.\Lambda\beta.\lambda^{n:\alpha\to\beta} arg : \alpha.\ \texttt{proceed}\ arg)\{\gamma\}$, respectively. When the substituted FIL expreesion is applied with an argument of proper type, advice `n2nd` will be triggered, too. Therefore, when the `proceed` in both intermediate expressions are replaced with equivalent values, the two body-substituted expressions will be equivalent, too.

To be applicable to an AspectFun program, the open equivalence relation has to be built on some kind of mutual agreement between the environment of static weaving and the FIL-conversion environment and the advice store. We begin with a definition which ensures that the type bindings in both systems are isomorphic instances and the pointcuts and type scopes of all advices agree.

Definition 5 (Respect of Environment) *A static weaving environment* Γ *is said to respect an* FIL*-conversion environment,* Δ*, and an advice store* $\mathcal{A}$*, written as* $\Gamma \propto (\Delta, \mathcal{A})$*, if*

(1) $x :_{(*)} \sigma \in \Gamma \Leftrightarrow x : \varsigma \in \Delta$ *where* $\sigma \overset{type}{\rightarrowtail} \varsigma'$ *and* ς' *is an instance of* ς.

(2) $n : \forall\overline{a}.\overline{p}.t_y \to t_n \bowtie f \ \in \Gamma \Leftrightarrow (n : \varsigma, \overline{pc}, \tau_y, e_n^I) \in \mathcal{A}$ *where* $t_y \overset{type}{\rightarrowtail} \tau_y$, $(\forall\overline{a}.t_y \to t_n) \overset{type}{\rightarrowtail} \varsigma$ *and* $JPMatch(f, pc_i) \equiv \mathsf{true}$ *for some* i.

Next, we need to define the mutual agreement between the binding definitions found in the static weaving environment and those kept by the FIL-conversion environment and the advice store. As advice predicates may appear in the binding definitions produced by static weaving, we cannot apply the $\simeq$ and chain expansion directly to relate them to those in the FIL-conversion environment. Hence, we need to provide conditional form of equivalence which matches an AspectFun expression with advice predicates to an FIL expression in a way that is compliant with the $\simeq$ relation and satisfies the underlying advice predicates. First, we notice that the predicates created during static weaving can be realized at run-time through functions – and their associated advice – of appropriate types. This is captured by the notion of *feasibility*.

Definition 6 (Feasibility to predicates) *Given* Γ, Δ*, and* $\mathcal{A}$ *with* $\Gamma \propto (\Delta, \mathcal{A})$*, an expanded expression* e *is said to be* feasible *to a predicate* $g : t_g$*, written as* $e \fallingdotseq g : t_g$*, if* $wv(g : t_g)$ *and* $e \simeq g\{\overline{\tau}\}$ *where* $\forall\overline{\alpha}.\tau_g = \Delta(g)$ *and types* $\overline{\tau}$ *satisfies* $t_g \overset{type}{\rightarrowtail} [\overline{\tau}/\overline{\alpha}]\tau_g$.

As an example of predicate feasibility, consider the definition of `h` in Example 6 (Section 3.3): `h x = g [x]`. According to the static weaving described in Section 4.7, function `g` is typed with a predicate `f:a->a`. When it is applied to an argument of type `[b]` inside `h`, we get a more instantiated predicate `f:[b]->[b]`. As the type scope of the advice `nscope` for `f` matches the application context, $wv(\texttt{f} : [\texttt{b}] \to [\texttt{b}])$ holds. Hence, in this context, `f` can be statically woven with advice `nscope`. Besides, as stated above, $(\texttt{nscope f}) \simeq \texttt{f}\{[\beta]\}$, and thus $(\texttt{nscope f}) \fallingdotseq (\texttt{f} : [\texttt{b}] \to [\texttt{b}])$.

Next, we define the conditional form of equivalence, called *respect of expressions*, based on the definition of feasibility and the $\simeq$ relation. It specifies that an AspectFun expression with advice predicates respects a corresponding FIL expression under a type-constrained context if they are equivalent ($\simeq$) after the advice predicates involved are properly realized by feasible expressions.

Definition 7 (Respect of Expressions) *Given an AspectFun program π, let e be an expression of π and $\mathcal{A}$ be the advice store derived from π. Suppose that $\Gamma \vdash e : \overline{p}.t \rightsquigarrow e'$, $fv(\overline{p}) \subseteq fv(t)$, and the environment Γ contains the set of predicates $\overline{(g : t_g \rightsquigarrow dg)}$ introduced via the* (Pred) *rule. We say that e' respects an FIL expression, e^I, under the pair of types $(\overline{p}.t, \tau)$, written as $(\overline{p}.t, \tau) \vdash e' \propto e^I$, if for all type substitutions S, S', and expressions $\overline{e_p}$, $\overline{e_g}$, such that $\overline{e_p} \Leftarrow S\overline{p}$, $\overline{e_g} \Leftarrow S\overline{(g : t_g)}$, and $St \overset{type}{\rightarrowtail} \mathsf{S}'\tau$, then $[\![[\overline{e_g}/\overline{dg}]e' \ \overline{e_p}]\!] \simeq (\mathsf{S}'e^I)$ under Γ and $\mathcal{A}$ holds.*

Now we can define the mutual agreement between the binding definitions in the static weaving environment and those in the FIL-environment and the advice store. It is specified in terms of the "respect of expression" relation for each form of binding declarations which may occur in an AspectFun program, such as global functions, global advices, and local functions.

Definition 8 (Respect of Bindings) *Given an* AspectFun *program π, bindings derived from π in a static weaving environment Γ is said to* respect *those in an* FIL*-converting environment, Δ, and an advice store, $\mathcal{A}$, written as $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, if*

(1) $\Gamma \propto (\Delta, \mathcal{A})$,

(2) for all $x : \forall\overline{a}.\overline{p}.t \rightsquigarrow x \in \Gamma$ introduced via (Global) *as a global variable and $x : \forall\overline{\alpha}.\tau_x \in \Delta$, we have $(\overline{p}.t, \tau_x) \vdash Body(x) \propto body^I(x)$,*

(3) for all $g : \forall\overline{a}.\overline{p}.t \rightsquigarrow g \in \Gamma$ introduced via (Global) *as a global function and $g : \forall\overline{\alpha}.\tau_x \rightarrow \tau_g \in \Delta$, we have $(\overline{p}.t, \tau_x \rightarrow \tau_g) \vdash Body(g) \propto \lambda x : \tau_x.body^I(g)$,*

(4) for all advice $n : \forall\overline{a}.\overline{p}.t \bowtie f \in \Gamma$ and $(n : \forall\overline{\alpha}.\tau_x \rightarrow \tau_n, \overline{pc}, \tau_x, Body^I(n)) \in \mathcal{A}$, we have $(\overline{p}.t, \tau_x \rightarrow \tau_n) \vdash Body(n) \propto \lambda x : \tau_x.body^I(n)$,

(5) for all $x : \forall\overline{a}.\overline{p}.t \rightsquigarrow x \in \Gamma$ introduced via (Let) *and $x : \sigma_x \in \Delta$, we have $(\overline{p}.t, \tau_x) \vdash Body(x) \propto body^I(x)$,*

(6) there are no constraints on those bindings introduced via (Abs) *or* (Pred).

Based on the above definitions, we can proceed to develop the intermediate results that will lead the the correctness of static weaving. The first result is that there is a direct correspondence between the static weaving derivations and the FIL-conversions. In particular, the types derived in the two systems follow the type conversion defined in Section 3. Indeed, ignoring the code emitting part, the FIL conversion rules listed in figures 5 and 6 are simply a conservative extension of the Hindley-Milner type inference rules with explicit type abstraction and application.

We present the result as two theorems. The first theorem concerns the correspondence of expression types in two systems under proper program environments.

Theorem 1 (Correspondence of Expression Types) *If* $\Gamma \propto (\Delta, \mathcal{A})$ *and* $\Gamma \vdash e : \rho \rightsquigarrow e'$*, then we can obtain a corresponding* FIL*-conversion* $\Delta \vdash e : \tau \rightarrowtail e^I$ *such that* $\rho \overset{type}{\rightarrowtail} \tau$*.*

The second theorem further extends the correspondence to the types of an AspectFun program, including top-level declarations.

Theorem 2 (Correspondence of Program Types) *If* $\vdash \pi : t \rightsquigarrow e'$*, then there is a derivation* $\emptyset \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}$ *and* $t \overset{type}{\rightarrowtail} \tau$

Next, we investigate the correspondence between the expressions produced by static weaving and those by the FIL-conversion. Essentially, the main target here is the chain expressions, which are the other core products of our static weaving scheme besides advice predicates. In particular, a key step towards proving the correctness of our static weaving scheme is that the chain expression assembled by (Var-A) rule "respects" the FIL expression of applying the underlying advised function (associated with the variable expression operated by (Var-A)) to the types in context. We accomplished this step via the following two lemmas about advice chaining and chain expansion. Before stating the lemmas, we define some auxiliary functions to specify advice names and type substitutions involved in a chain expression.

Definition 9 (AdviceName and AdviceSet)

$$
\begin{array}{lcl}
\mathit{AdviceSet}(\lambda \overline{y}.\langle f\ \overline{y}\ , \{\overline{e}\}\rangle) & = & \{\mathit{AdviceName}(e_i) \mid e_i \in \overline{e}\} \\
\mathit{AdviceName}(e) & = & \mathsf{case}\ e\ \mathsf{of} \\
 & & \quad n\ \overline{dg} \rightarrow n \\
 & & \quad \langle n\ \overline{dg}\ , \{\overline{adv}\}\rangle \rightarrow n \\
\mathit{AdviceUnifiers}(e, \tau) & = & \mathit{let}\ (\mathit{AdviceName}(e) : \forall \overline{\alpha}.\tau_n, ...) \in \mathcal{A} \\
 & & \quad\quad [\overline{\tau}/\overline{\alpha}]\tau_n = \tau \\
 & & \quad \mathit{in}\ \overline{\tau}
\end{array}
$$

The first lemma shows that the set of advices selected by (Var-A) rule is the same as those returned by *Choose* function of the FIL operational semantics.

Lemma 1 (Advice Selection) *If* $\Gamma \propto (\Delta, \mathcal{A})$ *and* $\Gamma \vdash e_0 : \overline{p}.t \rightsquigarrow \lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle$ *then*

(1) *there exist some types* τ_1*,* τ_2 *and* $\overline{\tau}$ *such that* $\Delta \vdash e_0 : \tau_1 \rightarrow \tau_2 \rightarrowtail f\{\overline{\tau}\}$

(2) *for any type substitutions* S *and* S' *that satisfy* $St \overset{type}{\rightarrowtail} \mathsf{S}'(\tau_1 \rightarrow \tau_2)$*, we have* $\mathit{AdviceSet}(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle) = \mathit{Names}(\mathit{Choose}(f, \mathsf{S}'\tau_1))$ *where* $\mathit{Names}(s) = \{n \mid (n : \varsigma_n, \cdots) \in s\}$*.*

The second lemma shows that the chain expression assembled by (Var-A) rule respects the corresponding FIL-converted expression under the program context.

Lemma 2 (Respect of Chain Expressions) *Suppose that* $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$*. Let* f *be an advised function or advice that* $\Delta(f) = \forall \overline{\alpha}.\tau_f$*. If* $\Gamma \vdash f : \overline{p}.t \rightsquigarrow \lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle$*,*

$fv(\overline{p}) \subseteq fv(t)$, *and* $(t, \mathsf{S}\tau_f) \vdash e_i \propto \mathit{AdviceName}(e_i)\{\mathit{AdviceUnifiers}(e_i, \mathsf{S}\tau_f)\}$, *then* $(\overline{p}.t, \mathsf{S}\tau_f) \vdash \lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle \propto f\{\overline{\tau}\}$ *where* $\mathsf{S} = [\overline{\tau}/\overline{\alpha}]$ *and* $t \overset{type}{\rightarrowtail} \mathsf{S}\tau_f$.

The following theorem shows that, given an AspectFun program π with respectful binding environments, any expressions of π transformed by static weaving will also respect the corresponding FIL-converted expressions.

Theorem 3 (Soundness of Expression Weaving) *If* $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $\Gamma \vdash e : \overline{p}.t \rightsquigarrow e'$, *and* $fv(\overline{p}) \subseteq fv(t)$, *then there exists an* e^I *such that* $\Delta \vdash e : \tau \rightarrowtail e^I$ *and* $(\overline{p}.t, \tau) \vdash e' \propto e^I$

Finally, the correctness of our static weaving scheme is established by the following theorem.

Theorem 4 (Soundness of Static Weaving) *Let* π_0 *be an AspectFun program. If* $\emptyset \vdash \pi_0 : t \rightsquigarrow e'$, *then there exists an FIL-converted program,* $\pi^I = (\mathcal{A}, e^I)$, *such that* $\pi_0 \overset{prog}{\rightarrowtail} \pi^I$ *and* $[\![e']\!] \sim\sim e^I$.

6. 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 a 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. [Masuhara et al. 2003]. 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 uses of a variant of control flow keyword – `cflow`– and type-scoped control flow (eg. `cflow(f(_ :: Int))`).
(2) Installing state-based mechanism in woven code, including insertion of dynamic tests.
(3) Analyzing and optimizing woven code produced at step (2) to compile away as many dynamic tests as possible.

After presenting the formal semantics of control-flow based pointcut, we shall describe these steps in more detail in the rest of this section.

6.1 Semantics of control-flow based pointcut

The semantics of control-flow based pointcut is defined by modifying the operational semantics for FIL introduced in section 2. The modification is so small that existing conditions or proofs based on the new definitions are all valid with minor modification.

First, We modify the reduction-based big-step operational semantics function $\Downarrow_{\mathcal{A}}$ to carry a *stack* $\mathcal{S}$, written as $\Downarrow^{\mathcal{S}}_{\mathcal{A}}$, denoting that the progress is done under stack

environment $\mathcal{S}$. $\mathcal{S}$ is a list of function names capturing the stack of nested calls (represented by the corresponding function names) that have been invoked but not yet returned at the point of reduction.

Second, two intermediate expression constructions, *stack closure* and *stack frame* are added. Stack closure is written as $(\!|e, \mathcal{S}|\!)$, it means that e should be evaluated under the stack $\mathcal{S}$, ignoring the currently active stack. This is needed since we adopt lazy semantics for AspectFun. Note that applying a substitution ($[e'/x]$) on a stack closure affects only the expression associated with the closure, leaving the stack unchanged: $[e'/x](\!|e, \mathcal{S}|\!) = (\!|[e'/x]e, \mathcal{S}|\!)$. Stack frame, written as $\lambda^f x : \tau_x.e^I$, is updated in (OS:App) and used in *Trigger*. It enables e^I to be evaluated under current stack with f pushed. The difference between annotated lambda and stack frame is that type information is not carried in a stack frame. Thus, Stack frame is also a value:

$$\text{Values } v^I ::= c \mid \lambda^{jp} x : \tau_x.\ e^I \mid \lambda^f x : \tau_x.\ e^I \mid \Lambda\alpha.\ e^I$$

Most rules remain unchanged except that all occurrences of $\Downarrow$'s are replaced by $\Downarrow^{\mathcal{S}}$'s. Rule (OS:App) is changed into two rules, the invocations of which depend on the wether the function applied is an advised function.

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

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

Rule (OS:Let), which is often thought of as a syntactic sugar for the underlying lambda application, changes in similar fashion as (OS:App):

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

Finally a new rule for closure evaluation is needed:

$$\text{(OS:Clos)}\ \frac{e^I \Downarrow^{\mathcal{S}} v^I}{(\!|e^I, \mathcal{S}|\!) \Downarrow^{\mathcal{S}'} v^I}$$

As (OS:App) is changed, the auxiliary functions *Trigger*, *Weave*, *Choose* and *JPMatch* are changed accordingly. Specifically, they now need to look up for the join points existing the stack $\mathcal{S}$ when selecting the appropriate advices to be woven.

$$
\begin{array}{lll}
\mathit{Trigger}' & : & e^I \times jp \times \mathit{stack} \to e^I \\
\mathit{Trigger}'(e^I, \epsilon, \mathcal{S}) & = & e^I \\
\mathit{Trigger}'(\lambda^{f:\tau_f} x:\tau_x.\ e^I, f:\tau_f, \mathcal{S}) & = & \mathit{Weave}'(\lambda^{f:\tau_f} x:\tau_x.\ e^I, \tau_f, \mathcal{S}, \mathit{Choose}'(f, \tau_x, \mathcal{S}))
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Weave}' & : & e^I \times \tau \times \mathit{stack} \times \overline{\mathrm{Adv}} \to e^I \\
\mathit{Weave}'(e^I, \tau_f, \mathcal{S}, []) & = & e^I \\
\mathit{Weave}'(e^I_f, \tau_f, \mathcal{S}, a:\mathit{advs}) & = & \text{Let } (n : \forall\overline{\alpha}.\ \tau_n, \overline{pc}, \tau, \Lambda\overline{\alpha}.\ e^I) = a \\
 & & \quad \text{In } \text{If } \neg(\tau_n \trianglerighteq \tau_f) \text{ Then } \mathit{Weave}(e^I_f, \tau_f, \mathit{advs}) \\
 & & \quad\quad \text{Else Let } \overline{\tau} \text{ be types such that } [\overline{\tau}/\overline{\alpha}]\tau_n = \tau_f \\
 & & \quad\quad\quad (e^I_p,\ e^I_a) = (\mathit{Weave}'(e^I_f, \tau_f, \mathcal{S}, \mathit{advs}), (\Lambda\overline{\alpha}.\ e^I)\{\overline{\tau}\}) \\
 & & \quad\quad\quad \lambda^{n:\tau_n} x:\tau_x.\ e^I_n = [e^I_p/\texttt{proceed}]e^I_a \\
 & & \quad\quad \text{In } \mathit{Trigger}'(\lambda^n x:\tau_x.\ e^I_n, n:\tau_n, \mathcal{S})
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Choose}'(f, \tau, \mathcal{S}) = & \{(n_i : \varsigma_i, \overline{pc_i}, \tau_i, e^I_i) \mid & (n_i : \varsigma_i, \overline{pc_i}, \tau_i, e^I_i) \in \mathcal{A},\ \tau_i \trianglerighteq \tau, \\
 & & \quad \exists pc \in \overline{pc_i}\ s.t.\ \mathit{JPMatch}'(f, pc, \mathcal{S})\} \\
\mathit{JPMatch}'(f, pc + \texttt{cflowbelow}(g), \mathcal{S}) & = & \mathit{JPMatch}'(f, pc, \mathcal{S}) \wedge\ g \in \mathcal{S} \\
\mathit{JPMatch}'(f, pc - \texttt{cflowbelow}(g), \mathcal{S}) & = & \mathit{JPMatch}'(f, pc, \mathcal{S}) \wedge\ g \notin \mathcal{S} \\
\mathit{JPMatch}'(f, pc, \mathcal{S}) & = & \mathit{JPMatch}(f, pc)
\end{array}
$$

6.2 De-sugaring

This step aims to transform source programs into ones that are amenable to static type inference and weaving. Specifically, type-scoped control flow (eg. `cflow(f(_ :: Int))`) and `cflow` can be considered as 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 in
n@advice around {k + cflow(n')} (arg) = ...
```

Note that, with the help of second-order advice, `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 translate all `cflow`-pointcuts into pointcuts involving `cflowbelow`. Doing so reduces the number of cases to be considered during compilation. The 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 the 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.

6.3 State-based Implementation

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

Example 7

```
// 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)
```

The first (comment) line in the code above displays a meta-data structure capturing the association of the advice `n` with the `cflowbelow` pointcut `k+cflowbelow(g)`.[5] This implies that dynamic testing is needed at calls to function `k` to determine if `n` should be invoked; ie., whether `k` is called in the context of a call to `g`. We call `g` the "`cflowbelow` *advised function*", while `k` is simply called an "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 kinds 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 7, 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* [Jones 1995]. In pseudo-code format, the encoding of `g` will be as follows:[6]

[5] During static weaving, advice `n` is assumed to have `k` as its pointcut, instead of `k+cflowbelow(g)`.
[6] 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.

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

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

Next, uses of `cflowbelow` advices appear in various chain expressions; eg., `<k,{n}>` occurs in two places in Example 7. For these uses, we insert code to perform lookups 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 7 is as follows:

Example 7a

```
// 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`.

6.4 Control-Flow Pointcut Analysis and Optimization

From Example 7a, 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. [Avgustinov et al. 2005] that such optimization and its associated analysis can be more effectively performed on the woven code. In our system, we employ two interprocedural analyses to determine the opportunity for optimizing guarded expressions. These are **mayCflow** and **mustCflow** analysis (cf. [Avgustinov et al. 2005]).

Since the subject language is polymorphically typed and higher-order, we adopt an *annotated-type and effect* system for our analyses. This approach has been described in detail in [Nielson et al. 1999]. 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 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 contexts, but the former captures all contexts in which e may be evaluated, whereas the latter captures those in which e must be invoked.

6.4.1 *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 = e  in
f x = x in
g x = x + 1 in
g (f 3)
```

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)`, according to the revised semantic rules (OS:App') and (OS:Clos), the sub-expression `(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 actual 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 analyses are oblivious to the call semantics of the language.

6.4.2 *The Analysis and Optimization Details.* Figure 13 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 second rule above indicates 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 13 together yield a set of constraints over context variables φ. The least solution of the constraints is the one containing the most information.

In **mustCflow** analysis, the directions of set inclusions in rule (App) and (Subt1) are inversed and the solution of the constraints which contains the most information is the greatest one.

$$(\textsc{Const})\ \hat{\Gamma} \vdash_{\texttt{may}} c : \hat{\tau}_c \,\&\, \varphi \qquad (\textsc{Var})\ \hat{\Gamma} \vdash_{\texttt{may}} x : \Gamma(x) \,\&\, \varphi$$

$$(\textsc{Lambda})\ \frac{\hat{\Gamma}.x : \hat{\tau}_x \vdash_{\texttt{may}} e : \hat{\tau}_e \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \lambda x.e : \hat{\tau}_x \xrightarrow{\varphi'} \hat{\tau}_e \,\&\, \varphi} \quad (\textsc{App})\ \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \hat{\tau}_2 \xrightarrow{\varphi_F} \hat{\tau} \,\&\, \varphi \quad \neg \mathit{guarded}(e_1) \qquad \hat{\Gamma} \vdash_{\texttt{may}} e_2 : \hat{\tau}_2 \,\&\, \varphi \quad \varphi \subseteq \varphi_F}{\hat{\Gamma} \vdash_{\texttt{may}} (e_1\ e_2) : \hat{\tau} \,\&\, \varphi}$$

$$(\textsc{Let})\ \frac{\hat{\Gamma}.x : \hat{\tau}_1 \vdash_{\texttt{may}} e_1 : \hat{\tau}_1 \,\&\, \varphi \qquad \hat{\Gamma}.x : \hat{\tau}_1 \vdash_{\texttt{may}} e_2 : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \hat{\tau} \,\&\, \varphi} \quad (\textsc{If})\ \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \mathit{Bool} \,\&\, \varphi \qquad \hat{\Gamma} \vdash_{\texttt{may}} e_2 : \hat{\tau} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e_3 : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \hat{\tau} \,\&\, \varphi}$$

$$(\textsc{Decl})\ \frac{\hat{\Gamma}.(f : \hat{\tau}_x \xrightarrow{\varphi_F} \hat{\tau}').(x : \hat{\tau}_x) \vdash_{\texttt{may}} e_f : \hat{\tau}' \,\&\, \varphi_F \cup \{\mathrm{f}\} \qquad \hat{\Gamma}.(f : \hat{\tau}_x \xrightarrow{\varphi_F} \hat{\tau}') \vdash_{\texttt{may}} p : \hat{\tau} \,\&\, \emptyset}{\hat{\Gamma} \vdash_{\texttt{may}} f\ x = e_f\ \mathsf{in}\ p : \hat{\tau} \,\&\, \emptyset}$$

$$(\textsc{Gd-app})\ \frac{\hat{\Gamma} \vdash_{\texttt{may}} e_1 : \mathit{Bool} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} e_3 : \hat{\tau} \,\&\, \varphi \quad \hat{\Gamma} \vdash_{\texttt{may}} (e_2\ e_3) : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \texttt{<|}\, e_1, e_2 \,\texttt{|>}\ e_3 : \hat{\tau} \,\&\, \varphi}$$

$$(\textsc{Chain})\ \frac{\hat{\Gamma} \vdash_{\texttt{may}} [\![\langle e\,, \{e_1, \ldots, e_n\}\rangle]\!] : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} \langle e\,, \{e_1, \ldots, e_n\}\rangle : \hat{\tau} \,\&\, \varphi} \quad (\textsc{Subs})\ \frac{\hat{\Gamma} \vdash_{\texttt{may}} e : \hat{\tau} \,\&\, \varphi}{\hat{\Gamma} \vdash_{\texttt{may}} e : \hat{\tau}' \,\&\, \varphi} \quad \text{if } \hat{\tau} \leq \hat{\tau}'$$

$$(\textsc{Subt1})\ \frac{\hat{\tau}_1' \leq \hat{\tau}_1 \quad \hat{\tau}_2 \leq \hat{\tau}_2' \quad \varphi' \subseteq \varphi}{\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \leq \hat{\tau}_1' \xrightarrow{\varphi'} \hat{\tau}_2'} \qquad (\textsc{Subt2})\quad \hat{\tau} \leq \hat{\tau}$$

Fig. 13. **mayCflow** inference rules

Applying the analysis over the woven code given in Example 7a, we obtain the following contexts for the body of each of the functions:

$$\begin{array}{ll} \varphi_k^{\texttt{may}} = \{\mathtt{f}, \mathtt{g}\} & \text{May-context for body of } \mathtt{k} \\ \varphi_g^{\texttt{may}} = \{\mathtt{f}\} & \text{May-context for body of } \mathtt{g} \\ \varphi_f^{\texttt{may}} = \emptyset & \text{May-context for body of } \mathtt{f} \end{array}$$

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 point out the resulting contexts produced by performing **mustCflow** analysis over the same example:

$$\begin{array}{ll} \varphi_k^{\texttt{must}} = \emptyset & \text{Must-context for body of } \mathtt{k} \\ \varphi_g^{\texttt{must}} = \{\mathtt{f}\} & \text{Must-context for body of } \mathtt{g} \\ \varphi_f^{\texttt{must}} = \emptyset & \text{Must-context for body of } \mathtt{f} \end{array}$$

After collecting all the **mayCflow**and **mustCflow** information, we perform optimization over the woven code 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. In the grey area that $f \in \varphi^{\texttt{may}}$ but $f \notin \varphi^{\texttt{must}}$, it takes no advantage of our analyses. The guard is still necessary. This happens when a function is called from different paths and some of them make the control flow condition satisfied but some do not.

Going back to Example 7a, we are thus able to eliminate all the guarded expressions, yielding the following woven code:

Example 7b

```
// 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.

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

Example 8

```
// 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 $Int \xrightarrow{\{\texttt{f}\}} Int$ in **mustCflow** analysis, making `n` to be statically woven into it. Furthermore, the one used as an argument of `g` has annotated type $Int \xrightarrow{\{\texttt{g}\}} Int$ in **mayCflow** analysis, indicating that

`f` is not in the context when the call to `g` is invoked; 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,{}>)
```

7. RELATED WORK

7.1 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 [Dantas et al. 2007; 2005] and Aspectual Caml [Masuhara et al. 2005], 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 an aspect-oriented functional language.

AspectML [Dantas et al. 2007; 2005] 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 run-time type analysis is performed to handle the matching of type-scoped pointcuts. This completely 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 [Masuhara et al. 2005], 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 clean 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 lexical 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 lexical weaving scheme to handle nested advices, second-order advices or control flow based pointcuts such as cflow.

7.2 Type-Scoped Programming

Our type-directed translation was originally inspired by the dictionary translation of Haskell type classes [Wadler and Blott 1989]. A number of subsequent applications of type classes [Lewis et al. 2000; Jones 1999] 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 aspect-oriented functional languages with Haskell type classes [Sulzmann and Wang 2007]. The encoding is light-weight and allows easy integration with existing advanced language features such as type classes and GADTs [Peyton Jones et al. 2005]. In that work, all candidate advices are piled up at function calls and correct advice chainings are done implicitly by 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.

In another dimension, Washburn and Weirich demonstrate type-directed programming in AspectML [Washburn and Weirich 2006]. They show that, with a run-time type check mechanism, aspects can be used as an alternative of type classes and this alternative approach can even perform better in cases where type classes struggle.

7.3 Static Optimization

The implementation and optimization of AspectFun took inspirations from the AspectBench Compiler for AspectJ (ABC) [Avgustinov et al. 2005]. ABC implements a series of optimizations which significantly improve 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 graph 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 [Masuhara et al. 2003], Masuhara, Kiczales and Dutchyn propose a compilation and optimization model for aspect-oriented programs. Their approach employs partial evaluation to optimize an evaluator for aspect-oriented languages implemented in Scheme. The limited power of their 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.

8. CONCLUSION

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. This paper consolidates our previous research results [Wang et al. 2006b; 2006a; Chen et al. 2007], and makes several significant revisions and extensions to multiple dimensions of our research. Not only do we provide a complete treatment to several advanced features in AspectFun, we also present the compilation process in full detail. Above all, we provide a formal account of the correctness of our static typing and weaving rules with respect to the operational semantics of AspectFun.

Moving ahead, we intend to 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 also investigate additional optimization techniques and conduct empirical experiments of performance gain.

We also would like to investigate the use of aspects to capture side-effecting computation for AspectFun. This is particularly promising, as our compilation model automatically converts the base program to monadic form.

On another frontier, we plan to explore the possibility of 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 [Jagadeesan et al. 2006]

```
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 [Jagadeesan et al. 2006], the above aspect cannot be handled by their aspect language because the type-erasure semantics of Java prohibits any dynamic type test to be performed. We speculate that our type-directed weaving could be a key to the solution of the problem.

9. ACKNOWLEDGMENT

This research is partially supported by the National University of Singapore under research grant "R-252-000-252-112", and by the National Science Council, Taiwan, R.O.C. under grant number "NSC 95-2221-E-004-004-MY2".

REFERENCES

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 2005. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 117–128.

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

Dantas, D. S., Walker, D., Washburn, G., and Weirich, S. 2005. PolyAML: a polymorphic aspect-oriented functional programmming language. In *Proc. of ICFP'05*. ACM Press.

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

Jagadeesan, R., Jeffrey, A., and Riely, J. 2006. Typed parametric polymorphism for aspects. *Science of Computer Programming 63,* 3, 267–296.

Jones, M. P. 1992. Qualified types: Theory and practice. Ph.D. thesis, Oxford University.

Jones, M. P. 1995. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*. 97–136.

Jones, M. P. 1999. Exploring the design space for type-based implicit parameterization. Tech. rep., Oregon Graduate Institute of Science and Technology.

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

Lewis, J. R., Shields, M., Launchbury, J., and Meijer, E. 2000. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*. 108–118.

Masuhara, H., Kiczales, G., and Dutchyn, C. 2003. A compilation and optimization model for aspect-oriented programs. In *CC*. 46–60.

Masuhara, H., Tatsuzawa, H., and Yonezawa, A. 2005. Aspectual Caml: an aspect-oriented functional language. In *Proc. of ICFP'05*. ACM Press.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

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

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

Sereni, D. and de Moor, O. 2003. Static analysis of aspects. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, M. Aksit, Ed. ACM Press, 30–39.

Sulzmann, M. and Wang, M. 2007. Aspect-oriented programming with type classes. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*. ACM, New York, NY, USA, 65–74.

Tucker, D. B. and Krishnamurthi, S. 2003. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. ACM, 60–76.

Wang, M., Chen, K., and Khoo, S.-C. 2006a. On the pursuit of static and coherent weaving. In *Foundations of Aspect-Oriented Languages Workshop at AOSD 2006*. Iowa State University, TR 06-01, 37–46.

Wang, M., Chen, K., and Khoo, S.-C. 2006b. Type-directed weaving of aspects for higher-order functional languages. In *PEPM '06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press.

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

A. PROOF OF CORRECTNESS OF STATIC WEAVING

In this appendix, we detail the proof of the correctness of static weaving. After presenting a few straightforward propositions, we shall prove the key lemmas and theorems stated in Section 5. Note that, as in the main text, e is used to stand for both an AspectFun source expression and an expanded expression, e^I for an FIL expression, and e' for an intermediate expression produced by static weaving.

We begin by listing down some properties about FIL conversion and basic properties about open and close equivalence.

Proposition 1 *If* $\forall \overline{a}.\overline{p}.t \overset{type}{\rightarrowtail} \forall \overline{\alpha}.\tau$, *then* $\forall S, \exists \mathsf{S}'$ *such that* $St \overset{type}{\rightarrowtail} \mathsf{S}'\tau$

Proposition 2 *Given an* expanded expression e *and an* FIL *annotated lambda expression* $\lambda^{f:\tau_f} x:\tau_x.e^I$, *if* $e \sim\sim$ *Trigger*$(\lambda x:\tau_x.e^I, f:\tau_f)$, *then* $e \sim\sim \lambda^{f:\tau_f} x:\tau_x.e^I$.

Proposition 2a *Given an* expanded expression e *and an* FIL *annotated lambda expression* $\lambda^{f:\tau_f} x:\tau_x.e^I$, *if* $e \simeq$ *Trigger*$(\lambda x:\tau_x.e^I, f:\tau_f)$, *then* $e \simeq \lambda^{f:\tau_f} x:\tau_x.e^I$.

Proposition 3 *Suppose* $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$. *If* x *is a non-lambda-bound identifier in* Γ, *then*

$$e \simeq e^I \textit{ under } \Gamma \textit{ and } \mathcal{A} \Leftrightarrow [Body'(x)/x]e \simeq [Body^I(x)/x]e^I \textit{ under } \Gamma_x \textit{ and } \mathcal{A}$$

where $Body'(x)$ *is the same as the one defined in Definition 4 and* Γ_x *is* Γ *with the binding of* x *removed.*

Proposition 4 *If* $e_1 \simeq e_1^I$ *then* $[e_1/\texttt{proceed}]e \simeq [e_1^I/\texttt{proceed}]e^I \Leftrightarrow e \simeq e^I$.

Theorem 1 (Correspondence of Expression Types) *If* $\Gamma \propto (\Delta, \mathcal{A})$ *and* $\Gamma \vdash e : \rho \rightsquigarrow e'$, *then there exists a corresponding* FIL*-conversion* $\Delta \vdash e : \tau \rightarrowtail e^I$ *such that* $\rho \overset{type}{\rightarrowtail} \tau$.

Proof. By induction on the height of the derivation tree for $\Gamma \vdash e : \rho \rightsquigarrow e'$. □

Theorem 2 (Correspondence of Program Types) *If* $\vdash \pi : t \rightsquigarrow e'$, *then there is an* FIL*-conversion* $\emptyset \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}$ *such that* $t \overset{type}{\rightarrowtail} \tau$

Proof. This can be proved by induction on the length of declarations in π using a stronger hypothesis which is detailed in the proof of Theorem 4. □

Lemma 1 (Advice Selection) *If* $\Gamma \propto (\Delta, \mathcal{A})$ *and* $\Gamma \vdash e_0 : \overline{p}.t \rightsquigarrow \lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle$, *then*

(1) *there exist some types* τ_1, τ_2 *and* $\overline{\tau}$ *such that* $\Delta \vdash e_0 : \tau_1 \rightarrow \tau_2 \rightarrowtail f\{\overline{\tau}\}$

(2) *for any type substitutions* S *and* S*' that satisfy* $St \overset{type}{\rightarrowtail} \mathsf{S}'(\tau_1 \rightarrow \tau_2)$, *we have* $AdviceSet(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle) = Names(Choose(f, \mathsf{S}'\tau_1))$ *where* $Names(s) = \{n \mid (n : \varsigma_n, \cdots) \in s\}$.

PROOF.

(1) We prove the first part by showing that the translation in the lemma is derived via (VAR-A) rule and e_0 is simply an advised function, f. This is proved by induction on number of predicates in $\overline{p}$ as follows.
If $\overline{p}$ is empty, then clearly the translation is just an instance of (VAR-A) rule due to the presence of advice chain. Suppose that when $\overline{p}$ contains n predicates, it is still an instance of (VAR-A). Now assume that there are $n+1$ predicates in $\overline{p}$. If the outermost predicate is generated by applying the (PRED) rule, then we have the derivation, $\Gamma.x : t \rightsquigarrow x_t \vdash e : \rho \rightsquigarrow e'_t$ where ρ has n predicates and t is an instance of the type scheme of x. Obviously, $\Gamma.x : t \propto (\Delta, \mathcal{A})$. Thus, by the induction hypothesis, this translation is derived via (VAR-A) rule and $e \equiv x$ is an advised function. This contradicts the condition of the (PRED) rule. Since no other rule produces a translation with the given form of predicated typed and chain expression except (VAR-A), the translation in the lemma is indeed an instance of (VAR-A), and e_0 is simply an advised function, f.

(2) We prove the second part in two steps:
(a) if $\Gamma \vdash f : \overline{p}.t \rightsquigarrow e_c$ and $\Gamma \vdash f : S(\overline{p}.t) \rightsquigarrow e'_c$, then $\mathit{AdviceSet}(e_c) = \mathit{AdviceSet}(e'_c)$.
(b) if $\Gamma \vdash f : S(\overline{p}.t) \rightsquigarrow \lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e'}\}\rangle$ and $St \overset{\text{type}}{\rightarrowtail} \mathsf{S}'(\tau_1 \rightarrow \tau_2)$, then $\mathit{Names}(\mathit{Choose}(f, \mathsf{S}'\tau_1)) = \mathit{AdviceSet}(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e'}\}\rangle)$
Combining these two steps yields the set equality in the lemma.

(a) According to the premise of (VAR-A), it is obvious that $\mathit{AdviceSet}(e_c) \subseteq \mathit{AdviceSet}(e'_c)$. Hence we only show that $\mathit{AdviceSet}(e'_c) \subseteq \mathit{AdviceSet}(e_c)$. This is done by contradiction.
Assume that there exists an advice binding $n : \forall\overline{b}.\overline{q}.t_n \bowtie f$ such that $t_n \trianglerighteq St$ but $t_n \not\trianglerighteq t$. Let $t_1 \rightarrow t_2 = t$ and $t_k \rightarrow t_{n_k} = t_n$. By $t_n \trianglerighteq St$, we have $t_k \trianglerighteq St_1$, which in turn implies that t_k and t_1 are unifiable. So, by the condition $wv(f : t_1 \rightarrow t_2)$, we have $t_n \trianglerighteq t$. This contradicts the assumption.
Since no such advice exists, we conclude that $\mathit{AdviceSet}(e'_c) \subseteq \mathit{AdviceSet}(e_c)$.

(b) Let $\mathit{Aset} = \mathit{AdviceSet}(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e'}\}\rangle)$, $\mathit{Cset} = \mathit{Names}(\mathit{Choose}(f, \mathsf{S}'\tau_1))$, $t'_1 \rightarrow t'_2 = St$ and $\tau'_1 \rightarrow \tau'_2 = \mathsf{S}'(\tau_1 \rightarrow \tau_2)$. By $\Gamma \propto (\Delta, \mathcal{A})$,

$$n_i : \forall\overline{b}.\overline{q}.t_i \rightarrow t_{n_i} \bowtie f \in \Gamma \ \Leftrightarrow$$
$$(n_i : \forall\overline{\alpha}.\tau_x \rightarrow \tau_{n_i}, \overline{pc_i}, \tau_i, e_i^I) \in \mathcal{A} \wedge \exists pc \in \overline{pc_i}.\ \mathit{JPMatch}(f, pc) \equiv \mathsf{true}$$

Let us consider the advice selection criteria for both Aset and Cset. For advice n_i to be selected in Aset, (VAR-A) requires that $t_i \rightarrow t_{n_i} \trianglerighteq t'_1 \rightarrow t'_2$. By contrast, Cset requires that $\tau_i \trianglerighteq \tau'_1$ according to $\mathit{Choose}(f, \tau'_1)$.
First, it is easy to see that $\mathit{Aset} \subseteq \mathit{Cset}$. By $\Gamma \propto (\Delta, \mathcal{A})$, we have $t_i \overset{\text{type}}{\rightarrowtail} \tau_i$. Given $t_i \rightarrow t_{n_i} \trianglerighteq t'_1 \rightarrow t'_2$ and $t'_1 \rightarrow t'_2 \overset{\text{type}}{\rightarrowtail} \tau'_1 \rightarrow \tau'_2$, it is obvious that $t_i \trianglerighteq t'_1 \Leftrightarrow \tau_i \trianglerighteq \tau'_1$. Thus, $\mathit{Aset} \subseteq \mathit{Cset}$.
Second, we show that $\mathit{Cset} \subseteq \mathit{Aset}$ by assuming otherwise and get a contradiction due to $\Gamma \propto (\Delta, \mathcal{A})$. If there exists an advice n_k such that $n_k \in \mathit{Cset}$ but not Aset. Then, $\tau_k \trianglerighteq \tau'_1$ and $t_k \trianglerighteq t'_1$ since $t_k \overset{\text{type}}{\rightarrowtail} \tau_k$, but $S_1 t_{n_k} \not\trianglerighteq S_1 t'_2$

where $t'_1 = S_1 t_k$. Let $\forall \overline{a}.\overline{p}.t_{f_1} \to t_{f_2} = \Gamma(f)$. Consider the kinds of advice binding for n_k.
(i) (ADV): By the condition of (ADV), $t_k \to t_{n_k} \trianglerighteq t_{f_1} \to t_{f_2}$, which in turn implies that $t_{n_k} \trianglerighteq t'_2$ since $t_{f_1} \to t_{f_2} \trianglerighteq St$. This contradicts the assumption.
(ii) (ADV-AN): By the condition of (ADV-AN), there exists a substitution S_0 such that

$$t_k = S_0 t_{f_1} \quad (1)$$

$$\text{and} \qquad t_{n_k} \trianglerighteq S_0 t_{f_2} \quad (2)$$

Then $t'_1 = S_1 S_0 t_{f_1}$. Now, since $t_{f_1} \to t_{f_2} \trianglerighteq t'_1 \to t'_2$, there exists a substitution S_2 such that $t'_2 = S_2 S_1 S_0 t_{f_2}$. Hence by (2),

$$S_1 t_{n_k} \trianglerighteq S_1 S_0 t_{f_2} \trianglerighteq S_2 S_1 S_0 t_{f_2} = t'_2$$

This contradicts the assumption.
Since no such advice exists, we conclude that $Cset \subseteq Aset$.

□

Lemma 2 (Respect of Chain Expressions) *Suppose that* $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$. *Let* f *be an advised function or advice that* $\Delta(f) = \forall \overline{\alpha}.\tau_f$. *If* $\Gamma \vdash f : \overline{p}.t \rightsquigarrow \lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle$, $fv(\overline{p}) \subseteq fv(t)$, *and* $(t, \mathsf{S}\tau_f) \vdash e_i \propto \mathit{AdviceName}(e_i)\ \{\mathit{AdviceUnifiers}(e_i, \mathsf{S}\tau_f)\}$ *for all* $e_i \in \overline{e}$, *then* $(\overline{p}.t, \mathsf{S}\tau_f) \vdash \lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle \propto f\{\overline{\tau}\}$ *where* $\mathsf{S} = [\overline{\tau}/\overline{\alpha}]$ *and* $t \overset{type}{\rightarrowtail} \mathsf{S}\tau_f$.

PROOF. Let $\forall \overline{a}.\overline{p_f}.t_f = \Gamma(f)$ and $S_0 t_f = t$. Then, by $\Gamma \propto (\Delta, \mathcal{A})$ and Proposition 1, such S exists.

Next, we proceed to prove the lemma according to the requirements stated in Definition 7 (respect of expressions). Specifically, given S, S', $\overline{e_p}$, and $\overline{e_g}$ such that $St \overset{\text{type}}{\rightarrowtail} \mathsf{S}'\mathsf{S}\tau_f$, $\overline{e_p} \precapprox S\overline{p}$ and $\overline{e_g} \precapprox S\overline{(g : t_g \rightsquigarrow dg)}$ as in Definition 7, we need to prove that

$$[\overline{e_g}/\overline{dg}][\![(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle)\ \overline{e_p}]\!] \simeq \mathsf{S}'(f\{\overline{\tau}\})$$

By Proposition 3 and definition of $Body^I(f)$, it is equivalent to show that

$$[[\![Body(f)]\!]/f][\overline{e_g}/\overline{dg}][\![(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle)\ \overline{e_p}]\!] \simeq \mathsf{S}'\mathsf{S}(\lambda^{f:\tau_f} x : \tau_x.body^I(f))$$

Let $\mathsf{S}'' = \mathsf{S}' \circ \mathsf{S}$. By applying Proposition 2a on the RHS, it suffices to show that

$$[[\![Body(f)]\!]/f][\overline{e_g}/\overline{dg}][\![(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle)\ \overline{e_p}]\!] \simeq \mathit{Trigger}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(body^I(f)), f : \mathsf{S}''\tau_f)$$

By the definition of $\mathit{Trigger}(\cdot)$, we can rewrite the RHS of the above equation to

$$\mathit{Weave}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(body^I(f)), \mathsf{S}''\tau_f, \mathit{Choose}(f, \mathsf{S}''\tau_x))$$

According to Lemma 1, $\mathit{AdviceSet}(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle) = \mathit{Names}(\mathit{Choose}(f, \mathsf{S}''\tau_x))$.

Thus, it suffices to show that

$$[[\![Body(f)]\!]/f][\overline{e_g}/\overline{dg}][\![(\lambda \overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle)\ \overline{e_p}]\!] \simeq \mathit{Weave}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(body^I(f)), \mathsf{S}''\tau_f, \{\mathcal{A}(n_i) \mid n_i \leftarrow \mathit{AdviceSet}(e_c)\}) \quad (3)$$

where $e_c \equiv \lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\overline{e}\}\rangle$. We prove the above open equation (3) by mathematical induction on the rank of e_c and then the length of $\overline{e}$.

Induction basis for the rank $\mathit{rank}(e_c) = 1$:

When $\mathit{rank}(e_c) = 1$, every expression in $\overline{e}$ has rank zero. We do induction again on the length of $\overline{e}$.

Induction basis for the length: $|\overline{e}| = 0$

In this case, $e_c \equiv \lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\}\rangle$. We proceed by reducing both sides of (3).

$$\begin{array}{rcl}
\text{LHS} & \equiv & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f][[\![(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\}\rangle)\ \overline{e_p}]\!] \\
& = & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f](\lambda\overline{dp}.f\ \overline{dp})\ \overline{e_p} \\
& \rightarrow_\beta^* & [\overline{e_g}/\overline{dg}][[\![Body(f)\ \overline{e_p}]\!] \\
& = & [\overline{e_g}/\overline{dg}][[\![(Body(f)\ \overline{e_p})]\!] \\
\text{RHS} & \equiv & \mathit{Weave}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(body^I(f)), \mathsf{S}''\tau_f, \{\}) \\
& = & \mathsf{S}''(\lambda x : \tau_x.body^I(f))
\end{array}$$

By $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $(\overline{p_f}.t_f, \tau_f) \vdash Body(f) \propto \lambda x : \tau_x.body^I(f)$. As the substitutions S and S' satisfy the condition in Definition 7, $SS_0t_f \overset{\text{type}}{\rightarrowtail} \mathsf{S}'\mathsf{S}\tau_f$, the LHS is thus open equivalent to the RHS. Hence (3) holds in this case.

Induction step for the length ($|\overline{e}| = m$):

Suppose that (3) holds for $|\overline{e}| = m-1$. Now let $\overline{e} = e_1, e_2, \cdots, e_m$ and $e_1 = n_1\ \overline{e_1'}$. We reduce the LHS of (3) as follows:

$$\begin{array}{rl}
& [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f][[\![(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{e_1, e_2, \cdots, e_m\}\rangle)\ \overline{e_p}]\!] \\
= & [\overline{e_g}/\overline{dg}][[\![\langle Body(f)\ \overline{e_p}\ , \{e_1, e_2, \cdots, e_m\}\rangle]\!] \\
& \mathit{rank}(e_i) = 0, \text{by chain expansion} \\
= & [\overline{e_g}/\overline{dg}]\mathit{proceedApply}(e_1, \langle Body(f)\ \overline{ep}\ , \{e_2, \cdots, e_m\}\rangle) \\
= & [\overline{e_g}/\overline{dg}][[\![n_1\ \overline{e_1'}\ \langle Body(f)\ \overline{ep}\ , \{e_2, \cdots, e_m\}\rangle]\!] \\
& \text{to continue, we apply Proposition 3 and replace } n_1 \text{ with } [\![\lambda\overline{dq}.\lambda\texttt{proceed}.\lambda y.body(n_1)]\!] \\
= & [\overline{e_g}/\overline{dg}][[\![(\lambda\texttt{proceed}.\lambda y.[\overline{e_1'}/\overline{dq}]body(n_1))\ \langle Body(f)\ \overline{ep}\ , \{e_2, \cdots, e_m\}\rangle]\!] \\
& \text{by substitution and } \beta - \mathit{reduction} \\
= & [\overline{e_g}/\overline{dg}][[\![\langle Body(f)\ \overline{e_p}\ , \{e_2, \cdots, e_m\}\rangle]\!]/\texttt{proceed}][[\![(\lambda y.[\overline{e_1'}/\overline{dq}]body(n_1))]\!]
\end{array}$$

Let $\Lambda\overline{\beta}.\lambda^{n_1:\tau_{n_1}} y : \tau_{y_0}.e_0^I = Body^I(n_1)$. We reduce the RHS of (3) according to the definitions of *Weave* (in Figure 8), $Body^I$ and $body^I$, and get

$$\begin{array}{rcl}
[\overline{\tau_1}/\overline{\beta}]\tau_{n_1} & = & \mathsf{S}''\tau_f \equiv \mathsf{S}'(\mathsf{S}\tau_f) \\
e_p^I & = & \mathit{Weave}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(body^I(f)), \mathsf{S}''\tau_f, \{\mathcal{A}(n_2), \cdots, \mathcal{A}(n_m)\}) \\
e_a^I & = & Body^I(n_1)\{\overline{\tau_1}\} \\
e_{n_1}^I & = & [e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]body^I(n_1) \\
\text{Hence, RHS} & = & \mathit{Trigger}(\lambda y : \tau_y.e_{n_1}^I, n_1 : [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}) = \mathit{Weave}(\lambda y : \tau_y.e_{n_1}^I, \tau_{n_1}, \{\}) \\
& = & \lambda y : \tau_y.e_{n_1}^I\ =\ \lambda y : \tau_y.[e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]body^I(n_1)
\end{array}$$

Thus, we need to show that

$$\begin{array}{rl} & [\overline{e_g}/\overline{dg}][[\![\langle Body(f)\ \overline{e_p}\ , \{e_2, \cdots, e_m\}\rangle]\!]/\texttt{proceed}][[\![(\lambda y.[\overline{e_1}'/\overline{dq}]body(n_1))]\!] \\ \simeq & [e_p^I/\texttt{proceed}](\lambda y : \tau_y.[\overline{\tau_1}/\overline{\beta}]body^I(n_1)) \end{array}$$

By the induction hypothesis for the length ($|\overline{e}| = m - 1$):

$$[\overline{e_g}/\overline{dg}][[\![\langle Body(f)\ \overline{ep}\ , \{e_2, \cdots, e_m\}\rangle]\!] \simeq e_p^I$$

Hence, by Proposition 4, we in turns need to show that

$$[\overline{e_g}/\overline{dg}]([[\![(\lambda y.[\overline{e_1}'/\overline{dq}]body(n_1))]\!] \simeq \lambda y : \tau_y.[\overline{\tau_1}/\overline{\beta}]body^I(n_1) \quad (*)$$

Finally, by the assumption of the Lemma,

$$(t, \mathsf{S}\tau_f) \vdash (n_1\ \overline{e_1}') \propto n_1\ \{AdviceUnifiers(e_1, \mathsf{S}\tau_f)\}$$

Hence, given $St \overset{\text{type}}{\rightarrowtail} \mathsf{S}'\mathsf{S}\tau_f$, by Proposition 3 and Definition 7, we have

$$[\overline{e_g}/\overline{dg}]([[\![Body'(n_1)\ \overline{e_1}']\!]) \simeq Body^I(n_1)\ \{\overline{\tau_1}\}$$

Note that, since the above respect assumption holds with no advice predicates, if there are any advice parameters in $\overline{e_1}'$, they must belong to $\overline{dg}$. Then, by the enhanced definition of $Body'(x)$ and $rank(e_1) = 0$, we have

$$[\overline{e_g}/\overline{dg}][[\![(\lambda y.[\overline{e_1}'/\overline{dq}]body(n_1))]\!] \simeq (\lambda y : \tau_y.[\overline{\tau_1}/\overline{\beta}]body^I(n_1))$$

Thus the open equivalence to show (*) holds. This concludes the case for $rank(e_c) = 1$.

Induction step for the rank ($rank(e_c)$):

Suppose that the equivalence (3) holds for all e_c with $rank(e_c) < r$. When $rank(e_c) = r$, we do induction on the length of $|\overline{e}|$.

If $|\overline{e}| = 0$, then e_c actually has rank one, and the equivalence holds by the induction base. Suppose it holds for $|\overline{e}| = m - 1$.

Now consider the case $|\overline{e}| = m$. Let $\overline{e} = e_1, e_2, \cdots, e_m$. If $rank(e_1) = 0$, then the proof is the same as the one for the case of $rank(e_c) = 1$. Otherwise, assume that $0 < rank(e_1) < r$, and let $e_1 = (\lambda\overline{dr}.\langle n_1\ \overline{dr}\ , \{\overline{e_n}\}\rangle)\ \overline{e'}$. Consider the open equation 3 that we need to prove:

$$\begin{array}{rcl} \text{LHS} & = & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f][[\![(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{e_1, e_2, \cdots, e_m\}\rangle)\ \overline{e_p}]\!] \\ & = & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f](\lambda\overline{dp}.proceedApply(e_1, \langle f\ \overline{dp}\ , \{e_2, \cdots, e_m\}\rangle))\ \overline{e_p} \\ & = & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f](\lambda\overline{dp}.[[\![\langle n_1\ \overline{e'}\ \langle f\ \overline{dp}\ , \{e_2, \cdots, e_m\}\rangle\ , \{\overline{e_n}\}\rangle]\!])\ \overline{e_p} \\ & = & [\overline{e_g}/\overline{dg}][[\![Body(f)]\!]/f] \\ & & (\lambda\overline{dp}.[[\![\langle(\lambda\overline{dq}.\lambda\texttt{proceed}.\lambda y.body(n_1))\ \overline{e'}\ \langle f\ \overline{dp}\ , \{e_2, \cdots, e_m\}\rangle\ , \{\overline{e_n}\}\rangle]\!])\ \overline{e_p} \\ & = & [\overline{e_g}/\overline{dg}][[\![\langle(\lambda\overline{dq}.\lambda\texttt{proceed}.\lambda y.body(n_1))\ \overline{e'}\ \langle Body(f)\ \overline{ep}\ , \{e_2, \cdots, e_m\}\rangle\ , \{\overline{e_n}\}\rangle]\!] \\ & \rightarrow_\beta^* & [\overline{e_g}/\overline{dg}][[\![\langle[[\![\langle Body(f)\ \overline{e_p}\ , \{e_2, \cdots, e_m\}\rangle]\!]/\texttt{proceed}](\lambda y.[\overline{e'}/\overline{dq}]body(n_1))\ , \{\overline{e_n}\}\rangle]\!] \end{array}$$

As to the RHS, applying the definition of *Weave*, we have:

$$
\begin{array}{rcl}
[\overline{\tau_1}/\overline{\beta}]\tau_{n_1} & = & \mathsf{S}''\tau_f \\
e_p^I & = & \mathit{Weave}(\lambda x : \mathsf{S}''\tau_x.\mathsf{S}''(\mathit{body}^I(f)), \mathsf{S}''\tau_f, \{n_2, \cdots, n_m\}) \\
e_a^I & = & \mathit{Body}^I(n_1)\{\overline{\tau_1}\} \\
e_{n_1}^I & = & [e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]\mathit{body}^I(n_1) \\
\text{So, RHS} & \equiv & \mathit{Trigger}(\lambda y : \tau_y.[e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]\mathit{body}^I(n_1), n_1 : [\overline{\tau_1}/\overline{\beta}]\tau_n) \\
& = & \mathit{Weave}(\lambda y : \tau_y.[e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]\mathit{body}^I(n_1), [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}, \mathit{Choose}(n_1, [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}))
\end{array}
$$

By Lemma 1, $\mathit{AdviceSet}(\lambda\overline{dr}.\langle n_1\ \overline{dr}\ , \{\overline{e_n}\}\rangle) = \mathit{Names}(\mathit{Choose}(n_1, [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}))$. Thus, we need to show that

$$
\begin{array}{rl}
& [\overline{e_g}/\overline{dg}][\![\langle[\![\langle \mathit{Body}(f)\ \overline{e_p}\ , \{e_2, \cdots, e_m\}\rangle]\!]/\texttt{proceed}](\lambda y.[\overline{e'}/\overline{dq}]\mathit{body}(n_1))\ , \{\overline{e_n}\}\rangle]\!] \\
\simeq & \mathit{Weave}(\lambda y : \tau_y.[e_p^I/\texttt{proceed}][\overline{\tau_1}/\overline{\beta}]\mathit{body}^I(n_1), [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}, \{\mathcal{A}(n_i) \mid n_i \leftarrow \overline{e_n}\}) \quad (*)
\end{array}
$$

The open equation above can be proved by induction and Proposition 4 as follows. By the induction hypothesis for the length of e_P^I ($\{e_2, \cdots, e_m\}$), we have

$$
[\overline{e_g}/\overline{dg}][\![(\lambda\overline{dp}.\langle \mathit{Body}(f)\ \overline{dp}\ , \{e_2, \cdots, e_m\}\rangle)\ \overline{e_p}]\!] \simeq e_p^I \tag{4}
$$

Besides, since $\mathit{rank}(e_1) < r$, by the induction hypothesis of the rank, we have

$$
\begin{array}{l}
[[\![\mathit{Body}(n_1)]\!]/n_1][\overline{e_g}/\overline{dg}][\![(\lambda\overline{dr}.\langle n_1\ \overline{dr}\ , \{\overline{e_n}\}\rangle)\ \overline{e'}]\!] \\
= [\overline{e_g}/\overline{dg}][\![\langle\lambda y.[\overline{e'}/\overline{dq}]\mathit{body}(n_1))\ , \{\overline{e_n}\}\rangle]\!] \\
\quad \simeq \mathit{Weave}(\lambda y : \tau_y.[\overline{\tau_1}/\overline{\beta}]\mathit{body}^I(n_1), [\overline{\tau_1}/\overline{\beta}]\tau_{n_1}, \{\mathcal{A}(n_i) \mid n_i \leftarrow \overline{e_n}\}))
\end{array} \tag{5}
$$

Finally, comparing the open equivalence to show (*) and (5), we see that (*) can be constructed by replacing the *proceed*s in both sides of (5) with the terms on the same sides of (4), respectively. By Proposition 4, the open equivalence of (*) holds for $|\overline{e}| = m$ and $\mathit{rank}(e_c) = r$. Hence the lemma follows by induction. □

Theorem 3 (Soundness of Expression Weaving) *If* $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $\Gamma \vdash e : \overline{p}.t \rightsquigarrow e'$, *and* $fv(\overline{p}) \subseteq fv(t)$, *then there exists an* e^I *such that* $\Delta \vdash e : \tau \rightarrowtail e^I$ *and* $(\overline{p}.t, \tau) \vdash e' \propto e^I$

Proof. By Theorem 1, $\Delta \vdash e : \tau \rightarrowtail e^I$ holds with $t \overset{\text{type}}{\rightarrowtail} \tau$. We prove the second part, $(\overline{p}.t, \tau) \vdash e' \propto e^I$, by induction on the height (h) of the derivation tree for $\Gamma \vdash e : \overline{p}.t \rightsquigarrow e'$.

Induction basis ($h = 1$):

There is only one case, namely (Var), and $e \equiv x$ for some variable x that $x : \sigma \rightsquigarrow e' \in \Gamma$. We prove the basis case by considering the kinds of bindings which introduce x into Γ:

case (Abs) :

Here $e' \equiv x$ and x is monomorphic, i.e., $\sigma \equiv t_x$ for some type t_x. Hence $\overline{p}$ is empty, $t = t_x$. Consequently, by $\Gamma \propto (\Delta, \mathcal{A})$ and (Expr:Var), $e^I \equiv x$. The result holds obviously.

case (PRED) :

In this case, x is advised and $x :_* \forall\bar{a}.\bar{p}_x.t_x \in \Gamma$, $\bar{p}$ is empty, $t = [\bar{t}/\bar{a}]t_x$, and $e' \equiv x_t$, a variable. Moreover, If $\Delta(x) = \forall\overline{\alpha}.\ \tau_x$, then, by $\Gamma \propto (\Delta, \mathcal{A})$ and Proposition 1, there are types $\overline{\tau}$ that $t \overset{\text{type}}{\rightarrowtail} [\overline{\tau}/\overline{\alpha}]\tau_x$. Hence, by (EXPR:TY-APP), $\tau = [\overline{\tau}/\overline{\alpha}]\tau_x$ and $e^I \equiv x\{\overline{\tau}\}$. We need to prove that $(t, \tau) \vdash x_t \propto x\{\overline{\tau}\}$.

According to Definition 7, given S, S', with $St \overset{\text{type}}{\rightarrowtail} \mathsf{S}'\tau$ and $\overline{e_g}$ feasible to $S\overline{(g : t_g)}$, we must show that $[\overline{e_g}/\overline{dg}]e' \simeq \mathsf{S}'x\{\overline{\tau}\}$. Obviously, x belongs to $\overline{g}$ and thus x_t belongs to $\overline{dg}$. Hence there is an $e_x \in \overline{e_g}$ such that $e_x \Cup S(x : t)$. Therefore, $[\overline{e_g}/\overline{dg}]e' = e_x \simeq \mathsf{S}'(x\{\overline{\tau}\})$ by the definition of feasibility. Consequently, $(t, \tau) \vdash e' \propto e^I$.

case (GLOBAL), (LET), (ADV), (ADV-AN) :

In all these cases, $e' \equiv x$ and $\overline{p}.t$ is an generic instance of $\sigma(\equiv \forall\bar{a}.\bar{p}_x.t_x)$, $\overline{p}.t = [\bar{t}/\bar{a}]\bar{p}_x.t_x$. If $\Delta(x) = \forall\overline{\alpha}.\ \tau_x$, then, by $\Gamma \propto (\Delta, \mathcal{A})$, Proposition 1 and (EXPR:TY-APP), $\tau = [\overline{\tau}/\overline{\alpha}]\tau_x$, $e^I \equiv x\{\overline{\tau}\}$, and $t \overset{\text{type}}{\rightarrowtail} \tau$. We need to prove that $(\overline{p}.t, \tau) \vdash x \propto x\{\overline{\tau}\}$.

According to Definition 7, given S, S', $\overline{e_p}$, and $\overline{e_g}$ with $(S \circ [\bar{t}/\bar{a}])t_x \overset{\text{type}}{\rightarrowtail} (\mathsf{S}' \circ [\overline{\tau}/\overline{\alpha}])\tau_x$, we need to show that

$$[\overline{e_g}/\overline{dg}]x\ \overline{e_p} \simeq (\mathsf{S}'x\{\overline{\tau}\})$$

By proposition 3, it suffices to show that

$$[\overline{e_g}/\overline{dg}][\![Body'(x)]\!]\ \overline{e_p} \ \simeq\ \mathsf{S}'(Body^I(x)\{\overline{\tau}\}) \tag{6}$$

We show it on a case by case basis.

subcase Global variable, (LET) :

Here $Body'(x) = Body(x)$ and $Body^I(x)\{\overline{\tau}\} = [\overline{\tau}/\overline{\alpha}]body^I(x)$. Besides, $(S \circ [\bar{t}/\bar{a}])t_x \overset{\text{type}}{\rightarrowtail} (\mathsf{S}' \circ [\overline{\tau}/\overline{\alpha}])\tau_x$. By $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $(\bar{p}_x.t_x, \tau_x) \vdash Body(x) \propto body^I(x)$. Hence the open equation (6) follows by applying to this respect condition with $S \circ [\bar{t}/\bar{a}]$, $\mathsf{S}' \circ [\overline{\tau}/\overline{\alpha}]$, $\overline{e_p}$, and $\overline{e_g}$, according to Definition 7.

subcase Global function :

Let $\mathsf{S}'' = \mathsf{S}' \circ [\overline{\tau}/\overline{\alpha}]$ and we can rewrite the RHS of (6) as $\mathsf{S}''(\lambda^{x:\tau_x} y : \tau_y.body^I(x))$. By proposition 2a, the open equation (6) can be verified by proving:

$$\begin{aligned}
[\overline{e_g}/\overline{dg}][\![Body(x)]\!]\ \overline{e_p} \ &\simeq\ \mathit{Trigger}(\mathsf{S}''(\lambda y : \tau_y.body^I(x)), x : \mathsf{S}''\tau_x) \\
&=\ \mathit{Weave}(\mathsf{S}''(\lambda y : \tau_y.body^I(x)), \mathsf{S}''\tau_x, \mathit{Choose}(x, \mathsf{S}''\tau_x)) \\
&=\ \mathit{Weave}(\mathsf{S}''(\lambda y : \tau_y.body^I(x)), \mathsf{S}''\tau_x, \{\}) \qquad (7) \\
&=\ \mathsf{S}''(\lambda y : \tau_y.body^I(x)) \qquad (8)
\end{aligned}$$

Equality (7) holds because any x's typed by the (VAR) rule is not advised (i.e., $\mathit{Choose}(x, \mathsf{S}''\tau_x)) = \{\}$). Moreover, by $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $(\bar{p}_x.t_x, \tau_x) \vdash Body(x) \propto \lambda y : \tau_y.body(x)$. Similar to the previous case, equality (8) follows by applying to this respect condition with $S \circ [\bar{t}/\bar{a}]$, S'', $\overline{e_p}$, and $\overline{e_g}$, according to definition 7.

subcase (ADV), (ADV-AN) : Similar to the subcase of global function except that we use the enhanced version of $Body'(x)$.

Induction step:

Suppose that the respect condition hold for all derivation trees of height less then h. We prove that, for an expression e with a derivation tree, $(\overline{p}.t, \tau) \vdash e' \propto e^I$, of height h. Consider the last step of the derivation:

case (PRED) :

We have a derivation of the form:

$$\frac{x :_* \forall \bar{a}.\overline{p}_x.t_x \in \Gamma \quad [\bar{t}/\bar{a}]t_x \trianglerighteq t_1 \qquad \Gamma.x : t_1 \rightsquigarrow x_t \vdash e : \overline{q}.t \rightsquigarrow e'_t}{\Gamma \vdash e : (x : t_1).\overline{q}.t \rightsquigarrow \lambda x_t.e'_t}$$

Hence, $\overline{p} \equiv (x : t_1).\overline{q}$ and $e' \equiv \lambda x_t.e'_t$.

Since $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$ and t_1 is an instance of t_x, $\Gamma.x : t_1 \rightsquigarrow x_t \overset{\pi}{\propto} (\Delta, \mathcal{A})$. By the induction hypothesis, there exist τ_1 and e_1^I such that $t \overset{\text{type}}{\rightarrowtail} \tau_1$ and $(\overline{q}.t, \tau_1) \vdash e'_t \propto e_1^I$. Moreover, since (PRED) has no corresponding rule in FIL, $e^I \equiv e_1^I$ and $\tau \equiv \tau_1$. Now to finish the proof of this case, we need to show that

$$(\overline{p}, \tau) \vdash \lambda x_t.e'_t \propto e^I$$

In other words, given S, S', $\overline{e_p}$, and $\overline{e_g}$ with $St = \mathsf{S}'\tau$ as in Definition 7,

$$[\![[\overline{e_g}/\overline{dg}](\lambda x_t.e'_t)\ \overline{e_p}]\!] \simeq \mathsf{S}'e^I$$

Since $p \equiv (x : t_1).\overline{q}$, we can write e_p as $e_x.\overline{e_q}$ such that $e_x \in \overline{e_p}$ and $e_x \Cup (x : t_1)$. Then, by the induction hypothesis and Definition 7, we have

$$[\![[\overline{e_g}.e_x/\overline{dg}.x_t]e'_t\ \overline{e_q}]\!] \simeq \mathsf{S}'e^I$$

Since

$$\begin{aligned} \text{LHS} &= [\![[\overline{e_g}.e_x/\overline{dg}.x_t]e'_t\ \overline{e_q}]\!] \\ &= [\![[\overline{e_g}/\overline{dg}][e_x/x_t]e'_t\ \overline{e_q}]\!] \\ &= [\![[\overline{e_g}/\overline{dg}](\lambda x_t.e'_t)\ e_x\ \overline{e_q}]\!] \\ &= [\![[\overline{e_g}/\overline{dg}](\lambda x_t.e'_t)\ e_x.\overline{e_q}]\!], \end{aligned}$$

we get $[\![[\overline{e_g}/\overline{dg}](\lambda x_t.e'_t)\ \overline{e_p}]\!] \simeq \mathsf{S}'e^I$.

case (REL) :

We have a derivation of the form:

$$\frac{\Gamma \vdash e : (x : t_1).\rho \rightsquigarrow e'_1 \qquad \Gamma \vdash x : t_1 \rightsquigarrow e'_2 \quad x \neq e}{\Gamma \vdash e : \rho \rightsquigarrow e'_1\ e'_2}$$

Hence $\overline{p}.t \equiv \rho$ and $e' \equiv e'_1\ e'_2$. By the induction hypothesis on the first derivation, there exists an τ_1 and e_1^I such that $(x : t_1.\rho, \tau_1) \vdash e'_1 \propto e_1^I$ where $\rho \equiv \overline{p}.t$.

Moreover, let τ be τ_1 and e^I be e_1^I, then we need to show that $(\rho, \tau) \vdash e_1' e_2' \propto e^I$. In other words, given S, S', $\overline{e_p}$, and $\overline{e_g}$ with $St = \mathsf{S}'\tau$ as in Definition 7,

$$[\![[\overline{e_g}/\overline{dg}](e_1' e_2')\ \overline{e_p}]\!] \simeq \mathsf{S}' e^I$$

Let $\forall\overline{\alpha}.\tau_x = \Delta(x)$, then, by the induction hypothesis on the second derivation, we get $(t_1, \tau_x') \vdash e_2' \propto x\{\overline{\tau}\}$ where $\tau_x' = [\overline{\tau}/\overline{\alpha}]\tau_x$ and $t_1 \overset{\text{type}}{\rightarrowtail} \tau_x'$. By Definition 7 and a reasoning similar to the previous case, it suffices to show that $e_2' \precsim S(x : t_1)$. Now, since $(t_1, \tau_x') \vdash e_2' \propto x\{\overline{\tau}\}$, we have $[\![[\overline{e_g}/\overline{dg}]e_2']\!] \simeq \mathsf{S}' x\{\overline{\tau}\}$, which proves that $e_2' \precsim S(x : t_1)$.

case (VAR-A) :

In this case, x must be an advised function or advice. We can prove it by applying Lemma 2 as follows.

Let $\forall\bar{a}.\bar{p}_x.t_x = \Gamma(x)$ and $\forall\overline{\alpha}.\tau_x = \Delta(x)$. Then, $\overline{p} = [\bar{t}/\bar{a}]p_x$ and $t = [\bar{t}/\bar{a}]t_x$. $e^I \equiv x\{\overline{\tau}\}$. By $\Gamma \propto (\Delta, \mathcal{A})$ and Proposition 1, there exists an $\mathsf{S} = [\overline{\tau}/\overline{\alpha}]$ such that $t \overset{\text{type}}{\rightarrowtail} \mathsf{S}\tau_x$, as required by Lemma 2.

Finally, from the premise of the rule, we have $\Gamma \vdash n_i : t \rightsquigarrow e_i$. By the induction hypotheses of these sub-derivations, there exist e_i^I such that $\Delta \vdash n_i : \tau \rightarrowtail e_i^I$ and $(t, \mathsf{S}\tau_x) \vdash e_i \propto e_i^I$. Moreover, let $\forall\overline{\beta_i}.\tau_{n_i} = \Delta(n_i)$. Then, by Theorem 1, $t \overset{\text{type}}{\rightarrowtail} [\mathit{AdviceUnifiers}(e_i, \mathsf{S}\tau_x)/\overline{\beta_i}]\tau_{n_i}$. Hence, by (EXPR:TY-APP) it is obvious that $(n_i\ \{\mathit{AdviceUnifiers}(e_i, \mathsf{S}\tau_x)\})$ are such e_i^I. This case then follows directly from Lemma 2.

case (LET) We have a derivation of the form:

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

Here $e' \equiv \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'$ and f can be either a variable or a function. We present only the proof for the case of variable since the case of function is similar.

Suppose $\rho \equiv \overline{q}.t'$. By the induction hypothesis of the first sub-derivation, there is an τ_f and e_f^I such that $\Delta \vdash e : \tau \rightarrowtail e^I$, $t' \overset{\text{type}}{\rightarrowtail} \tau_f$" and $(\rho, \tau_f) \vdash e_1' \propto e_f^I$.

Since $e_1' = \mathit{Body}(f)$ and $e_f^I = \mathit{body}^I(f)$, by Definition 8 and $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$, $\Gamma.f : \forall\overline{a}.\rho \rightsquigarrow f \overset{\pi}{\propto} (\Delta.f : \forall\overline{\alpha}.\tau_f, \mathcal{A})$. Hence, by the induction hypothesis of the second sub-derivation, there exist an type τ and e_2^I such that $t \overset{\text{type}}{\rightarrowtail} \tau$ and

$$(t, \tau) \vdash e_2' \propto e_2^I \quad \text{Under } \Gamma.f : \sigma \rightsquigarrow f$$

Now, applying the substitution S' and $\overline{e_g}$ derived from Definition 7 to the above respect condition, we get

$$[\![[\overline{e_g}/\overline{dg}]e_2']\!] \simeq \mathsf{S}' e_2^I \quad \text{Under } \Gamma.f : \sigma \rightsquigarrow f$$

Given $\mathit{Body}(f) = e_1'$, and $\mathit{Body}^I(f) = \Lambda\overline{\alpha}.e_f^I$, by Proposition 3 we have

$$[\![[\overline{e_g}/\overline{dg}][e_1'/f]e_2']\!] \simeq \mathsf{S}'[\Lambda\overline{\alpha}.e_f^I/f]e_2^I \quad \text{Under } \Gamma$$

By β-reduction, the open equation above is equivalent to

$$[\![[\overline{e_g}/\overline{dg}](\mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2')]\!] \simeq \mathsf{S}'(\mathcal{LET}\ f = \ \Lambda\overline{\alpha}.\ e_f^I\ \mathcal{IN}\ e_2^I)$$

Therefore, the FIL expression $\mathcal{LET}\ f = \ \Lambda\overline{\alpha}.\ e_f^I\ \mathcal{IN}\ e_2^I$ is the e^I for this case.

case (ABS) :

We have a derivation of the form:

$$\frac{\Gamma.x : t_1 \rightsquigarrow x \vdash\ e_b : t_2 \rightsquigarrow e_b'}{\Gamma \vdash\ \lambda x.e_b : t_1 \to t_2 \rightsquigarrow \lambda x.e_b'}$$

Pick a τ_x such that $t_1 \overset{\text{type}}{\rightarrowtail} \tau_x$. Then, by $\Gamma \overset{\pi}{\propto} (\Delta, \mathcal{A})$ and Definition 8, we have $\Gamma.x : t_1 \rightsquigarrow x \overset{\pi}{\propto} (\Delta.x : \tau_x, \mathcal{A})$. Then, by the induction hypothesis on the first sub-derivation we have τ_b and e_b^I with which we can apply (EXPR:ABS) and get

$$\frac{\Delta.x : \tau_x \vdash e_b : \tau_b \rightarrowtail e_b^I}{\Delta \vdash \lambda x.\ e_b : \tau_x \to \tau_b \rightarrowtail \lambda x : \tau_x.\ e_b^I}$$

where $t_2 \overset{\text{type}}{\rightarrowtail} \tau_b$, and $(t_2, \tau_b) \vdash e_b' \propto e_b^I$. Thus, $\bar{p}$ is empty, $t \equiv t1 \to t_2$, $\tau \equiv \tau_x \to \tau_b$, $e \equiv \lambda x.e_b$ and $e' \equiv \lambda x.e_b'$ and $e^I \equiv \lambda^\epsilon x : \tau_x.e_b^I$.

We need to prove that $(t, \tau) \vdash e' \propto e^I$. By Definition 7, given S, S', and $\overline{e_g}$, we must show

$$[\![[\overline{e_g}/\overline{dg}](\lambda x.e_b')]\!] \simeq \mathsf{S}'(\lambda^\epsilon x : \tau_x.e_b^I) \tag{9}$$

By the induction hypothesis, we have

$$[\![[\overline{e_g}/\overline{dg}]e_b']\!] \simeq \mathsf{S}'e_b^I \quad \text{under } \Gamma.x : t_1$$

According to Definition 4, given $e_v \sim\!\!\sim e_v^I$, we have

$$[Body'(\overline{f})/\overline{f}][e_v/x][\![[\overline{e_g}/\overline{dg}]e_b']\!] \sim\!\!\sim [Body^I(\overline{f})/\overline{f}][e_v^I/x]\mathsf{S}'e_b^I \quad \text{under} \quad \mathcal{A}$$

By Definition 3,

$$[Body'(\overline{f})/\overline{f}][\![[\overline{e_g}/\overline{dg}]\lambda x.e_b']\!] \sim\!\!\sim [Body^I(\overline{f})/\overline{f}]\mathsf{S}'(\lambda x : \tau_x.e_b^I) \quad \text{under} \quad \mathcal{A}$$

By Definition 4,

$$[\![[\overline{e_g}/\overline{dg}]\lambda x.e_b']\!] \simeq \mathsf{S}'(\lambda x : \tau_x.e_b^I) \quad \text{under} \quad \Gamma \text{ and } \mathcal{A}$$

which proves the case.

case (APP) :

By straightforward induction on e_1 and e_2 of $(e_1\ e_2)$.

□

A special case of Definition 7 occurs when there is no binding in Γ which is introduced via (PRED). For example, after translating the body of a top level definition, such bindings do not exist. In such a case, we do not have to consider $\overline{e_g}$ in Definition 7 and we add a superscript T on the judgement to distinguish it from the general respect of expressions, written as

$$(\overline{p}.t, \tau) \vdash^T e' \propto e^I$$

Proposition 5 *If $(\overline{p}.t, \tau) \vdash^T e \propto e^I$ and $\overline{p}$ is empty, then $[\![e]\!] \simeq \mathsf{S}e^I$ for any type substitution S.*

Theorem 4 (Soundness of Static Weaving) *Let π_0 be an AspectFun program. If $\emptyset \vdash \pi_0 : t \rightsquigarrow e'$, then there exists an FIL-converted program, $\pi^I = (\mathcal{A}, e^I)$, such that $\pi_0 \overset{prog}{\rightarrowtail} \pi^I$ and $[\![e']\!] \sim\sim e^I$ under $\mathcal{A}$.*

Proof. It is equivalent to show that if $\vdash \pi_0 : t \rightsquigarrow e'$ then $\vdash_D \pi_0 : \tau \rightarrowtail e^I; \mathcal{A}$ and $[\![e']\!] \simeq e^I$ under $\emptyset$ and $\mathcal{A}$. We use a stronger proposition to prove it. Suppose $\Gamma \overset{\pi_0}{\propto} (\Delta, \mathcal{A})$ and all bindings in Γ are introduced only via (Global), (Adv), or (Adv-An). If $\Gamma \vdash \pi : t \rightsquigarrow e'$ for a sub-program π of π_0, i.e. $\pi_0 \equiv \overline{d_0}.\pi$, then there exists an FIL-conversion $\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A}_1$ such that $[\![e']\!] \simeq e^I$ under Γ and $\mathcal{A} \cup \mathcal{A}_1$. Afterwards, the original result of the theorem can be obtained by assigning $\emptyset$ to Γ, Δ, $\mathcal{A}$ and $\overline{d_0}$.

Let $\pi \equiv \overline{d}.e$. We prove the above proposition by induction on the length of declarations of π, $|\overline{d}|$.

Induction basis:
$\overline{d} = 0$: we have $\pi \equiv e_0$. Since no binding in Γ is introduced via (Pred), this case is a direct consequence of Theorem 3 and Proposition 5.

Induction step:
When the proposition holds for π with length$(\overline{d}) = k$, we shall prove it for π_1 with length$(\overline{d}) = k + 1$. Let $\pi_1 \equiv d.\pi$. The Induction step to prove is that if $\Gamma_1 \overset{\pi_0}{\propto} (\Delta_1, \mathcal{A}_1)$ and $\Gamma_1 \vdash \pi_1 : t_1 \rightsquigarrow e_1'$ then $\Delta_1 \vdash_D \pi_1 : \tau_1 \rightarrowtail e_1^I; \mathcal{A}_2$ and $[\![e_1']\!] \simeq e_1^I$ under Γ_1 and $\mathcal{A}_2 \cup \mathcal{A}_1$. We prove it by a case analysis on d and induction on the derivation for $\Gamma_1 \vdash d.\pi : t_1 \rightsquigarrow e_1'$:

case (GLOBAL variable) :
We have a derivation of the form:

$$\frac{\Gamma_1 \vdash e_x : \rho_x \rightsquigarrow e_x' \quad \sigma = gen(\Gamma_1, \rho_x) \quad \Gamma_1.x : \sigma \rightsquigarrow x \vdash \pi : t_1 \rightsquigarrow e'}{\Gamma_1 \vdash \ x = e_x \text{ in } \pi : t_1 \rightsquigarrow \ x = e_x' \text{ in } e'}$$

Given $\Gamma_1 \vdash e_x : \rho_x \rightsquigarrow e_x'$, by Theorem 3, $\Delta_1 \vdash e_x : \tau_x \rightarrowtail e_x^I$, $\rho_x \overset{\text{type}}{\rightarrowtail} \tau_x$ (1) and $(\rho_x, \tau_x) \vdash e_x' \propto e_x^I$. Besides, since no binding in Γ is introduced via (Pred), we have $(\rho_x, \tau_x) \vdash^T Body(x) \equiv e_x' \propto e_x^I \equiv body^I(x)$. So, by $\Gamma_1 \overset{\pi_0}{\propto} (\Delta_1, \mathcal{A}_1)$ and Definition 8,

$$\Gamma \equiv \Gamma_1.x : gen(\Gamma_1, \rho_x) \rightsquigarrow x \quad \overset{\pi_0}{\propto} \quad (\Delta_1.x : gen(\Delta, \tau_x), \mathcal{A}_1) \equiv (\Delta, \mathcal{A}_1)$$

Thus, by the induction hypothesis of the second derivation, $\Gamma \vdash \pi : t_1 \rightsquigarrow e'$, we have

$$\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A} \ \ (2) \ \text{ and } [\![e']\!] \simeq e^I \ \text{ under } \Gamma \text{ and } \mathcal{A} \cup \mathcal{A}_1$$

Then, by (1) and (2), we can apply the (Decl:Var) rule and get

$$\begin{array}{ll} & \Delta_1 \vdash_D \pi_1 : \tau_1 \rightarrowtail e_1^I; \mathcal{A}_1 \\ \text{where} & \tau_1 \equiv \tau \\ & e_1^I \equiv \mathcal{LET}\ x = Body^I(x)\ \mathcal{IN}\ e^I \\ & \mathcal{A}_2 \equiv \mathcal{A} \\ \text{and} & [\![e']\!] \simeq e^I \ \text{ under } \Gamma \text{ and } \mathcal{A}_2 \cup \mathcal{A}_1 \end{array}$$

Applying Proposition 3, we get

$$[\![[Body(x)/x]e']\!] \simeq [Body^I(x)/x]e^I \text{ under } \Gamma_1 \text{ and } \mathcal{A}_2 \cup \mathcal{A}_1$$

and finally

$$[\![e'_1]\!] \equiv [\![x = e'_x \text{ in } e']\!] \simeq e^I_1 \text{ under } \Gamma_1 \text{ and } \mathcal{A}_2 \cup \mathcal{A}_1$$

by β-reduction.

case (GLOBAL function) :
We have a derivation of the form:

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

Similar to the case of global variables, except that the respect of expression is $(\rho_f, \tau_f) \vdash^T (Body(f) \equiv e'_f) \propto (\lambda x : \tau_x.body^I(f) \equiv e^I_f)$, and we should apply the (DECL:FUNC) rule to construct e^I_1.

case (ADV) :
We have a derivation of the form:

$$\frac{\begin{array}{l}\Gamma_1.proceed : t_p \vdash \lambda x.e_a : \rho_n(\equiv \bar{p}.t_p) \rightsquigarrow e'_a \quad f_i : \forall \bar{a}.t_i \in \Gamma_{base} \\ t_p \trianglerighteq [\bar{t}/\bar{a}]t_i \quad \Gamma.n : \sigma \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t_1 \rightsquigarrow e' \quad \sigma = gen(\Gamma_1, \rho_n)\end{array}}{\Gamma_1 \vdash n@\text{advice around } \{\bar{f}\}\ (x) = e_a \text{in } \pi : t_1 \rightsquigarrow n = e'_a \text{ in } e'}$$

First, we show that we can apply the FIL conversion rule, (DECL:ADV-AN), via the (DECL:ADV) rule, to this advice and obtain the required FIL derivation.

Pick a τ_p such that $t_p \overset{\text{type}}{\rightarrowtail} \tau_p$. Then, by Theorem 3 there exist $\tau_n \equiv \tau_1 \to \tau_2$ and e^I_n that

$$\Delta_1.\texttt{proceed} : \tau_p \vdash \lambda x.e_a : \tau_n \rightarrowtail \lambda x : \tau_1.e^I_n,\ \rho_n \overset{\text{type}}{\rightarrowtail} \tau_n, \text{ and } (\rho_n, \tau_n) \vdash e'_a \propto \lambda x : \tau_1.e^I_n$$

Moreover, by $t_p \overset{\text{type}}{\rightarrowtail} \tau_n$, we can choose τ_p so that $\tau_p \equiv \tau_n$. Thus,

$$\Delta_1.\texttt{proceed} : \tau_p \vdash \lambda x.e_a : \tau_p \rightarrowtail \lambda x : \tau_1.e^I_n$$

Since no binding is Γ is introduced via (PRED), we have

$$(\rho_n, \tau_n) \vdash^T Body(n) \equiv e'_a \propto \lambda x : \tau_1.body^I(n) \equiv \lambda x : \tau_1.e^I_n$$

To apply the (DECL:ADV-AN) rule, the (DECL:ADV) rule adds the type scope, a, to the advice. Obviously, $a \overset{\text{type}}{\rightarrowtail} \alpha$ and hence $\alpha \sqcap \tau_1 = \tau_1$. Thus the third element in the Adv tuple is τ_1 and we have $t_1 \overset{\text{type}}{\rightarrowtail} \tau_1$ where t_1 is the parameter part of ρ_n. Combining the above results, we have

$$\Gamma \equiv \Gamma_1.n : gen(\Gamma, \rho_n) \bowtie f \rightsquigarrow n \overset{\pi_0}{\propto} (\Delta_1, \mathcal{A}_1.(n : gen(\Delta, \tau_n), \overline{f}, \tau_1, Body^I(n))) \equiv (\Delta, \mathcal{A}_0)$$

Second, by the induction hypothesis of $\Gamma \vdash \pi : t \rightsquigarrow e'$, we have

$$\Delta \vdash_D \pi : \tau \rightarrowtail e^I; \mathcal{A} \text{ and } [\![e']\!] \simeq e^I \text{ under } \Gamma \text{ and } \mathcal{A}_0 \cup \mathcal{A}$$

Thus we get all the premises of (DECL:ADV-AN) fulfilled. Hence

$$\begin{array}{ll} & \Delta_1 \vdash_D \pi_1 : \tau_1 \rightarrowtail e_1^I; \mathcal{A}_2 \\ \text{where} & \tau_1 \equiv \tau \\ & e_1^I \equiv e^I \\ & \mathcal{A}_2 = \mathcal{A}.(n : gen(\Delta, \tau_n), \overline{f}, \tau_1, Body^I(n)) \\ \text{and} & [\![e']\!] \simeq e^I \text{ under } \Gamma \text{ and } \mathcal{A}_0 \cup \mathcal{A} \end{array}$$

As the last step, we need to show that $[\![e'_1]\!] \simeq e_1^I$ under Γ_1 and $\mathcal{A}_2 \cup \mathcal{A}_1$ where $e'_1 \equiv n = e'_a \text{ in } e'$. Clearly $\mathcal{A}_2 \cup \mathcal{A}_1 = \mathcal{A}_0 \cup \mathcal{A}$. Now applying Proposition 3 to $[\![e']\!] \simeq e^I$ under Γ and $\mathcal{A}_0 \cup \mathcal{A}$, we get

$$[\![[e'_a/n]e']\!] \simeq [Body^I(n)/n]e_1^I \text{ under } \Gamma_1 \text{ and } \mathcal{A}_2 \cup \mathcal{A}_1$$

Thus

$$[\![e'_1]\!] \equiv [\![n = e'_a \text{ in } e']\!] \simeq e_1^I \text{ under } \Gamma_1 \text{ and } \mathcal{A}_2 \cup \mathcal{A}_1$$

by β-reduction and that advice names never appear in e_1^I.

case (ADV-AN) : We have a derivation of the form:

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

Pick τ_1 and τ_2 such that $t_x \rightarrow t \overset{\text{type}}{\rightarrowtail} \tau_1 \rightarrow \tau_2$. The rest of the proof is very similar to the previous case, except the part for the third element (i.e., $\tau_x \sqcap \tau_1$) in the Adv tuple, to be included in the advice store by (DECL:ADV-AN). As required by (DECL:ADV-AN), the τ_x must satisfy $t_x \overset{\text{type}}{\rightarrowtail} \tau_x$, hence we can choose $\tau_x \equiv \tau_1$. This gives us $\tau_x \sqcap \tau_1 = \tau_x$, which is still the parameter part of the FIL converted type of n, $\tau_1 \rightarrow \tau_2$.

□

計畫成果自評

本計畫成果摘要如下:

1. 提出了型態導向的多型剖面靜態織入(static weaving)方法，不僅可以處理 type-coped advice, indirect function calls, higher-order function, overlapping advices，而且完全是以 static typing 獲致一致(coherent)的織入結果。
2. 設計了一實驗性的剖面導向函數語言 AspectFun，並實作其編譯器，將其程式轉換為可執行的 Haskell 語言程式碼 (可於 Open Foundry 下載)。
3. 定義了 AspectFun 語言的操作型語意，並藉以證明我們的靜態織入方法的正確性。
4. 發展了可以支援 cflow pointcut 的編譯技術，並實作於 AspectFun 的編譯器內。
5. 發展了優化 cflow 剖面的分析技術。

發表於下列國際學術會議:

The 14th International Static Analysis Symposium (SAS 2007)
Kongens Lyngby, Denmark
22-24 August 2007
論文名稱: A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages

並指導陳忠信以 AspectFun 語言的型態分析與織入的實作完成碩士論文。

當初預期目標絕大部分都已完成，並有一些非規劃內的成果，例如: cflow 剖面的編譯與優化分析。

出席國際會議研究心得報告及發表論文

政治大學資訊科學系

陳 恭

97/10

計畫名稱：以型態導向方法發展多型剖面的織入技術與應用

編號：NSC 95－2221－E—004－004－MY2

執行期間：95/08/01~97/07/31

本計畫執行結果已發表於下列國際學術會議:
The 14th International Static Analysis Symposium (SAS 2007)
Kongens Lyngby, Denmark
22-24 August 2007
論文名稱: A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages
收錄於論文集: Riis Nielson, Gilberto File (Eds.): Static Analysis, 14th International Symposium, SAS 2007, Lecture Notes in Computer Science 4634 Springer 2007, ISBN 978-3-540-74060-5

Static Analysis Symposium 是程式分析國際社群一年一度的學術會議，今年是第十四屆，在丹麥首都哥本哈根近郊的 Kongens Lyngby 的丹麥科技大學舉行，會議從 8/22 到 8/24 舉行，來自全球各地學術界與產業界共計有近百人左右與會，本人與參與計畫的學生翁書鈞是僅有的兩位來自台灣的與會人員。會議議程除一般研究論文外，大會也特別邀請了兩位資深學者進行專題演講，內容相當豐富。

本人的論文是發表在第一天的上午，論文題目為” A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages”。此篇論文是我們專題研究計畫的主要成果，針對 Aspect-Oriented Functional Languages，提出型態導向的多型剖面織入方法的理論與實作雛型技術。我們的方法可以處理 type-coped advice, indirect function calls, higher-order function, overlapping advices，而且完全是以 static typing 獲致一致的織入(weaving)結果。普林斯頓大學的 PolyAML 不能完全以 static typing 做到 type-scoped advice 的織入.東京大學的 Aspectual Caml 無法處理 indirect function calls, higher-order function, overlapping advices 的一致性。我們除了發展理論外，也設計並實作了 AspectFun 語言，將一些常用的剖面導向機制納入，例如 curried pointcuts 與 cflow pointcuts。並以靜態分析技術，針對使用 cflow 的剖面進行 control flow 優化分析。論文將 AspectFun 語言的編譯過程做了重點式的介紹。限於篇幅，一些細節與靜態織入方法的證明，未來將以期刊論文方式發表。

會議中，曾與其他幾位學者就我們的論文結果進行了一些較深入的討論，並交換心得。值得一提的是美國加州大學洛杉磯分校的資深教授 Jens Palsberg 也跟我們分享了他對於 control flow 分析的一些心得並對我們提出建議。依據他的經驗，在這方面，must cflow 的分析結果，要比 may cflow 的分析結果對程式效能的影響大，所以他建議我們可針對這一點再進行更廣泛的實證性探討。

此外，由於我們的編譯器中使用到 reader monad，並實作了部分的 monad 式的程式轉換，在這個基礎上，也有學者建議我們可以進一步去探討 monadification 轉換的技術，並將它應用到 AspectFun 語言的編譯器上。我們就此思考後，認為可以就 side-effect 運算以 monad 方式納入 AspectFun 作為未來研究的一個主題。一般說來，使用純粹函數式語言時，所有牽涉到共用狀態(shared state)的函數通通要改寫，將狀態當成一額外的參數，在彼此之間相互傳遞(threaded with a state parameter)。近來 Haskell 引進 monad 的機制，可以將狀態處理封裝在 monad 之內，大幅簡化狀態處理的程式撰寫，但是一旦使用 monad，程式所有相關部分都要改成 monadic 的寫法，因此也是一種橫跨式的大幅變動。偏偏 aspect 常用來實現的 profiling 與 tracing 等工作，都是需要有狀態處理功能的，所以即便我們可以將狀態處理部分限縮在 aspect 內，以 monad 實現模組化狀態處理的需求，但是原本是單純的函數構成的程式本體，卻也必須跟著改寫成 monadic 方式，如此一來，大大減損了使用 aspect 模組化的優勢。所以未來一個重點就是要探討如何在 AspectFun 直接提供具備狀態處理功能的剖面 (side-effecting aspects)，透過編譯技術將其與程式本體自動轉換成 monadic 方式，以提高純粹函數程式的處理狀態的模組化程度。

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

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 34–51, 2007.

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 determine *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.

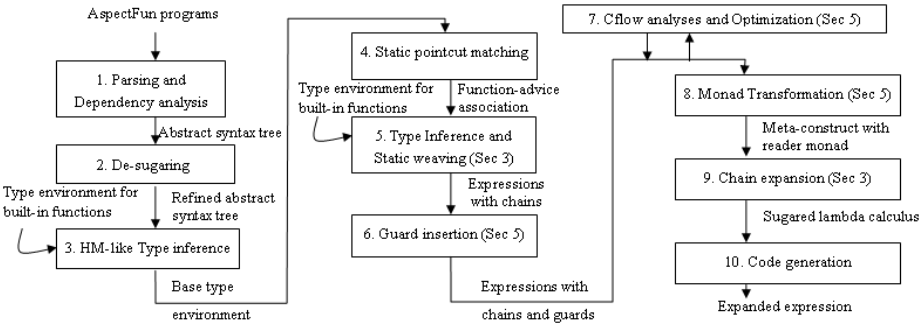

Fig. 2. Compilation Model for AspectFun

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-order*

[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.

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 abbreviation

[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.

for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and 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	$::= d$ in $\pi \mid e$
Declarations	d	$::= x = e \mid f\ \overline{x} = e \mid n$@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	$::=$ `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	$::= \forall \bar{a}.\rho$

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

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

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.

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

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

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

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

$$
(\text{EXPR:VAR})\ \frac{\tau = \Delta(x)}{\Delta \vdash x : \tau \rightarrowtail x}\ (\text{EXPR:TY-APP})\ \frac{\forall\overline{\alpha}.\ \tau = \Delta(x) \quad \tau_x = [\overline{\tau'}/\overline{\alpha}]\tau}{\Delta \vdash x : \tau_x \rightarrowtail x\{\overline{\tau'}\}}
$$

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

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

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

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:

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

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

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

$$\text{JPMatch}(f, pc) = (f \equiv pc)$$

Fig. 6. Operational Semantics for 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.

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].

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

Expressions:

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

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

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

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

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

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}.

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 $S(t_1 \to t_2)$ 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:

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

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

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

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

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

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.

$$
\begin{array}{lll}
e_M & : & \text{Expressions containing meta-constructs} \\
\text{addProceed} & : & e_M \longrightarrow e_M \\
\text{addProceed}(\mathsf{let}\ n\ df\ arg\ =\ e_1\ \mathsf{in}\ e_2) & = & \mathit{if}\ (n \text{ is an advice})\ \mathit{then} \\
 & & \quad \mathsf{let}\ n\ df\ \mathit{proceed}\ arg\ =\ e_1 \\
 & & \quad \mathsf{in}\ \text{addProceed}(e_2) \\
 & & \mathit{else}\ \mathsf{let}\ n\ df\ arg\ =\ e_1\ \mathsf{in}\ \text{addProceed}(e_2) \\
\text{addProceed}(e) & = & e
\end{array}
$$

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

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

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

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 $\Downarrow_{\mathcal{A}}^{\mathcal{S}}$, 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 $(\!|e, S|\!)$:

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

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

$(\!|e, \mathcal{S}|\!)$ 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
```

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

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 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.

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. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising Aspect J. In: PLDI '05. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, pp. 117–128. ACM Press, NewYork (2005)
2. Chen, K., Weng, S.-C., Wang, M., Khoo, S.-C., Chen, C.-H.: 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. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: PolyAML: A polymorphic aspect-oriented functional programmming language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
4. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. ACM Transactions on Programming Languages and Systems (TOPLAS) (to appear, 2006)
5. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming (to appear, 2006)
6. Jones, M.P.: Qualified Types: Theory and Practice. D.phil. thesis, Oxford University (September 1992)
7. M.P. Jones.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming, pp. 97–136 (1995)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
9. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: CC, pp. 46–60 (2003)
10. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
11. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc, Secaucus, NJ, USA (1999)
12. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 59–68. ACM Press, NewYork (2005)
13. Sereni, D., de Moor, O.: Static analysis of aspects. In: Aksit, M. (ed.) AOSD. 2nd International Conference on Aspect-Oriented Software Development, pp. 30–39. ACM Press, NewYork (2003)
14. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, pp. 60–76. ACM, NewYork (1989)
15. Wang, M., Chen, K., Khoo, S.-C.: Type-directed weaving of aspects for higher-order functional languages. In: PEPM '06. Workshop on Partial Evaluation and Program Manipulation, ACM Press, NewYork (2006)

出席國際會議研究心得報告及發表論文

政治大學資訊科學系

陳 恭

97/10

計畫名稱：以型態導向方法發展多型剖面的織入技術與應用

編號：NSC 95－2221－E—004－004－MY2

執行期間：95/08/01~97/07/31

本計畫執行結果已發表於下列國際學術會議:
The 14th International Static Analysis Symposium (SAS 2007)
Kongens Lyngby, Denmark
22-24 August 2007
論文名稱: A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages
收錄於論文集: Riis Nielson, Gilberto File (Eds.): Static Analysis, 14th International Symposium, SAS 2007, Lecture Notes in Computer Science 4634 Springer 2007, ISBN 978-3-540-74060-5

Static Analysis Symposium 是程式分析國際社群一年一度的學術會議，今年是第十四屆，在丹麥首都哥本哈根近郊的 Kongens Lyngby 的丹麥科技大學舉行，會議從 8/22 到 8/24 舉行，來自全球各地學術界與產業界共計有近百人左右與會，本人與參與計畫的學生翁書鈞是僅有的兩位來自台灣的與會人員。會議議程除一般研究論文外，大會也特別邀請了兩位資深學者進行專題演講，內容相當豐富。

本人的論文是發表在第一天的上午，論文題目為” A Compilation Model for Aspect-Oriented Polymorphically Typed Functional Languages”。此篇論文是我們專題研究計畫的主要成果，針對 Aspect-Oriented Functional Languages，提出型態導向的多型剖面織入方法的理論與實作雛型技術。我們的方法可以處理 type-coped advice, indirect function calls, higher-order function, overlapping advices，而且完全是以 static typing 獲致一致的織入(weaving)結果。普林斯頓大學的 PolyAML 不能完全以 static typing 做到 type-scoped advice 的織入.東京大學的 Aspectual Caml 無法處理 indirect function calls, higher-order function, overlapping advices 的一致性。我們除了發展理論外，也設計並實作了 AspectFun 語言，將一些常用的剖面導向機制納入，例如 curried pointcuts 與 cflow pointcuts。並以靜態分析技術，針對使用 cflow 的剖面進行 control flow 優化分析。論文將 AspectFun 語言的編譯過程做了重點式的介紹。限於篇幅，一些細節與靜態織入方法的證明，未來將以期刊論文方式發表。

會議中，曾與其他幾位學者就我們的論文結果進行了一些較深入的討論，並交換心得。值得一提的是美國加州大學洛杉磯分校的資深教授 Jens Palsberg 也跟我們分享了他對於 control flow 分析的一些心得並對我們提出建議。依據他的經驗，在這方面，must cflow 的分析結果，要比 may cflow 的分析結果對程式效能的影響大，所以他建議我們可針對這一點再進行更廣泛的實證性探討。

此外，由於我們的編譯器中使用到 reader monad，並實作了部分的 monad 式的程式轉換，在這個基礎上，也有學者建議我們可以進一步去探討 monadification 轉換的技術，並將它應用到 AspectFun 語言的編譯器上。我們就此思考後，認為可以就 side-effect 運算以 monad 方式納入 AspectFun 作為未來研究的一個主題。一般說來，使用純粹函數式語言時，所有牽涉到共用狀態(shared state)的函數通通要改寫，將狀態當成一額外的參數，在彼此之間相互傳遞(threaded with a state parameter)。近來 Haskell 引進 monad 的機制，可以將狀態處理封裝在 monad 之內，大幅簡化狀態處理的程式撰寫，但是一旦使用 monad，程式所有相關部分都要改成 monadic 的寫法，因此也是一種橫跨式的大幅變動。偏偏 aspect 常用來實現的 profiling 與 tracing 等工作，都是需要有狀態處理功能的，所以即便我們可以將狀態處理部分限縮在 aspect 內，以 monad 實現模組化狀態處理的需求，但是原本是單純的函數構成的程式本體，卻也必須跟著改寫成 monadic 方式，如此一來，大大減損了使用 aspect 模組化的優勢。所以未來一個重點就是要探討如何在 AspectFun 直接提供具備狀態處理功能的剖面 (side-effecting aspects)，透過編譯技術將其與程式本體自動轉換成 monadic 方式，以提高純粹函數程式的處理狀態的模組化程度。

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

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 34–51, 2007.

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 determine *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.

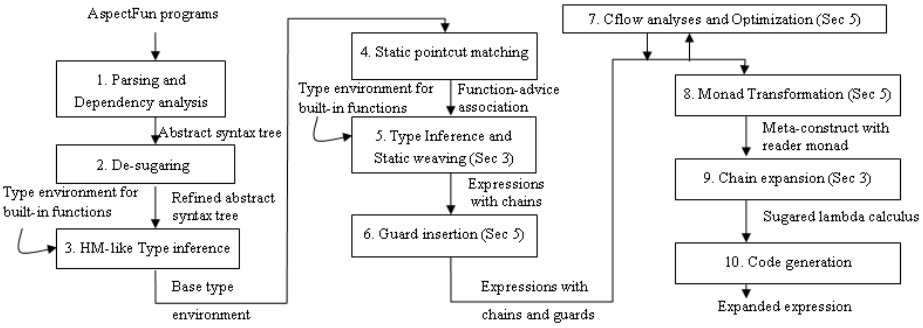

Fig. 2. Compilation Model for AspectFun

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-order*

[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.

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 abbreviation

[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.

for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and 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	$::= d$ in $\pi \mid e$
Declarations	d	$::= x = e \mid f\ \overline{x} = e \mid n$@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	$::=$ `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	$::= \forall \bar{a}.\rho$

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

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

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.

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

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

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

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

$$
(\text{EXPR:VAR})\ \frac{\tau = \Delta(x)}{\Delta \vdash x : \tau \rightarrowtail x}\ (\text{EXPR:TY-APP})\ \frac{\forall\overline{\alpha}.\ \tau = \Delta(x) \quad \tau_x = [\overline{\tau'}/\overline{\alpha}]\tau}{\Delta \vdash x : \tau_x \rightarrowtail x\{\overline{\tau'}\}}
$$

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

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

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

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:

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

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

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

$$\text{JPMatch}(f, pc) = (f \equiv pc)$$

Fig. 6. Operational Semantics for 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.

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].

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

Expressions:

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

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

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

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

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

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}.

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 $S(t_1 \to t_2)$ 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:

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

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

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

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

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

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.

$$
\begin{array}{lll}
e_M & : & \text{Expressions containing meta-constructs} \\
\text{addProceed} & : & e_M \longrightarrow e_M \\
\text{addProceed}(\mathsf{let}\ n\ df\ arg\ =\ e_1\ \mathsf{in}\ e_2) & = & \mathit{if}\ (n \text{ is an advice})\ \mathit{then} \\
 & & \quad \mathsf{let}\ n\ df\ \mathit{proceed}\ arg\ =\ e_1 \\
 & & \quad \mathsf{in}\ \text{addProceed}(e_2) \\
 & & \mathit{else}\ \mathsf{let}\ n\ df\ arg\ =\ e_1\ \mathsf{in}\ \text{addProceed}(e_2) \\
\text{addProceed}(e) & = & e
\end{array}
$$

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

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

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

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 $\Downarrow_{\mathcal{A}}^{\mathcal{S}}$, 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 $(\!|e, S|\!)$:

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

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

$(\!|e, \mathcal{S}|\!)$ 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
```

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

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 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.

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. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising Aspect J. In: PLDI '05. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, pp. 117–128. ACM Press, NewYork (2005)
2. Chen, K., Weng, S.-C., Wang, M., Khoo, S.-C., Chen, C.-H.: 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. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: PolyAML: A polymorphic aspect-oriented functional programmming language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
4. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. ACM Transactions on Programming Languages and Systems (TOPLAS) (to appear, 2006)
5. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming (to appear, 2006)
6. Jones, M.P.: Qualified Types: Theory and Practice. D.phil. thesis, Oxford University (September 1992)
7. M.P. Jones.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming, pp. 97–136 (1995)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
9. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: CC, pp. 46–60 (2003)
10. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: Proc. of ICFP'05. ACM Press, NewYork (2005)
11. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc, Secaucus, NJ, USA (1999)
12. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 59–68. ACM Press, NewYork (2005)
13. Sereni, D., de Moor, O.: Static analysis of aspects. In: Aksit, M. (ed.) AOSD. 2nd International Conference on Aspect-Oriented Software Development, pp. 30–39. ACM Press, NewYork (2003)
14. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, pp. 60–76. ACM, NewYork (1989)
15. Wang, M., Chen, K., Khoo, S.-C.: Type-directed weaving of aspects for higher-order functional languages. In: PEPM '06. Workshop on Partial Evaluation and Program Manipulation, ACM Press, NewYork (2006)