On the Pursuit of Static and Coherent Weaving

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

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

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

Abstract

Aspect-oriented programming (AOP) has been shown to be a useful model for software development. Special care must be taken when we try to adapt AOP to strongly typed functional languages which come with features like type inference mechanism, polymorphic types, higher-order functions and *type-scoped* pointcuts. Specifically, it is highly desirable that weaving of aspect-oriented functional programs can be performed statically and coherently. In [13], we showed a type-directed weaver which resolves all advice chainings coherently at static time. The novelty of this paper lies in the extended framework which supports static and coherent weaving in the presence of polymorphic recursive functions, advising advice bodies and higher-order advices.

1. Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system [7]. In AOP, a program consists of many functional modules and some *aspects* that encapsulate 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 behavior 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 behavior of those functional modules impacted by aspects being modified accordingly.

Two highly desirable properties of weaving are it being *static* and *coherent*. Static weaving refers to making as many weaving decisions at compilation time as possible, usually by static translation to a "less-aspect-oriented" program. A direct benefit out of static weaving is that less run-time checking overhead is required. In addition, a weaver should allow different invocations of a function with inputs of the same type to be advised with the same set of advices. This property is known as coherence. Coherent weaving is also crucial as it ensures correct and understandable behavior of programmes. However, it is far from straightforward to bring these two properties together particularly under a strongly typed functional language setting. Let's consider a small example to have a feel of the intricacy involved.

Example 1

```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg) = proceed arg in
let h x = x in
let f x = h x in
(f 1) + (h 2)
```

This piece of code defines two pieces of advice namely `n1` and `n2`; it also defines a main program consisting of declarations of `f` and `h` and a main expression specifying applications of `f` and `h`. The first advice, `n1`, designates execution of `h` as its pointcut. It also contains a type constraint, which is called a *type scope*, attached to the first argument. `n1` is only triggered when `h` is executed with an Int argument. On the other hand, the pointcut of `n2` is not constrained by a type-scope. Thus all executions of function `h` match the pointcut. Consequently, pointcuts of `n1` and `n2` overlap in that the former is subsumed by the latter. In general, it is not possible to determine locally if a particular advice should be triggered. Let's consider the main program of the above example.

From a syntactic viewpoint, function `h` will be called in the body of `f`. If we naively infer that the argument `x` to function `h` in the RHS of `f`'s definition is of polymorphic type, we will be tempted to conclude that (1) advice `n2` should be triggered at the call, and (2) advice `n1` should not be called as its type scope is less general than $\forall a.a \rightarrow a$. As a result, `n2` will be statically chained to the call to `h`.

Unfortunately, this approach will cause incoherence behavior of `h` at run-time. Specifically, in the main expression, `(h 2)` will trigger both advices `n1` and `n2`. On the other hand, `(f 1)` in the main expression will actually pass integer argument `1` to `h`. There, triggering of `n1` is missed out since the weaver has mistakenly committed its choice in the definition of `f`. The only coherent behavior of a weaver in this case is to have `h` being advised by both `n1` and `n2`, during both invocations of `h`, *i.e.*, `(h 1)` and `(h 2)`.

It appears that the goals of achieving static weaving while ensuring coherent weaving are not in tandem here. In PolyAML [4], dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [9] takes a syntactic approach which sacrifices coherence for static weaving.

In our earlier work [13], we designed a static weaving strategy that smoothly incorporates essential features of aspects into a core functional language with parametric polymorphism and higher-order functions. In contrast with the work done on PolyAML and Aspectual Caml, our strategy synthesizes functional core and aspects during compilation, thus successfully reconciling the desires to be static and to be coherent. The central idea there is to make full advantage of type information, both from the base program and the type-scoped pointcuts, to guide the weaving of aspects. Specifically, it advocates a source-level type inference system for a higher-order, polymorphic language coupled with type-scoped pointcuts.

A type-directed translation scheme is then devised to resolve all advice applications, thus eliminating any future need for dynamic type checking. The translation removes advice declarations from source programs and produces translated codes in an intermediate language which is essentially polymorphically typed lambda calculus with a small extension. The program in example 1 is translated as follows.

```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed arg in
let h x = x in
let f dh x = dh x in
(f <h,{n1,n2}> 1) + (<h,{n1,n2}> 2)
```

Note that all advice declarations are translated into functions and are woven in. A special syntax $\langle _ , \{\ldots\} \rangle$ is used to chain together advices and advised functions. For instance, $\langle \texttt{h} , \{\texttt{n1}, \texttt{n2}\} \rangle$ denotes the chaining of advices `n1` and `n2` to advised function `h`. In the above example, the two invocations of `h` in the original aspect program have been translated to an invocation of the chained function $\langle \texttt{h} , \{\texttt{n1}, \texttt{n2}\} \rangle$. This shows that our weaver respects the coherence property.

This coherent weaving of advices to `h` entails passing appropriate chained expressions of `h` to those function calls in the program text from which `h` may be called indirectly. This requirement is satisfied by allowing functions of those affected calls to carry extra parameters. In the code above, the translated definition of function `f` carries such an additional parameter, `dh`. The original `(f 1)` call is then translated to `(f <h,n1,n2> 1)`, in which the chained expression for `h` is passed.

In this paper, we re-engineer our type system and translation scheme to handle recursive functions, advising advice bodies and higher-order advices; this gives full-fledged support of aspects.[1] Previously, functions and advices in our framework were treated very differently. In particular, advices cannot be the targets of advising, neither directly by another advice nor indirectly through calls to advised functions in advice bodies. While completely blurring the distinction between functions and advices is not desirable, maintaining an unnecessary wide gap between them can also make aspect programming overly restrictive. As well argued by Rajan et al [11], two-layered models of advice and function cannot provide proper modularization for higher-order crosscutting concerns. Therefore, we refine our framework by devising new typing and translation rules that handle both advising advice bodies and higher-order advices, i.e., advices advising other advices.

The main contributions of this paper are:

- A translation scheme that enables *static and recursive weaving* of advices into recursive function definitions.
- A set of novel type rules with the support of an intermediate language that ensure static and coherent weaving of advanced aspect-oriented features. Specifically,
  - The weaving of *advices into other advices' bodies*. Static and coherent weaving of such advices has been challenging because the decision for weaving is only known after the context of invoking the underlying advice is known.
  - The weaving of *higher-order advices*. These are advices that advise other named advices. Such feature demands a uniform typing and translation scheme that not only infer the types of both functions and advices consistently but also weave in proper advices in a cascading manner according the type context.

[1] Previously supported features such as higher-order functions, curried pointcuts and any pointcut are compatibly supported in the new system even though a discussion of them are omitted from this paper.

The outline of the paper is as follows: Section 2 describes an aspect-oriented language and provides background information and terminologies used. In Section 3, we describe the intermediate language to be used as target of type-directed weaving. Section 4 describes our type-directed weaving algorithm, and presents our solutions to the handling of giving advice to both recursive functions and other advices, and the handling of higher-order advices. Section 5 surveys related work done in this field, and Section 6 concludes our work.

2. Aspect Language

In this section, we introduce an aspect-oriented functional language for our investigation. We shall focus on only some essential features of aspects, namely, *around* advice with *execution* pointcuts. Note that we drop the description of some features discussed previously in [13], namely higher-order functions, curried pointcuts and *any* pointcuts, as they are orthogonal to the discussion of this paper. However, it should be understood that they are still safely supported by the new system. The following syntax specifies the language.

Expressions	e	::=	$x \mid \lambda x.e \mid e\ e \mid$
			$\mathsf{let}\ f\ =\ e\ \mathsf{in}\ e \mid proceed \mid$
			$n@\mathsf{advice\ around}\ pc = e\ \mathsf{in}\ e$
Arguments	arg	::=	$x \mid x :: t$
Pointcuts	pc	::=	$\{\overline{jp}\}\ (arg)$
Joinpoints	jp	::=	$f \mid n$

We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. expressions, types etc). Note that we generally assume $\bar{o}$ and o denotes non-related objects which should not be confused. The term $[\overline{o/a}]o'$ denotes simultaneous substitution of o_i for variables a_i in o', for $i = 1, \ldots, n$. We write $t_1 \sim t_2$ to specify equality between two types t_1 and t_2 (*a.k.a*, unification) to avoid confusing with assignment $=$. We write $fv(o)$ to denote the free variables in some object o.

For simplicity, we leave out type annotations, user defined data types, `if` statements and patterns but may make use of them in examples. Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment.

In our language, an aspect is an advice declaration which includes a piece of advice and its target *pointcut*. Pointcuts are represented by $\{\overline{jp}\}\ (arg)$ where jp stands for joinpoints, comprising f, ranging over functions, and n, ranging over advices. A pointcut describes the point in time any function or advice from the set is executed. Usually function names are included in the pointcuts to designate the target functions for advice weaving. Since advices are also named, we allow advices advising other advices, i.e., higher-order advices. The argument variable *arg* is bound to the actual argument of the function execution and it may contain an annotated type.

Advice is a function-like expression that executes before, after, or around a pointcut. Note that *around* advice executes in place of the indicated pointcut, allowing a function to be replaced. A special function *proceed* 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 pointcut. It is easy to see that both *before* advice and *after* advice can be simulated by *around* advice that always *proceeds*. Therefore, our aspect language only needs to support *around* advice.

There are two things about our pointcuts that merit further discussion. Firstly, the pointcut designator `around pc` represents the point in time when the functions or advices in `pc` are about to execute. Like the *execution* pointcuts of AspectJ, these pointcuts cover the cases when functions are explicitly invoked as well as those

when they are implicitly called. They are necessary for languages with functions as first-class values, for, in such languages, functions can be applied directly through name-based invocation as well as indirectly through aliasing and functional arguments to a higher-order function. The following simple program illustrates the situations.

```
n@advice around {f} (arg) = e in
let f x = x in
let g = f in
let h k x = k x in
(f True, g 'a', h g 3)
```

Clearly, in this example, if we look for only the function calls made to `f`, following the call pointcuts of AspectJ, we will not be able to capture the applications of `f` through `g` and `k`. However, deriving a static weaving scheme for advices on execution pointcuts in a statically typed functional language is not as easy as it may appear. In AspectJ, the pointcut designator, `execution f (arg)`, will direct the weaver to insert the advice call into the body of `f`. As a result, the invocations of `f` through `g` and `k` will also trigger the advice. However, this naive approach will encounter great static typing difficulties when handling advices with additional type constraints, which is strongly related to the following discussion on type-scoped advices.

Secondly, as well demonstrated in [4] and [9], it is very often that we need to have advices with type constraints to confine the applicable scope of such advices. Our aspect language support such advices as it allows type constraints to be imposed on the arguments of those functions occurring in pointcuts. We call such pointcuts *type-scoped pointcuts*. Advices with type scoped pointcuts are henceforth called *type-scoped advices*. However, having such type-scoped advices in a statically typed language will pose significant challenges for advice weaving, since it calls for a smooth reconciliation between type-based advice dispatch and static weaving. In our opinions, previous work did not address this issue adequately. In designing Aspectual Caml, Masuhara et al. also suggest using execution pointcuts to handle indirect function calls. But they followed the weaving scheme of AspectJ by inserting a call to the associated advice in the advised function. Apparently, this scheme will only work for monomorphic functions; dynamic type-dispatch is needed to support polymorphic yet type-scoped advices. This may also partly explain why Dantas et al. include runtime type analysis mechanism in their design of PolyAML. By contrast, our aspect language supports type-scoped advices while retaining both static typing and static weaving.

The following syntax defines the type expressions in our aspect language.

Types	t	$::=$	$a \mid t \rightarrow t$
Type Schemes	σ	$::=$	$\forall \bar{a}.\rho$
Advised Types	ρ	$::=$	$(x : t).\rho \mid t$

Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment. Central to our approach is the construct of *advised types*, inspired by the *predicated types* [12] used in Haskell's type classes. These advised types augment common type schemes with *advice predicates*, which are used to capture the need of future advice weaving dependent on the type context. For example, the type scheme for the function *g* in the above example will be $\forall a.(f : a \rightarrow a).a \rightarrow a$, which indicates that whenever *g* is applied in a specific context, the advices on *f* will also be triggered. We shall explain them in detail in Section 4.

In the next a few (sub)sections, we show how the features discussed in this paper are used when programming with aspects. The challenges in incorporating them into a static and coherent weaving framework are also outlined.

2.1 Recursive Functions

Recursive functions are widely used in functional programming. When type-scope advices are defined on a polymorphic recursive function, it may yield an interesting advised type which has a predicate refers the function itself. Let's consider a small example for illustration.

Example 2

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```

The function `f` above defines a generic traversal of an input list. When the input list contains *Int* elements, advice `n` intercepts the execution and applies function `g` to the list head. Thus, it simulates the behavior of the standard *map* function.

In an AO system which performs weaving by static translation, the definition of function `f` should be translated into an expression with relevant advices chained. However, because of the recursion, the translation of `f` requires a translated definition of itself which results into a cyclic process!

A syntactic weaver may sees this matter from a syntactic view by chaining advices into the type-annotated abstract syntax tree. In that case, coherence is lost as the execution of the recursive calls may be chained with a different set of advices other than those of the initial call even when the recursion is monomorphic.

Let's consider the program in Example 2. The typed AST annotates the initial `f`-call in the main expression with type $[Int] \rightarrow [Int]$ and the recursive `f`-calls in the definition of `f` with type $[a] \rightarrow [a]$. Thus, the former is chained with advise `n` whereas the later is not even though both receive arguments of type $[Int] \rightarrow [Int]$ during the actual execution.

In Section 4, we show how a fixed point combinator can be employed to achieve coherent weaving of recursive functions.

2.2 Nested Advices

Aspects are not limited to observing base programs. Inside the bodies of advice definitions, there may be calls to other functions that are advised. We call these *nested advices*.

The program in example 2 increments all the elements of a list by one through the interception of aspect `n`. However, when `f` is called with the empty list `[]`, the program crashes as the advice attempts to extract the head from `[]` before the test of list length in the body of `f` is performed.

To remedy this safety violation, we may implement a patch aspect by setting the head of `[]` to an invalid bit, say -1.

```
n1@advice around {head} (arg:[Int])
   = if arg == [] then -1 else proceed arg
```

Note that advice `n1` advises on a function called inside the body of an advice. In another words, `n1` is a nested advice.

This small example sheds light on a wide range of applications of nested advices. When aspects are used for enforcing safety and security concerns, it is important that the advices are applied to every execution of the target functions. Therefore, nested advices becomes the essential feature which supports this behavior.

Note that we do not allow circular *around* advices that apply to the execution of their own bodies, directly or indirectly. The reason for this restriction is that circular *around* advices together with

potential recursive functions that they are advising may form a scenario similar to polymorphic mutual recursion which threatens decidability of type inference. We leave this for future investigation.

Even without circular *around* advices, weaving nested advices statically is a challenging task primary for the following two reasons.

1. Advice chainings only appear in the woven program which is not a subject for further weaving. A syntactic approach to solve this problem is to have an iterative process which repetitively feeds the woven program back to the weaver until no more advice can be woven. One side condition for this approach is that both input and output of weaver are from the same language.
2. The typing context where an advice `n` is chained may not be sufficiently specific for another advice to be chained to the calls inside `n`'s body. This complicates coherent weaving.

In Section 4, we show in details how our translation works coherently without the need of iteratively feeding the woven program back to the weaver.

2.3 Higher-Order Advices

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 [11]. The idea of a multi-layered design dates back to [5, 1, 10]; and this is sometimes called *higher-order advices*.

In Section 2.2, we use a piece nested advice to patch an unsafe program. However, the result is not completely satisfactory as `n1` always inserts an extra invalid bit into the result list. The root of the problem is the inability of advising an advise directly. In this section, we show a solution with a higher-order advice which cause no undesirable side effects such as the extra bit.

```
n2@advice around {n} (arg)
   = if arg == [] then [] else proceed arg
```

Advice `n2` advise directly on `n` which allows us to short circuit the `head`-call when the input is `[]`.

There has been some argument that higher-order advices can be simulated by nested-advices. Take AspectJ as an example. Advices are nameless in AspectJ, hence we cannot directly advise another advice. Instead, if we know there are such requirements in advance, we can shape the target advice for advice nesting as follows: Move its entire advice body into a help method, and write a piece of advice that advise this help method, thus achieving the effect of advising advices to a certain degree. But there are at least two shortcomings using this way of simulation. Firstly, this only works for *before* and *after* advices, because *proceed* will take effect only when it occurs inside an around advice. Thus, the example given above cannot be handled. Secondly. This scheme does not scale up well. What if later we want yet another third-order advice on the second-order one?

Besides being used as patches for other advices, higher-order advices are also useful as development aspects. Let's say we want to compute the total amount of a customer order and apply discount rates according to certain regular rules as follows.

Example 3

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

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

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

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

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

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

Weaving higher-order advices involves allowing advices to be advised as functions. This adds in another layer of complexity to the translation. Again, we refer the readers to Section 4 for a detailed discussion of the solution.

3. Intermediate Language

Our type-directed weaving produces codes in an intermediate language, which explicitly expresses the chaining of advices. The intermediate language is based on a polymorphically typed lambda calculus plus let introductions, extended with chaining expressions which are used to model advice invocations triggered by function calls.

3.1 Operational Semantics

The syntax of the language is displayed below.

$$\begin{array}{lrcl} \text{Values} & v & ::= & \lambda x.e \mid \mu f.e \mid \langle v, \{\bar{v}\}\rangle \\ \text{Expressions} & e & ::= & v \mid x \mid \mathit{proceed} \mid \lambda x.e \mid e\ e \mid \langle e, \{\bar{e}\}\rangle \\ & & & \mid \mathsf{let}\ f\ =\ e\ \mathsf{in}\ e \end{array}$$

There is no notion of advices in this language as they are modelled straightforwardly as functions. Pointcuts are also not necessary since we assumes all advices are already woven in place. A chaining expression of the form $\langle e, \{\bar{e}\}\rangle$ consists of an expression e which evaluates to a function (or an advice) and a chain of expressions $\bar{e}$ which evaluates to advices to be triggered by the function call. We call e occurring at the left component the *active* function/advice, and $\{\bar{e}\}$ the *dormant* advices. When both e and $\bar{e}$ are values, the chaining expression is itself a value.

The set of β reductions are defined as follow:

$$\begin{array}{lcl} (\lambda x.e\ v) & \longmapsto_\beta & (e[v/x]) \\ (\mu f.e\ v) & \longmapsto_\beta & (e[\mu f.e/f]\ v) \\ (\mathsf{let}\ x = v\ \mathsf{in}\ e) & \longmapsto_\beta & (e[v/x]) \\ (\langle v, \{\}\rangle\ v') & \longmapsto_\beta & (v\ v') \\ (\langle v, \{v_1, \bar{v}\}\rangle\ v') & \longmapsto_\beta & (v_1[\langle v, \{\bar{v}\}\rangle/\mathit{proceed}]\ v') \end{array}$$

These rules specifies a call-by-value evaluation strategy which is orthogonal to the language design. The first three rules are standard β-rules for lambda calculus. In the fourth rule, when the advice sequence is empty, the chaining returns the original function. Otherwise, as shown in the last rule, the chaining replaces the *proceed* in the first advice in sequence by a value which chains the function v with the remainder of the advice sequence.

The substitution operation $e[v/x]$ performs the usual substitution of values for variables, with one exception: When the variable being substituted is *proceed*, and the expression is a chained expression, then the corresponding substitution is performed only on the active expression, but not the dormant expressions:

$$\langle e, \{\bar{e}\}\rangle[v/proceed] \equiv \langle e[v/proceed], \{\bar{e}\}\rangle.$$

Note that the dormant advices above, $\{\bar{e}\}$, have not been substituted, because the *proceed* is bound to the existing active expression e, not the dormant expression $\bar{e}$. This point is particularly important in the case of second-order advice, which will have different *proceed* value from the advice which the former is advising.

3.2 Type System

Programs produced in the intermediate language first undergo α-conversion. This frees the programs from scoping concerns. Consequently, the program can be type-checked for its correctness. The type system is defined in Figure 1.

$$(\textsc{Var})\quad \frac{x:\sigma \in \Gamma}{\Gamma \vdash_i x:\sigma} \qquad (\textsc{Chain})\quad \frac{\Gamma, proceed : t' \vdash_i \bar{e} : \bar{t} \quad \Gamma \vdash_i e : t' \quad t' \trianglelefteq \bar{t}}{\Gamma \vdash_i \langle e, \{\bar{e}\}\rangle : t'}$$

$$(\textsc{Abs})\quad \frac{\Gamma, x : t_1 \vdash_i e : t_2}{\Gamma \vdash_i \lambda x.e : t_1 \to t_2} \qquad (\textsc{Fix})\quad \frac{\Gamma, f : t \vdash_i e : t}{\Gamma \vdash_i \mu f.e : t}$$

$$(\forall\textsc{Elim})\quad \frac{\Gamma \vdash_i e : \forall a.\sigma}{\Gamma \vdash_i e : [t/a]\sigma} \qquad (\textsc{App})\quad \frac{\Gamma \vdash_i e_1 : t_1 \to t_2 \quad \Gamma \vdash_i e_2 : t_2}{\Gamma \vdash_i e_1\ e_2 : t_2}$$

$$(\forall\textsc{Intro})\quad \frac{\Gamma \vdash_i e : \sigma \quad a \notin \Gamma}{\Gamma \vdash_i e : \forall a.\sigma}$$

$$(\textsc{Let})\quad \frac{\Gamma, proceed : t', f : \sigma \vdash_i e_1 : \sigma \quad \Gamma, f : \sigma \vdash_i e_2 : t \quad \sigma \trianglerighteq t'}{\Gamma \vdash_i \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t}$$

Figure 1. Typing Rules

There is no introduction of new type syntax other than the one of the standard polymorphically typed lambda calculus.

Types	t	::=	$a \mid t \to t$
Type Schemes	σ	::=	$\forall \bar{a}.t$

The typing rules are presented in Figure 1 which are mostly standard except that type bindings of *proceed* is needed for function definitions introduced by let. The reason for this is that advices, which may contain *proceed*-calls, appear as functions in the intermediate language. In Rule (CHAIN), the advices are typed under the assumption that *proceed* is an instance of the function. We also require the advices have types more general than that of the function. We say a type scheme is more general than the other if it can be instantiated to the latter via variable substitutions. The relation is formally defined as:

$$(\textsc{Gen})\quad \frac{[\bar{t}/\bar{a}]t_1 \sim t_2}{\forall \bar{a}.t_1 \trianglerighteq \forall \bar{b}.t_2}$$

The type system enjoys the standard safety properties.

Theorem 1 (Progress) *If $\vdash_i e : \sigma$, then either e is a value or else there is some e' with $e \longmapsto_\beta e'$.*

Theorem 2 (Preservation) *If $\Gamma \vdash_i e : \sigma$ and $e \longmapsto_\beta e'$, then $\Gamma \vdash_i e' : \sigma$.*

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. For instance, in the main program given in Example 1, 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 type which should be no more general than* $a \to a$.

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 applications of advised functions in its definition.

$$(\textsc{AErase})\quad [\![\forall \bar{a}.(x:t).\rho]\!] = [\![\forall \bar{a}.\rho]\!] \quad [\![\forall \bar{a}.(x:t).t']\!] = \forall \bar{a}.t'$$

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

$$(\textsc{Card})\quad |o_1 ... o_k| = k \qquad (\textsc{Card}_p)\quad |\forall \bar{a}.\bar{p}.t|_{pred} = |\bar{p}|$$

Figure 2. Auxiliary Definitions

Figure 2 defines a set of auxiliary functions/relations that assists type inference. The letter t ranges over unification (type-)variables which are distinct from quantified rigid type variable a. Rule (AERASE) defines a function $[\![\cdot]\!]$ which removes all advice predicates from an advised type scheme. We also define, in rule ($\textsc{Gen}_F$), a generalization procedure which turns a type into a type scheme by quantifying type variables that do not appear free in the type environment. The (CARD) function, denoted by $|\cdot|$, returns the cardinality of a sequence of objects. The ($\textsc{Card}_p$) function returns the number of advice predicates in a type scheme.

The main set of type inference rules, as described in Figure 3, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \sigma \rightsquigarrow e'$ to denote that expression e has type σ under type environment Γ and it is translated to e'. We assume that the advice declarations are preprocessed and all the names which appear in any of the pointcuts are recorded in an initial global store A. We also assume that the base program is well typed in Hindley-Milner and the type information of all the functions are stored in Γ_{base}.

The typing environment Γ contains not only the usual type bindings (of the form $x : \sigma \rightsquigarrow e$) but also *advice bindings* of the form $n : \sigma \bowtie \bar{x}$. This states that an advice with name n of type σ is defined on $\bar{x}$. We may drop the $\bowtie \bar{x}$ part when it is not relevant. When the bound variable is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish from the non-advised case. We also use the notation $:_{(*)}$ to represent a binding which is either : or $:_*$.

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

4.1 Predicating and Releasing

There are two rules for variable lookups. Rule (VAR) is standard. In the case that variable x is advised, rule (VAR-A) will check all advices defined on x (we do not distinguish : and $:_*$-binding for

$$\text{(VAR)}\quad \frac{x : \sigma \rightsquigarrow e \in \Gamma}{\Gamma \vdash x : \sigma \rightsquigarrow e} \qquad \text{(VAR-A)}\quad \frac{\begin{array}{c} x :_* \sigma_x \in \Gamma \quad [\![\bar{\sigma}]\!] \not\trianglelefteq [\![\sigma']\!] \quad \Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i \\ \bar{n} :_{(*)} \bar{\sigma} \bowtie x \rightsquigarrow \bar{n}' \in \Gamma \quad \{n_i \mid [\![\sigma_i]\!] \trianglerighteq [\![\sigma']\!]\} \\ |\bar{y}| = |\sigma_x|_{pred} \quad \bar{y}\ is\ fresh \quad \sigma_x \trianglerighteq \sigma' \end{array}}{\Gamma \vdash x : \sigma' \rightsquigarrow \lambda \bar{y}.\langle x\ \bar{y}\ , \{e_i\}\rangle}$$

$$\text{(}\forall\text{ELIM)}\quad \frac{\Gamma \vdash e : \forall a.\sigma \rightsquigarrow e'}{\Gamma \vdash e : [t/a]\sigma \rightsquigarrow e'} \qquad \text{(}\forall\text{INTRO)}\quad \frac{\Gamma \vdash e : \sigma \rightsquigarrow e' \quad a \notin \Gamma}{\Gamma \vdash e : \forall a.\sigma \rightsquigarrow e'} \qquad \text{(APP)}\quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : t_1 \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : t_2 \rightsquigarrow (e_1'\ e_2')}$$

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

$$\text{(PRED)}\quad \frac{x :_* \sigma_x \in \Gamma \quad t \trianglelefteq [\![\sigma_x]\!] \quad \Gamma, x : t \rightsquigarrow x_t \vdash e : \rho \rightsquigarrow e_t' \quad x \in A}{\Gamma \vdash e : (x : t).\rho \rightsquigarrow \lambda x_t.e_t'} \qquad \text{(REL)}\quad \frac{\Gamma \vdash e : (x : t).\rho \rightsquigarrow e' \quad \Gamma \vdash x : t \rightsquigarrow e'' \quad x \in A \quad x \neq e}{\Gamma \vdash e : \rho \rightsquigarrow e'\ e''}$$

$$\text{(FIX)}\quad \frac{\Gamma, f :_{(*)} \rho \rightsquigarrow f \vdash e : \rho \rightsquigarrow e'}{\Gamma \vdash \mu f.e : \rho \rightsquigarrow e'} \qquad \text{(REL-F)}\quad \frac{\Gamma \vdash f : (f : t).\rho \rightsquigarrow e' \quad F\ fresh \quad f \in A}{\Gamma \vdash f : \rho \rightsquigarrow \mathsf{let}\ F = (e'\ F)\ \mathsf{in}\ F}$$

$$\text{(ADV)}\quad \frac{\Gamma, proceed : t \vdash \lambda x.e_a : \bar{p}.t \rightsquigarrow e_a' \quad f_i : \sigma' \in \Gamma_{base} \quad \sigma' \trianglelefteq [\![\sigma]\!] \quad \Gamma, n :_{(*)} \sigma \bowtie \bar{f} \rightsquigarrow n \vdash e : t' \rightsquigarrow e' \quad \sigma = gen(\Gamma, \bar{p}.t)}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x) = e_a \mathsf{in}\ e : t' \rightsquigarrow \mathsf{let}\ n = e_a'\ \mathsf{in}\ e'}$$

$$\text{(ADV-AN)}\quad \frac{\Gamma, proceed : t \vdash \lambda x : t_x.e_a : \bar{p}.t \rightsquigarrow e_a' \quad f_i : \forall \bar{a}.t_i \to t_i' \in \Gamma_{base} \quad t_x \trianglelefteq \forall \bar{a}.t_i \quad (t_i \to t_i') \sim t \quad \Gamma, n :_{(*)} \sigma \bowtie \bar{f} \rightsquigarrow n \vdash e : t' \rightsquigarrow e' \quad \sigma = gen(\Gamma, \bar{p}.t)}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: t_x) = e_a \mathsf{in}\ e : t' \rightsquigarrow \mathsf{let}\ n = e_a'\ \mathsf{in}\ e'}$$

Figure 3. Type-directed Weaving by translation

these advices here) to see whether any of them has a more specific type than x's. This is to ensure that chaining of advices is only done in a sufficiently specific context. We call this check *sufficiently specific context check*, and it is expressed in the rule as the guard $[\![\bar{\sigma}]\!] \not\trianglelefteq [\![\sigma']\!]$ (the relation $\not\trianglelefteq$ is defined in Section 3.2). If the check succeeds (*i.e.*, no advice has a more specific type than x), x will be chained with the translated forms of all those advices defined on it, having the same or more general types than x has. We give all these selected advices a non-advised type in the translation of them $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$. This ensures correct weaving of nested advices advising the bodies of the selected advices. The detail will be elaborated in Section 4.4. Finally, the final translated expression is *normalized* by bringing all the advice abstractions of x outside the chain $\langle \ldots \rangle$. This ensures type compatibility between the advised call and its advices as required by the type system of the intermediate language.

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

Before we describe rules (PRED) and (REL) in detail, we illustrate the application of these rules by derving the type and the woven code for the program shown in Example 1. During the derivation of the definition of f, we have:

$$\Gamma = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h, n_1 : I \to I \bowtie h\}$$

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

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

$$\Gamma_3 = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,\ n_1 : I \to I \bowtie h, f : \forall a.(h : a \to a).a \to a \rightsquigarrow f\}$$

$$\begin{array}{ll} \text{(VAR)} & \dfrac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : I \to I).I \to I \rightsquigarrow f} \quad ⓐ \\ \text{(REL)} & \dfrac{}{\Gamma_3 \vdash f : I \to I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle)} \quad \ldots \\ \text{(APP)} & \dfrac{}{\Gamma_3 \vdash (f\ 1) : I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle\ 1)} \end{array}$$

$$ⓐ = \quad \text{(VAR-A)}\quad \frac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad \ldots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow \langle h\ , \{n_1, n_2\}\rangle}$$

We note that rules (ABS),(LET), (APP), (∀INTRO) and (∀ELIM) are rather standard, with the tiny exception that rule (LET) will bind f with : when it is not in A; and with $:_*$ otherwise.

Rules (PRED) and (REL) respectively introduces and eliminates advice predicates just as (∀INTRO) and (∀ELIM) do to bound type variables. Rule (PRED) adds an advice predicate to a type (Note that we only allow sensible choices of t constrained by $t \trianglelefteq [\![\sigma_x]\!]$). 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.

4.2 Advising Recursive Functions

Now let's consider Example 2 given in Section 2.1 where the advised function `f` is recursive. The code is reproduced below.

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```

In our type system, rule (FIX) is used to type and translate recursive functions. In this above example, our translation produces an interesting advised type $\forall a.(f : [a] \rightarrow [a]).[a] \rightarrow [a]$ for `f`. If Rule (REL) is applied to release this type, the translation will not terminate as the derivation of $\Gamma \vdash f : [a] \rightarrow [a]$ depends on itself. The solution is to break the loop by using a fixed point combinator as the translation result. This is manifested in Rule (REL-F), by which example 2 is translated to the following:

```
let g x = x + 1 in
let n = \arg.(Cons (g (head arg)) (proceed arg)) in
let f df x = if (length x) > 0
             then df (tail x) else x in
(let F = \y.<f y,{n}> F in F) [1,2,3]
```

By a simple Let-lifting, we lift the local definition of `F` to the top level. The final translation result is:

```
let g x = x + 1 in
let n = \arg.(Cons (g (head arg)) (proceed arg)) in
let f df x = if (length x) > 0
             then df (tail x) else x in
let F = \y.<f y,{n}> F in
F [1,2,3]
```

The fixed point combinator `F` correctly captures the desired behavior by chaining every execution of `f` with `n`. In the following, we sketch the evaluation steps for the main expression `F [1,2,3]` based on the operational semantics given in Section 3.

For the sake of presentation, some long expressions are renamed as follows.

$v_1 = \backslash$`x.if (length x) > 0 then F (tail x) else x`
$v_2 = \backslash$`arg.(Cons (g (head arg)) (`v_1` arg))`

We also use $\longrightarrow_\beta{}^*$ to represent multiple steps of β reduction.

$$
\begin{array}{ll}
 & \texttt{F}\ [1,2,3] \\
\longrightarrow_\beta & (\backslash\texttt{y.}\langle\texttt{f y}\ ,\ \{\texttt{n}\}\rangle\ \texttt{F})\ [1,2,3] \\
\longrightarrow_\beta & \langle\texttt{f F}\ ,\ \{\texttt{n}\}\rangle\ [1,2,3] \\
\longrightarrow_\beta & \langle v_1\ ,\ \{\texttt{n}\}\rangle\ [1,2,3] \\
\longrightarrow_\beta & v_2\ [1,2,3] \\
\longrightarrow_\beta{}^* & \texttt{Cons 2 (F}\ [2,3]) \\
\ldots &
\end{array}
$$

4.3 Handling Advices

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 most specific context 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 all functions in the pointcut have type schemes that are not more general than the advice's. Note that the type information of all the functions are 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. We also require the type of all functions in the pointcut to be unifiable to the advice type, so that any bogus advices which can never be safely triggered will be rejected by our type system.

Note that we do not allow the annotated type t_x to be more general than the input type of any function in the pointcut, as this will be contrary to the intention of type-scoped advices.

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 chain all the advices. We illustrate this with an example.

Example 4

```
n1@advice around {f} (arg::Int) = e1 in
n2@advice around {f} (arg) = e2 in
let f x = x in
n3@advice around {g} (arg) = f arg in
let g x = x in
let h x = g x in
h 1
```

Here, advice `n3` calls `f` which is in turn being advised. The goal of our translation is to chain advices which are applicable to the call of `f` inside an advice. Concretely, when a call to `g` is chained with advice `n3`, the body of `n3` must also be advised. Moreover, the choice of advices must be coherent.

At the time when the declaration of `n3` is translated, the body of the advice is translated. An advised type is given to it since the currently context is not sufficiently specific.

When the translation attempts to chain an advice in Rule (VAR-A), the judgment $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$ in the premise forces the advice to have a non-advised type. This is to ensure that all the advice abstractions are fully released so that chaining can take effect.

In the case that this derivation fails, it signifies that the current context is not sufficiently specific for advising some of the calls in this advice's body, and chaining has to be delayed. In example 4, the call to `g` in the body of `h`'s definition is of type $a \rightarrow a$. This is sufficiently specific for advising `g`, since `n3` is the only candidate. Consequently, the call to `f` inside the body of `n3` is also of type $a \rightarrow a$. However, this type is not sufficiently specific for advising `f`. As a result, we have to give `h` an advised type and it is translated as follows.

```
let n1 = \arg.e1 in
let n2 = \arg.e2 in
let f x = x in
let n3 = \df.\arg.df arg in
let g x = x in
let h dg x = dg x in
h <g,{n3 <f,{n1,n2}>}> 1
```

`n3` is only chained in the main expression where the context is sufficiently specific for both the calls to `g` and `f`.

4.5 Higher-Order Advices

In our system, we show that, just like functions, advices can be advised liberally. An example is given below.

Example 5

```
n1@advice around {f} (arg::Int) = e1 in
n2@advice around {n1} (arg::Int) = e2 in
let f x = x in
let g x = f x in
g 1
```

The second advice declaration is higher-order as it advises another advice `n1`. The advising mechanism in our language does not prejudice functions over advices. The translation $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$ in the premise of Rule (VAR-A) not only translates bodies of advices but also chains n_i with advices defined on it.

In the premises of Rule (ADV) and (ADV-AN), we note that typing information of advices is not stored in Γ_{base}. Thus, we replace $f_i : \sigma' \in \Gamma_{base}$ by $n_i :_* \sigma' \in \Gamma$.[2] Consequently, the check $\sigma' \trianglelefteq [\![\sigma]\!]$ in (ADV) becomes $[\![\sigma']\!] \trianglelefteq [\![\sigma]\!]$ as σ' may be an advised type. By doing this, we assume advised advices are translated before the advices defined on them. This is valid because circular cases are precluded.

Thus, example 5 is translated into

```
let n1 = \arg.e1 in
let n2 = \arg.e2 in
let f x = x in
let g df x = df x in
g <f,{<n1,{n2}>}> 1
```

Note that advice `n1` is chained with `n2` before the chaining to `f`.

4.6 Correctness of Translation

One of the desirable properties of our type-directed weaving algorithm is its reliance on a type-inference system that is a conservative extension of the Hindley-Milner Type System. (Note that the notation $[\![\cdot]\!]$ is defined in Figure 2.)

Theorem 3 (Conservative Extension) *Given a program P consisting of a set of advices and a closed base program e. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash e : [\![\sigma]\!].$$

Our main theorem is to ensure that our translated program preserves the type of the original program. When the original program is of an advised type, the translated scheme will concretize the advice predicates into advice parameters, which constitute part of the translated program. To this end, we define a function η that translates advised type to normal polymorphic type.

$$\begin{array}{rcl} \eta(\forall \bar{a}.\rho) & = & \forall \bar{a}.\eta(\rho) \\ \eta((x:t).\rho) & = & t \rightarrow \eta(\rho) \\ \eta(t) & = & t \end{array}$$

This main theorem ensures that the type-directed weaving is type-safe.

Theorem 4 (Type Preservation) *Given a program P consisting of a set of advices and a closed base program. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash_i P' : \eta(\sigma).$$

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

5. Related Works and Discussions

Since the introduction of aspect-oriented paradigm [7], researchers have been developing its semantic foundations. Most of the works in this aspect were done in object-oriented context in which type inference, higher-order functions and parametric polymorphism are of little concern. Recently, researchers in functional languages have also started to study various issues of adding aspects to functional languages. Two notable works in this area, PolyAML [4] and Aspectual Caml [9], 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 aspects with parametric polymorphism and higher-order functions – two essential features of modern functional languages. Neither have they adequately addressed the issues of advising advices, which we have discussed in this paper.

PolyAML advocates first-class join points for constructing generic aspect libraries [4]. It allows programmers to define polymorphic advices using type-annotated pointcuts. Unfortunately, PolyAML in [4] does not support *around* advice. The authors are currently extending the language to remedy this [14, 3]. In order to support non-parametric polymorphic advice, PolyAML includes case-advices which are subsumed by our type-scoped advices. Its type system is a conservative extension to Hindley-Milner type inference algorithm with a form of local type inference based on the required annotation on pointcuts. A type-preserving translation inserts labels which serve as marks of control-flow points. During execution, advices are looked-up through the labels and runtime type analysis are performed to handle the matching of type-scoped pointcuts, through which *execution* pointcuts with higher-order functions are supported. It is worth mention that this translation has little resemblance to ours as it does not strive to make weaving decisions at static time. Lastly, advices are anonymous in PolyAML and apparently not intended to be the targets of advising, *aka.* no higher-order advices.

Aspectual Caml [9], on the other hand, does not require annotations on pointcuts. It gives pointcuts the most general types available in context and ensures that the types of the advices hinged on the pointcut are consistent with the type of the pointcut. Similar to PolyAML, it also allows a restricted form of type-scoped advices. Yet, unlike our approach, the types of the functions specified in a pointcut are not checked against the type of the pointcut during type inference. Type safety of advice application is considered later in the weaving process. After type inference, its weaver goes through all type-annotated functions to insert advice calls. For each expression, it looks for advice definitions which have pointcuts that match this expression. If the type of the pointcut is more general than the type of the matched expression, the expression will be replaced by an application to the advice function. This syntactic approach makes it easy to advise anonymous functions. However, for polymorphic functions invoked indirectly through aliases or functional arguments, this approach cannot achieve coherent weaving results. It is also not clear how to extend the syntactic weaving scheme to handle nested advices or higher-order advices.

The current work is a conservative extension of our previous work [13], where we developed a type-directed weaving strategy for functional languages featuring higher-order functions, curried pointcuts and overlapping type-scoped advices. *Around* advices are woven into the base program based on the underlying type context using a Hindley-Milner type inference system extended with advised types and source translation. Coherent translations are

achieved without using any dynamic typing mechanisms. However, in that work, advices and functions are still kept in two completely different levels: advices can neither invoke advised functions nor advise other advices. It was also not clear how to weave advice into polymorphic recursive functions properly. All these shortcomings are fully addressed in this paper by re-designing our type inference system and translation scheme.

In contrast to AspectJ's direct translation into a non-aspect-oriented language, our targeted intermediate language requires addition of chaining expressions. This has been designed for the purpose of presentation clarity. There are many well known schemes such as inlining and closure [2] which can be directly applied to translate the intermediate language into a main stream non-aspect-oriented language. For the purpose of this paper, we omit discussions on this aspect as the added complexity does not contribute any further insights into the static and coherent weaving problem addressed here. Another advantage of our intermediate language is that it supports incremental weaving. Note that a chaining expression $\langle f\ , \{\bar{e}\}\rangle$ has the same static semantics as f. Therefore, it is straightforward to extend our current system to incorporate chaining expressions of the form $\langle f\ , \{\bar{e}\}\rangle$ as the targets for chaining any future advices defined on *f*.

Our type-directed translation was originally inspired by the dictionary translation of Haskell type classes [12]. A number of subsequent applications of it [8, 6] also shares some similarities. However, the issues discussed in this paper are unique, which makes our translation substantially different from the others.

6. Conclusion

Static typing, static and coherent weaving are our main concerns in investigating how to incorporate the essential features of aspects into a core functional language with higher-order functions and parametric polymorphism. As a sequel to our previous results, this paper has advanced our investigation in a variety of ways. Firstly, the target language of our translational semantics of advice weaving has been refined and given a neater formalization. Secondly, we have devised new typing and translation rules to handle the weaving of advices on polymorphic recursive functions. Thirdly, while the basic structure of our type system remains the same, the typing rules have been significantly refined and extended beyond the two-layered model of functions and advices. Consequently, advices can also be advised, either directly or indirectly. All these are accomplished by fully exploring the type information available in context and a novel technique of threading the types of matching advice chains; it is truly a type-directed weaving.

Moving ahead, we shall continue this line of investigation in a few directions. Currently the operational semantics of the intermediate language is purely reduction-based and hence we need to perform α-conversions to avoid name clashes. We plan to look into a closure-based semantics for the intermediate language that should be free of such intricacies. At the aspect language side, some extensions of the pointcuts are worth further investigation. Specifically, we shall consider how to support the control-related *Cflow* pointcuts available in many Java-based aspect-oriented languages. Finally, a prototype implementation is surely a necessary means for us to explore potential applications of our type-scoped advices [13].

References

[1] Aspectwerkz project. http://aspectwerkz.codehaus.org.

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

[3] Daniel S. Dantas, January 2006. personal communication.

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

[5] Jboss aop project. http://www.jboss.org/products/aop.

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

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

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

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

[10] Harold Ossher and Peri Tarr. Aspectwerkz projectmulti-dimensional separation of concerns in hyperspace, 1999. IBM research report.

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

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

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

[14] Geoffrey Washburn, February 2006. personal communication.

A. Sample Derivations

In this section, we present the typing/translation derivation of the examples given in the paper. We use I as a short hand for Int to save space. Some obvious details are also omitted.

A.1 Example 1

The derivation of the definition of f is:

$$\Gamma = \{h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,\\ n_1 : I \to I \bowtie h\}$$

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

$$\dfrac{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)}{\Gamma \vdash \lambda x.(h\ x) : (h : t \to t).t \to t \rightsquigarrow \lambda dh.\lambda x.(dh\ x)}\ (\textsc{Pred})$$

The derivation of the main expression is:

$$\Gamma_3 = \{h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,$$

$$n_1 : I \to I \bowtie h, f : \forall a.(h : a \to a).a \to a \rightsquigarrow f\}$$

$$\dfrac{\dfrac{\dfrac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : I \to I).I \to I \rightsquigarrow f}\ (\text{VAR}) \quad @ \quad \ldots}{\Gamma_3 \vdash f : I \to I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle)}\ (\text{REL})}{\Gamma_3 \vdash (f\ 1) : I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \dfrac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad \ldots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow \langle h\ , \{n_1, n_2\}\rangle}\ (\text{VAR-A})$$

A.2 Example 2

The derivation of the definition of f is:

$$\Gamma = \{g : I \to I \rightsquigarrow g, head : \forall a.[a] \to a \rightsquigarrow head,\ tail : \forall a.[a] \to [a] \rightsquigarrow tail\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{f : [a] \to [a] \rightsquigarrow df \in \Gamma_2}{\Gamma_2 \vdash f : [a] \to [a] \rightsquigarrow df}\ (\text{VAR}) \quad \ldots}{\Gamma_2 = \Gamma_1, x : [a] \vdash f\ (tail\ x) : [a] \rightsquigarrow df\ (tail\ x)}\ (\text{APP})}{\Gamma_1 = \Gamma, f : [a] \to [a] \rightsquigarrow df \vdash \lambda x....then\ f\ (tail\ x)... : [a] \to [a] \rightsquigarrow \lambda x....then\ df\ (tail\ x)...}\ (*)}{\Gamma \vdash \lambda x....then\ f\ (tail\ x)... : (f : [a] \to [a]).[a] \to [a] \rightsquigarrow \lambda df.\lambda x....then\ df\ (tail\ x)...}\ (\text{PRED})$$

The derivation of the main expression is:

$$\Gamma_3 = \{g : I \to I \rightsquigarrow g, head : \forall a.[a] \to a \rightsquigarrow head,\ tail : \forall a.[a] \to [a] \rightsquigarrow tail, n : I \to I \bowtie f,\ f :_* \forall a.(f : [a] \to [a]).[a] \to [a] \rightsquigarrow f\}$$

$$\dfrac{\dfrac{\dfrac{f :_* \forall a.(h : [a] \to [a]).[a] \to [a] \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (f : [I] \to [I]).[I] \to [I] \rightsquigarrow \lambda y.\langle f\ y\ , \{n\}\rangle}\ (\text{VAR-A}) \quad \ldots}{\Gamma_3 \vdash f : [I] \to [I] \rightsquigarrow \mathsf{let}\ F = \lambda y.\langle f\ y\ , \{n\}\rangle\ F}\ (\text{REL-F})}{\Gamma_3 \vdash (f\ [1,2,3]) : [I] \rightsquigarrow (\mathsf{let}\ F = \lambda y.\langle f\ y\ , \{n\}\rangle\ F)\ [1,2,3]}\ (\text{APP})$$

A.3 Example 4

The derivation of the definition of n_3 is:

$$\Gamma = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 : I \to I \bowtie f,\ n_2 : \forall a.a \to a \bowtie f\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : a \to a \rightsquigarrow df \in \Gamma_2}{\Gamma_2 \vdash f : a \to a \rightsquigarrow df}\ (\text{VAR}) \quad \ldots}{\Gamma_2 = \Gamma_1, arg : a \rightsquigarrow arg \vdash f\ arg : a \rightsquigarrow df\ arg}\ (\text{APP})}{\Gamma_1 = \Gamma, f : a \to a \rightsquigarrow df \vdash \lambda\ arg.f\ arg : a \to a \rightsquigarrow \lambda\ arg.df\ arg}\ (\text{ABS})}{\Gamma_1 = \Gamma, prd : a \to a \vdash \lambda\ arg.f\ arg : (f : a \to a).a \to a \rightsquigarrow \mathsf{let}\ n = \lambda\ df.\lambda\ arg.df\ arg}\ (\text{PRED})}{\Gamma \vdash n3@advice\ around\ g\ (arg) = f\ arg\ \mathsf{in}\ \ldots : \ldots \rightsquigarrow \mathsf{let}\ n = \lambda\ df.\lambda\ arg.df\ arg\ \mathsf{in}\ldots}\ (\text{ADV})$$

Similarly, h is inferred to have type $(g : a \to a).a \to a$. The reason for this advised type is that n_3 fails to be chained with the g-call in that context as the sub-derivation $\Gamma \vdash n_3 : a \to a$ in (VAR-A) fails.

The derivation of the main expression is:

$$\Gamma_3 = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 : I \to I \bowtie f,\ n_2 : \forall a.a \to a \bowtie f, n_3 : \forall a.(f : a \to a).a \to a \bowtie g,\ g :_* \forall a.a \to a \rightsquigarrow g, h :_* \forall a.(g : a \to a).a \to a \rightsquigarrow h\}$$

$$\dfrac{\dfrac{\dfrac{h : \forall a.(g : a \to a).a \to a \rightsquigarrow h \in \Gamma_3}{\Gamma_3 \vdash h : (g : I \to I).I \to I \rightsquigarrow h}\ (\text{VAR}) \quad @ \quad \ldots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow (h\ \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle)}\ (\text{REL})}{\Gamma_3 \vdash (h\ 1) : I \rightsquigarrow (h\ \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \dfrac{\dfrac{\dfrac{\ldots}{\Gamma_3 \vdash n_3 : (f : I \to I).I \to I \rightsquigarrow n_3}\ (\text{VAR}) \quad \dfrac{\ldots}{\Gamma_3 \vdash f : I \to I \rightsquigarrow \langle f\ , \{n_1, n_2\}\rangle}\ (\text{VAR-A})}{\Gamma_3 \vdash n_3 : I \to I \rightsquigarrow n_3\ \langle f\ , \{n_1, n_2\}\rangle}\ (\text{REL})}{\Gamma_3 \vdash g : I \to I \rightsquigarrow \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle}\ (\text{VAR-A})$$

A.4 Example 5

The derivation of the main expression is:

$$\Gamma = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 :_* I \to I \bowtie f,\ n_2 : I \to I \bowtie n_1, g : \forall a.(f : a \to a).a \to a \rightsquigarrow g\}$$

$$\dfrac{\dfrac{\dfrac{g : \forall a.(h : a \to a).a \to a \rightsquigarrow g \in \Gamma}{\Gamma \vdash g : (f : I \to I).I \to I \rightsquigarrow g}\ (\text{VAR}) \quad @ \quad \ldots}{\Gamma \vdash g : I \to I \rightsquigarrow g\ \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle}\ (\text{REL})}{\Gamma \vdash (g\ 1) : I \rightsquigarrow (g\ \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \dfrac{\dfrac{n_1 :_* I \to I \rightsquigarrow n_1 \in \Gamma \quad \ldots}{\Gamma \vdash n_1 : I \to I \rightsquigarrow \langle n1\ , \{n2\}\rangle}\ (\text{VAR-A}) \quad \ldots}{\Gamma \vdash f : I \to I \rightsquigarrow \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle}\ (\text{VAR-A})$$