Type-Directed Weaving of Aspects for Higher-order Functional Languages

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 a type inference mechanism, polymorphic types, higher-order functions and *type-scoped* pointcuts. Our main contribution lies in a seamless integration of these two paradigms through a static weaving process which deals with *around* advices with type-scoped pointcuts in the presence of higher-order functions. We give a source-level type inference system for a higher-order, polymorphic language coupled with type-scoped pointcuts. The type system ensures that base programs are oblivious to the type of around advices. We present a type-directed translation scheme which resolves all advice applications at static time. The translation removes advice declarations from source programs and produces translated code which is typable in the Hindley-Milner system.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism,functions,Control structures; D.3.2 [*Language Classifications*]: Applicative (functional) languages; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

General Terms Languages,Theory

Keywords Aspect Oriented, Higher-Order, Type Inference, Weaving, Functional Language

1. Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system [13]. 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.

Since its inception, AOP has been closely investigated mainly in the contexts of object-oriented programming languages such as Java [12, 10, 2] and C++ [17]. Recently, researchers in functional languages have also started to study various issues of adding aspects to functional languages, ranging from foundational calculi [3, 9, 20] to prototype implementations [5, 16]. Two notable works in this area, PolyAML [5] and Aspectual Caml [16], have made many significant results on supporting polymorphic pointcuts and advices in strongly typed functional languages such as ML. Each of these works has its own approach to preserving type safety under the Hindley-Milner style type system.

While these works have been very illuminating for understanding the problems involved, they have not fully explored some of the subtle issues which are crucial for a full-fledged support of aspects in higher-order functional languages. For examples, PolyAML does not support *around advices* and calls for dynamic type checking to handle matching of type-scoped pointcuts. The syntax-directed weaving strategy of Aspectual Caml cannot handle higher-order functions properly. Essentially, strong-typing property of functional languages, in the presence of parametric polymorphism and higher-order functions, may be compromised when aspect weaving may be constrained by types. Indeed, it is a challenging task to properly reconcile all these features under a coherent framework.

To demonstrate the complexity of the matter, consider the following aspect-oriented functional program:

Example 1

```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg::[a])
    = proceed (tail arg) in
n3@advice around {h} arg = proceed arg in
n4@advice around {f} (arg) = e1 in
n5@advice around any (arg::Int) = e2 in
n6@advice around {f x} (arg::Int) = e3
in let h x = x in
   let f g y = g (h y) in
   (f h 1, f h [2.5], h)
```

This piece of code defines six advices named from `n1` to `n6`; it also defines a main program consisting of declarations of `f` and `h` and a main expression returning a triplet. The first two advices, `n1` and `n2`, designate calls to `h` as pointcuts. They differ in the type constraints, which are called *type scopes*, of their first argument. `n1` is only triggered when `h` is called with an Int argument, whereas `n2` is triggered when `h` is called with a list as argument. Due to the

PEPM '06 January 9–10, Charleston, South Carolina, USA.

differing type scopes, at most one of these advices will be triggered at a call to `h`. However, the only physical application of `h` in the main program appears in the RHS of `f`'s definition, and under a polymorphic type context. It will be an error to allow either `n1` or `n2` to intercept the call to `h` at this point. Instead, the actual calling context of `h` will only be revealed in the first two components of the triplet expression, where `h` also appears as higher-order argument of the two calls to `f`. In order to correctly chain the advice `n1` to the occurrence of `h` in `((f h) 1)`, and `n2` to that in `((f h) [2.5])`, we need a type inference system that deals with advice declarations in the presence of higher-order functions.

The third advice `n3` also designates calls to `h` but without any type constraint on `h`'s argument. Certainly, such advice should crosscut both the indirect calls of `h` via the two invocations of `f`; in addition, it should also crosscut any future call to the third component of the triplet, the isolated `h` function. This demonstrates the intricacy of defining weaving decision for a variable, such as `h`.

The fourth advice `n4` designates calls to `f` with no specified type scope as its pointcut. The fifth one `n5` uses an *any* pointcut which suggests that it be triggered at any function calls – including partial applications – with Int argument. The last one has a curried pointcuts. It matches the second partial application of `f` to an argument of type Int. All these variants of pointcuts have to be managed systematically for advices to be triggered at appropriate points.

In this paper, we present a novel approach to type-safe weaving of aspects for higher-order functional languages. Inspired by the predicated types of type classes [19], we introduce the notion of *advised types* to guard against unsafe weaving. The central idea 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, we give 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 which are typable in Hindley-Milner system. We show that our type inference system is a conservative extension of the Hindley-Milner system, and the translated program remains well-typed.

For the above example code, our translation produces the following code:

```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed (tail arg) in
let n3 = \arg -> proceed arg in
let n4 = \arg -> e1 in
let n5 = \arg -> e2 in
let n6 = \x.\arg -> let proceed = proceed x in e3 in
let h x = x in
let f dh g y = g (dh y) in
(<<f,{n4,n6}> <h,{n1,n3,n5}> <h,{n1,n3,n5}>,{n5}> 1,
 <f,{n4}> <h,{n2,n3}> <h,{n2,n3}> [2.5],
 <h,{n3}>)
```

Note that all advice declarations are translated into functions and are woven in. We use a combinator $\langle \_ \ , \{\ldots\}\rangle$ to chain advices and advised functions. For instance, $\langle$`f` , `{n4,n6}`$\rangle$ denotes the chaining of advices `n4` and `n6` to advised function `f`. It is important to note that the direct call to `h` in `f`'s definition above is not advised. Instead, it is abstracted into an advice parameter waiting to be instantiated. The purpose of this postponed decision is to ensure a *coherent weaving*. In the main program, those calls to `f` and `h` in the first two components of the triplet will trigger different sets of advices, as determined by their argument types. In the first component, before receiving the translated counterpart of `h` as first argument, the translated `f` is advised by `n4` and `n6`[1], and is supplied with the translated `h` to instantiate its advice parameter. Then, the partially applied `f` is advised by `n5` before applying to the Int argument. Note that in this case, both the indirect and direct calls of `h` trigger the same set of advices. But this may not be the case in general. The second component of the triplet, in contrast, is woven in some other set of advices, as the the actual argument fed to `f` is of type $[Float]$, and the curried pointcut does not match. Note that here and throughout the paper, we assume advices are triggered according to the textual order in which they are declared.

Our approach mimics the translation scheme used in handling type classes [19]. However, there are significant differences between the two. This is especially noticeably in the handling of overlapping type-scoped advices to ensure a coherent translation. We shall explain this in Section 4.

The main contributions of this paper are as follows.

- We formally specify a type inference system for a strongly-typed aspect-oriented functional language featuring polymorphism, higher-order functions and *around* advices with type-scoped pointcuts. Our type system ensures that well-typed programs are *oblivious* to the types of their around advices.
- We propose a novel technique for *type-directed weaving* of aspects. This is accomplished by means of a type-directed translation scheme which implements weaving statically. The translated program is typeable in the Hindley-Milner system. The technique performs weaving in the presence of higher-order functions and type-scoped curried pointcuts; it also ensures coherent advices are given to functions in the presence of overlapping type-scoped advices.

The outline of the paper is as follows: Section 2 acquaints the reader with the essence of aspect-oriented programs, and sets the background for our discussion. Section 3 outlines individual challenges faced in static weaving of strongly-typed higher-order polymorphic functional language. Our main technical contribution is described in Section 4, and its further extension is discussed in Section 5. We discuss related works in Section 6 before concluding in Section 7.

2. Preliminaries

An aspect-oriented program is generally divided into two parts: A *base* program and a number of *aspects*. The aspects can be viewed as observers of the base program, taking actions whenever certain events occur in the latter's execution. Actions taken by aspects are called *crosscutting*. A piece of aspect code that describes an intervening action is called an *advice*. In functional language setting, aspect code will usually be treated like global declarations in a program. We shall use the following syntax and conventions for describing our work:

Expressions	e	$::=$	$c \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ f\ =\ e\ \mathsf{in}\ e$
			$n@\mathsf{advice\ around}\ pc = e\ \mathsf{in}\ e$
Arguments	arg	$::=$	$x \mid x :: t$
Pointcuts	pc	$::=$	$\{\overline{f}\ \bar{x}\}\ (arg) \mid any\ (arg)$
Types	t	$::=$	$a \mid t \to t \mid T\ \bar{t}$
Type Schemes	σ	$::=$	$\forall \bar{a}.\rho$
Advised Types	ρ	$::=$	$(x : t).\rho \mid t$

We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, \ldots, o_n$ (e.g. expressions, types etc). 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

[1] Astute readers may notice that `n6` is chained with `f` instead of the partial application of it. The reason for this is explained in 4.1.3.

(*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, recursive function definitions and patterns but may make use of them in examples. We assume that c refers to constructors of user-defined data types $T\ \bar{a}$. Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment Γ_{init}.

In our language, an aspect is an advice declaration which includes a piece of advice and its target $pointcut$. Pointcuts are represented by $\{\overline{f\ \bar{x}}\}\ (arg)$. An advice will be triggered when a call is made to any function from the set of pointcuts associated with this advice.. We say a function is being advised when it appears in the pointcut of an aspect. The syntax here allows curried pointcuts which only match partially applied functions to their arguments $\bar{x}$. The argument variable arg will be bound to the actual argument of the function call and it may contain an annotated type. Only function calls with arguments of types not more general than the type scope is matched by the pointcut. When the type scope is absent, all calls to the functions from the set is matched. This design is very different from the polymorphic approach in [16] where type inference result of advices will affect the matching of the targeted pointcuts.

Advice is a function-like expression that executes *before*, *after*, or *around* a pointcut. Note that *around* advice is executed 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, in this paper, only *around* advice is considered. Furthermore, since an *around* advice may replace its advised function, its type must be unifiable with the type of the advised function. This is guaranteed by our type inference system. Hence, in this sense, programs in our system are *oblivious* [7] to the types of their around advices and type safety is preserved for advice execution. To capture all functions in a pointcut, we use *any* as a call to any function.

The intervening action of aspects is achieved by a *weaving* process. A weaver either dynamically or statically searches for matching pointcut of a function call; and *chains* the function with corresponding advices. In the syntax, we also introduce the notion of *advised types* to facilitate our translation. It will be formally introduced in Section 4. Reader can safely ignore it for the time being.

3. Challenges in Static Weaving

In a strongly-typed language, it is always desirable to be able to perform type erasure to programs without incurring additional runtime error during program execution. From the aspect-oriented perspective, it is therefore imperative to question if pointcuts constrained by types will affect the strong-typing property of the language.

Specifically, we allow 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*.

Consider the following code declaring two type-scoped advices:

Example 2
```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg::[a]) = proceed (tail arg)
in let h x = x in
   (h 1,h [2.5])
```

Both `n1` and `n2` are examples of type-scoped advices. Different advices will be required for different applications of the function `h`. As such, it is no longer sufficient to determine the applicability of an advice by matching the function names in the pointcut alone. Instead, type inference is required to decide on the choice of advices. In the above case, `h` should be advised by `n1`, but not `n2`, during the application of `(h 1)`. Similarly, it should be advised by `n2`, but not `n1`, during the `(h [2.5])` application.

Thus, the introduction of type-scoped advices complicates static weaving of aspects, particularly in the case of advising polymorphic functions.

Another important property of aspect-oriented languages which may be impacted by type-scoped advices is *obliviousness* [7]. Oblivious means programmers can add advices to a program "after-the-fact" in the typical aspect-oriented style. In the context of typing, the type inference of advices should not affect the typing of the original program. Consider this example.

Example 3
```
n1@advice around {h} (arg) = proceed (arg+1) in
n2@advice around {h} (arg::Int) = proceed (arg +1)
in let h x = x in
   (h True)
```

The advice `n1` is potentially unsafe. Note that the definition of `h` is of type $\forall a.a \rightarrow a$. In the base program without advices, it is called with argument `True`. This application, if advised by `n1`, will result in erroneous computation of `True + 1`. A correct type inference of AOP should not restrict the type of `h` to suit the advices. Instead, advice `n1` should be rightfully rejected. On the contrary, the second advice `n2` is safe since it is only triggered by calls with Int arguments.

3.1 Coherent Advices

Whereas advising on polymorphic functions can be intricate, the presence of *polymorphic type-scoped* advices brings forth even greater challenge to the determination of a static weaving. This is because different advices to a pointcut may have *overlapping* type scopes.

Consider the following advices:

Example 4
```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg) = proceed (arg) in
```

Here, `n2` is not constrained by type, and we can infer that it has the type $\forall a.a \rightarrow a$. Consequently, the type scopes 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. For example, consider the following main program for the above advices:

```
let h x = x in
let f x = h x in
(f 1)
```

From syntactic viewpoint, function `h` will be called in the body of `f`. Since 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 `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`. Apparently, this syntactic approach to static weaving is adopted by Aspectual Caml [16].

Unfortunately, this approach will cause incoherent behaviour of `h` at run-time. Specifically, in the subsequent context, the main expression (`f 1`) will actually pass integer argument `1` to `h`. There, advice from `n1` will be missed out since the weaver has mistakenly committed its choice (of chaining only `n2`) in the application of `h`. The only coherent behaviour of a weaver in this case is to have `h` being advised by both `n1` and `n2`.

3.2 Higher-Order Functions

The presence of higher-order functions further complicates the problem of static weaving because the physical separation of functions and their calls make it impossible to statically allocate advices to a call in a local context. Nevertheless, as we shall illustrate in Section 4, it is still possible to translate the original program into one in which while advice-wrapped functions are passed as arguments dynamically, dispatching of advices has been totally compiled away.

Consider a variant of Example 2.

Example 5

```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg::[a]) = proceed (tail arg)
in let h x = x in
   let f g y = g y in
   (f h 1,f h [2.5])
```

In the main program, there are two indirect calls to `h` which, due to higher-orderness, are not explicitly shown in the program text. A correct static weaving entails making decision about the appropriate advices for each call to `h` at each of the corresponding application of `f`.

3.3 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 6

```
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
```

Note that because of well-typedness, we insist that the advice body of `n2`, `e2`, to be of a function type $\cdot \to \cdot$, while the advice bodies `e1` and `e3` are of non-function types.

There is not yet any unambiguous agreement on the operational semantics of aspect-oriented programs for higher-order functions and partial function applications, although one such recommendation has been presented lately [18]. 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 [16]. This differs from the recommendation by [18] which triggers advices based on matching of run-time closures.

Back to the above example, we stipulates that advice `n2` will be triggered at the application (`f 1`). Furthermore, advice `n1` and `n3` will be triggered when the application of (`f 1`) to 2 is encountered at run-time.

4. Type Directed Weaving

Inspired by the concept of *predicated type* [19] used in type classes, we introduce *advised type* denoted as ρ to capture function names and their types that may be required for advice resolution. For instance, in the main program given in Example 4, 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 less 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 4 is $\forall a.a \to a$ since it does not contain any applications of advised functions in its definition.

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

$$(\text{GEN}) \qquad \frac{[\bar{t}/\bar{a}]t_1 \sim t_2}{\forall\bar{a}.t_1 \trianglerighteq \forall\bar{b}.t_2}$$

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

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

Figure 1. Auxiliary Definitions

Figure 1 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 (GEN) defines a relation $\trianglerighteq$ between two type schemes using a unification relation $\sim$. Specifically, $\sigma_1 \trianglerighteq \sigma_2$ if and only if σ_1 is more general than σ_2, and can be instantiated to a more specific type σ_2 via variable substitution. Rule (AERASE) defines a function $[\![\cdot]\!]$ which removes all advice predicates from an advised type scheme. We also define, in rule (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 (CARD_p) function returns the number of advice predicates in a type scheme.

The main set of type inference rules, as described in Figure 2, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash_{\leadsto} e : \sigma \leadsto e'$ to denote that expression e has type σ under type environment Γ and it is translated to e'. We write $\Gamma \vdash e : \sigma$ to denote a Hindley-Milner inference of a program without advices where we do not make use of rules (VAR-A), (LET-A), (ADV), (ADV-AN), (PRED) and (REL). We assume the base program without advice declaration is type correct; and the type information of all the functions are stored in an initial environment Γ_{base}.

The type environment Γ contains not only the usual type bindings (of the form $x : \sigma$) but also *advice bindings* of the form $n :_a \sigma \bowtie x$. This states that an advice with name n is defined on x. For the sake of convenience, we call σ the type of the advice even though advices are not first class citizens in our system. This type σ is inferred from the body and the type scope of the advice described in rules (ADV) and (ADV-AN); and it is used to guard advice application in rule (VAR-A). We also introduce *advised-function bindings* of the form $x :_* \sigma$ to bind advised functions to their inferred types. Such a binding is introduced by rule (LET-A).

We store all advised functions names in a global store A by extracting them from all the pointcuts. In the presence of an *any* advice, all named functions are included in A. Note that while it is possible to present the typing rules without the translation

$$
(\textsc{Var})\quad \frac{x : \sigma \leadsto e \in \Gamma}{\Gamma \vdash_{\leadsto} x : \sigma \leadsto e}
\qquad
(\textsc{Var-A})\quad \frac{\begin{array}{c} x :_* \sigma' \in \Gamma \quad \bar{\sigma} \ntrianglelefteq [\![\sigma']\!] \\ \bar{n} :_a \overline{\sigma} \bowtie (x \vee any) \in \Gamma \quad \{n_i \mid \sigma_i \trianglerighteq [\![\sigma']\!]\} \\ |\bar{y}| = |\sigma'|_{pred} \quad \bar{y}\ is\ fresh \end{array}}{\Gamma \vdash_{\leadsto} x : \sigma' \leadsto \lambda \bar{y}.\langle x\ \bar{y}\ , \{n_i\}\rangle}
$$

$$
(\forall\textsc{Elim})\quad \frac{\Gamma \vdash_{\leadsto} e : \forall a.\sigma \leadsto e'}{\Gamma \vdash_{\leadsto} e : [t/a]\sigma \leadsto e'}
\qquad
(\forall\textsc{Intro})\quad \frac{\Gamma \vdash_{\leadsto} e : \sigma \leadsto e' \quad a \notin \Gamma}{\Gamma \vdash_{\leadsto} e : \forall a.\sigma \leadsto e'}
$$

$$
(\textsc{Abs})\quad \frac{\Gamma, x : t_1 \leadsto x \vdash_{\leadsto} e : t_2 \leadsto e'}{\Gamma \vdash_{\leadsto} \lambda x.e : t_1 \rightarrow t_2 \leadsto \lambda x.e'}
\qquad
(\textsc{Let})\quad \frac{\begin{array}{c} \Gamma \vdash_{\leadsto} e_1 : \sigma \leadsto e_1' \quad f \notin A \\ \Gamma, f : \sigma \leadsto f \vdash_{\leadsto} e_2 : t \leadsto e_2' \end{array}}{\Gamma \vdash_{\leadsto} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \leadsto \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}
$$

$$
(\textsc{App})\quad \frac{\begin{array}{c} \Gamma \vdash_{\leadsto} e_1 : t_1 \rightarrow t_2 \leadsto e_1' \\ \Gamma \vdash_{\leadsto} e_2 : t_1 \leadsto e_2' \end{array}}{\Gamma \vdash_{\leadsto} e_1\ e_2 : t_2 \leadsto (e_1'\ e_2')}
\qquad
(\textsc{Let-A})\quad \frac{\begin{array}{c} \Gamma \vdash_{\leadsto} e_1 : \sigma \leadsto e_1' \quad f \in A \\ \Gamma, f :_* \sigma \leadsto f \vdash_{\leadsto} e_2 : t \leadsto e_2' \end{array}}{\Gamma \vdash_{\leadsto} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \leadsto \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}
$$

$$
(\textsc{Pred})\quad \frac{\Gamma, x : t \leadsto x_t \vdash_{\leadsto} e : \rho \leadsto e_t' \quad x \in A}{\Gamma \vdash_{\leadsto} e : (x : t).\rho \leadsto \lambda x_t.e_t'}
\qquad
(\textsc{Rel})\quad \frac{\begin{array}{c} \Gamma \vdash_{\leadsto} e : (x : t).\rho \leadsto e' \\ \Gamma \vdash_{\leadsto} x : t \leadsto e'' \quad x \in A \end{array}}{\Gamma \vdash_{\leadsto} e : \rho \leadsto e'\ e''}
$$

$$
(\textsc{Adv})\quad \frac{\begin{array}{c} \Gamma_{base}, proceed : t_1 \rightarrow t, x : t_1 \vdash e_a : t \quad \Gamma_{base} \vdash f_i : \sigma' \\ \sigma' \trianglelefteq t_1 \rightarrow t \quad \Gamma, n :_a \sigma \bowtie \bar{f} \vdash_{\leadsto} e : t' \leadsto e' \quad \sigma = gen(\Gamma, t_1 \rightarrow t) \end{array}}{\Gamma \vdash_{\leadsto} n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x) = e_a \mathsf{in}\ e : t' \leadsto \mathsf{let}\ n = \lambda x.e_a\ \mathsf{in}\ e'}
$$

$$
(\textsc{Adv-an})\quad \frac{\begin{array}{c} \Gamma_{base}, proceed : t_x \rightarrow t, x : t_x \vdash e_a : t \quad \Gamma_{base} \vdash f_i : t_i \rightarrow t_i' \\ t_x \trianglelefteq t_i \quad (t_i \rightarrow t_i') \sim (t_x \rightarrow t) \quad \Gamma, n :_a \sigma \bowtie \bar{f} \vdash_{\leadsto} e : t' \leadsto e' \quad \sigma = gen(\Gamma, t_x \rightarrow t) \end{array}}{\Gamma \vdash_{\leadsto} n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: t_x) = e_a \mathsf{in}\ e : t' \leadsto \mathsf{let}\ n = \lambda x.e_a\ \mathsf{in}\ e'}
$$

Figure 2. Type-directed Weaving by translation

detail by simply deleting the '$\leadsto e$' portion, it is not possible to present the translation rules independently since typing controls the translation.

There are two rules for variable lookups. Rule (VAR) is standard. In the case that the variable x is advised, rule (VAR-A) will check all advices defined on x 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 by the guard ($\bar{\sigma} \ntrianglelefteq [\![\sigma']\!]$). If the check fails (*i.e.*, no advice has a more specific type than x), x will be chained with all those advices defined on it, including those with pointcut any, having the same or more general types than x has. Note that the final translated expression is *normalized* by bringing all the advice abstractions of x outside $\langle\rangle$. This is to ensure type compatibility between an advised call and its advices. Section 4.3 gives more detailed discussion on this issue.

On the other hand, if there exists an advice for x with more specific type, then $x \in A$ must hold, and rule (PRED) will be applied. This rule introduces an *advice parameter* to the program (through the corresponding translation scheme). This advice parameter enables concrete *advise-chained functions* to be passed in at a later stage through application of rule (REL).

Rules (ABS),(LET), (APP), ($\forall$INTRO) and ($\forall$ELIM) are standard. Similar to (LET), rule (LET-A) binds advised functions.

Rules (PRED) and (REL) respectively introduces and eliminates advice predicates just as ($\forall$INTRO) and ($\forall$ELIM) do to bound type variables. Rule (PRED) adds an advice predicate to a type. Correspondingly, its translation yields a lambda abstraction with an advice parameter. On the other hand, rule (REL) removes an advice predicate from a type. Its translation generates a function application with an advised expression as argument.

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 first infer the type of e_a under the initial type environment extended with $proceed$ and variable x. The use of judgment $\vdash$ here indicates that we do not translate the body of advices. Thus, function calls in the body of advices should not be advised. We will come back to the issue of having advised function calls occurring within an advice's body in Section 5. After type inference of advice body, we ensure that all functions in the pointcut have type schemes that are not more general than $t_1 \rightarrow t$. 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.

Finally, the rules for advices with pointcut any is not shown. Nevetheless, it is essentially the same as (ADV) and (ADV-AN) respectively except that we replace the type judgement of advised functions by $\Gamma \vdash any : \forall ab.a \rightarrow b$.

4.1 Challenges Revisited

We revisit the challenges that were laid down in Section 3, and examine how the type inference system defined in Figure 2 ensures that proper advices are chained at each function calls, thus achieving type-directed weaving.

Let's begin by examining how type-scoped advices defined in Example 2 are being handled. The example is reproduced below:

```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg::[a]) = proceed (tail arg)
in let h x = x in
   (h 1,h [2.5])
```

Since `h` is an advised function, its type is kept in the type environment as an advised-function binding through application of rule (LET-A). Consequently, at each application of `h`, (VAR-A) is applied to chain advices to `h`. Only advices that ensure type obliviousness will be extracted for chaining, resulting in each call to `h` having a distinct set of advices being triggered.

The example is translated to

```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed (tail arg)
in let h x = x in
   (<h,{n1}> 1, <h,{n2}> [2.5])
```

In Example 3, the unsafe advice `n1` is correctly rejected by our type system (and no code is produced) since the inferred type of the advice which is $Int \rightarrow Int$ fails to be at least as general as `h`'s type $\forall a.a \rightarrow a$.

4.1.1 Coherent Advices

When type-scoped advices overlaps, our inference rules ensure that advices triggered are coherent to the expected run-time behaviour. Example 4 is reproduced below:

```
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)
```

As mentioned in Section 3.1, in order to provide coherent advices, the call `(h x)` should not be chained with concrete advices during the examination of `f`'s definition. This assurance is enforced by the sufficiently specific context check.

Since `n1` is of type $Int \rightarrow Int$, which is more specific than `h`'s instantiated type $a \rightarrow a$ in the current context, rule (VAR-A) fails to apply. Instead, `f` is inferred to have an advised type $(h : a \rightarrow a).a \rightarrow a$ through the application of rule (PRED). This means that decision for chaining advices will be deferred, and `f` will carry an advice parameter in the translated code.

In the main expression `(f 1)`, `f`'s type is instantiated from the polymorphic type to $Int \rightarrow Int$. Rule (REL) is thus applied to release the advices. In the premise of (REL), we try to derive the judgment $\Gamma \vdash_{\leadsto} h : Int \rightarrow Int \leadsto e''$. In this case, the guard in rule (VAR-A) is satisfied and `h` is chained to `n1` and `n2`. Consequently, the chained expression is passed as an argument to `f` through the rule (REL). The translated code is displayed below (detailed typing/translation derivation of this program is shown in Appendix A):

```
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)
```

This approach of avoiding early commitment to specific advices can be too conservative, and may fail to commit to any concrete advices when necessary. Consider a slight variant of the above example,

```
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 undefined)
```

Here, we conveniently use the Haskell's unique expression `undefined` to represent a value of type a for the purpose of demonstration.[2] Our type-directed weaving will produce the following translated code:

```
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
   \dh -> f dh undefined
```

Note that in the main expression of the translated code, `(\dh -> f dh undefined)`, advice application is still being abstracted even though we ought to commit to `n2` here as there will not be any further instantiation.

To circumvent this problem, we propose that during the translation of the main expression, we employ a variant of rule (VAR-A) which drops the sufficiently specific context check (*i.e.*, the guard $\bar{\sigma} \not\trianglelefteq [\![\sigma']\!]$) from the premise. As a result, we obtain the following translated code:

```
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,{n2}> undefined
```

Note that only advice `n2` is used in the translated main expression, as that is the only applicable advice in the current type context.

4.1.2 Higher-Order Functions

We handle higher-order named advised functions by replacing them with corresponding *advise-chained functions*. This is warranted when the function occurs in a sufficiently specific type context, and it is described by the rule (VAR-A). Thus, in the following code:

```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg::[a]) = proceed (tail arg)
in let h x = x in
   let f g y = g y in
   (f h 1,f h [2.5])
```

Function `h` appears in two call contexts, one with integer input and the other with float-list input. Application of type-directed weaving rules, and particularly the rule (VAR-A) to these occurrences of `h`, results in the generation of the following code:

```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed (tail arg)
in let h x = x in
   let f g y = g y in
   (f <h,{n1}> 1,f <h,{n2}> [2.5])
```

[2] The call `(f undefined)` will execute properly under call-by-need semantics. However, it should be understood that this does not restrict our proposed soluton to only handle programs with such semantics.

4.1.3 Curried Pointcuts

In [16], 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.

$$\text{(ADV-C)}\quad \frac{\begin{array}{c}\Gamma_{base}, proceed : t \to t_y \to t_a, x : t \vdash e_a : t_a \\ \Gamma_{base} \vdash f_i : t_1 \to t_2 \to t_3 \quad (t_1 \to t_2 \to t_3) \sim (t \to t_y \to t_a) \\ t_y \trianglelefteq t_2 \quad \Gamma, n :_a \sigma \bowtie \bar{f} \vdash_{\leadsto} e : t' \leadsto e' \\ \sigma = gen(\Gamma, t \to t_y \to t_a)\end{array}}{\begin{array}{l}\Gamma \vdash_{\leadsto} n@\mathsf{advice\ around}\ \{\bar{f}\ x\}\ (y :: t_y) = e_a \mathsf{in}\ e : t' \\ \quad \leadsto \mathsf{let}\ n = \lambda x.\lambda y.\mathsf{let}\ proceed = proceed\ x\ \mathsf{in}\ e_a\ \mathsf{in}\ e'\end{array}}$$

Given the following code example:

```
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
```

Using this rule, our type-directed translation will produce the following code:

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

Note that `e2` is expected to be a function type as ensured by the typing rules. Thus, the types of the three functions `n1`, `n2` and `n3` are unifiable with `f`'s.

4.2 Advising Anonymous Functions

So far, advices are only chained with named functions at the time of name lookup, as described in rule (VAR-A). Rule (ANY), on the other hand, deals with anonymous functions and partial applications. This rule looks for non-variable expressions of function types and chains applicable *any* advice with them.

$$\text{(ANY)}\quad \frac{\begin{array}{c}\bar{n} :_a \bar{t} \bowtie any \in \Gamma \quad \{n_i \mid t_i \trianglerighteq t\} \quad \Gamma \vdash_{\leadsto} e : t \leadsto e' \\ e\ is\ not\ var \quad t \sim t_1 \to t_2 \quad t_1, t_2\ fresh\end{array}}{\Gamma \vdash_{\leadsto} e : t \leadsto \langle e', \{n_i\}\rangle}$$

Different from the rule (VAR-A), (ANY) does not check whether the current context is sufficiently specific because an unnamed expression cannot be subsequently instantiated.

4.3 Operational Semantics of Chaining

In the translated program, we chain all applicable advices with an advised function using a special syntax $\langle \_ , \{\ldots\}\rangle$. In the same spirit as function composition, this operator sequentially chains a sequence of functions that have unifiable types; and the advice-chained function produced can be given the same type as the original function's. However, the chaining participants may not always have unifiable types due to the abstraction of advice parameters. A normalization step is required to reconcile the compatibility of types. Consider this example:

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

The translation chains `f` with `n2` whose types are not unifiable since the translated `f` takes one extra advice parameter.

```
n1 = \arg -> e1 in
n2 = \arg -> e2 in
let h x = x in
let f dh x = dh x in
<f,{n2}> <h,{n1}> 1
```

The normalization in Rule (VAR-A) moves advice parameters out of $\langle\rangle$. As a result, the main expression becomes `(\y.<f y,{n2}>) <h,{n1}> 1`. The chaining participants `f y` and `n2` now have unifiable types.

We briefly describe the operational semantics of the translated language. In addition to the standard grammar of the base language, we introduce chaining as an expression and chaining of values as a value.

$$\begin{array}{lcl} v & ::= & \ldots \mid \langle v, \{\bar{v}\}\rangle \\ e & ::= & \ldots \mid \langle e, \{\bar{e}\}\rangle \end{array}$$

The set of β reductions are defined as follow:

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

These rules specify a call-by-value evaluation strategy. This choice of evaluation strategy is orthogonal to the language design. The first two rules are standard β-rules for lambda calculus. In the third rule, when the advice sequence is empty, the chained value $\langle v, \{\}\rangle$ evaluates to the original function v. Otherwise, in the fourth rule, the chained value $\langle v, \{v_1, \bar{v}\}\rangle$ replaces the $proceed$ in the body of the first advice v_1 by a chained value which chains the function v with the remainder of the advice sequence.

4.4 Post-Translation of Advice Body

Our translation changes definitions of those functions having advised types by lambda-abstracting them with additional advice parameters. When these functions are called inside the body of some advices, the advices become ill typed since the calls still refer to the old function definitions. For this reason, we use the following post translation to guarantee type preservation.

$$f : \forall \bar{a}.\overline{(x : p)}.t \leadsto_{PT} f\ \bar{x}$$

Note that in the above typing of f, we do not distinguish the $:$ and $:_*$ binding. Essentially, this post translation goes through typed advice bodies and replaces each function with advised type by an application of that function to the functions associated with its advice predicates. Contrary to the main translation defined in Figure 2, functions produced here are not chained with advices as we do not allow advice body to be advised.

4.5 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 1.)

Theorem 1 (Conservative Extension) *Given a program P consisting of a set of advices and a closed base program e. If*

$$\vdash_{\leadsto} P : \sigma \leadsto P',$$

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

$$\vdash_{\leadsto} P : \sigma \leadsto P',$$

and $P' \leadsto_{PT} P''$ during post-translation, then

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

5. Handling Advices within Advice

In many AOP languages, an advice is just like a function, calls to other functions may occur in its body, including functions that are advised. We call this *nested advice*. Admittedly, this might be a powerful tool for meta or reflective programming, yet we are conservative about it, for it is very likely to create more confusion than it is worth. Consider the following example code:

```
n@advice around {f} (arg) = f arg
```

A program having such an advice will go into infinite loop when `f` is called. For this reason, we do not include nested advices in our main translation rules. Nevertheless, we describe here a simple extension to our translation scheme to deal with nested advices, through the translation of the following example:

Example 7

```
n1@advice around {f} (arg::Int) = e1 in
n2@advice around {f} (arg) = e2 in
n3@advice around {g} (arg) = f arg
in let g x = x in
   let f x = x in
   let h x = g x in
   h 2
```

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.

Now we make an attempt to translate this program. The call of `g` in the definition of `h` is of type $\forall a.a \rightarrow a$. According to our (VAR-A) rule that performs sufficiently specific context check, `n3`, the only advice defined on `g`, is no more specific than $\forall a.a \rightarrow a$. Thus, `n3` is chained to the application of `g`.

If there were no `f`-call in the body of `n3`, this choice would be correct. Even for the main expression (`h 2`) where the call to `g` is restricted to $Int \rightarrow Int$, `n3` will still be the only applicable advice.

However, the nested nature of `n3` changes the story. At the time of chaining `n3` to `g` mentioned above, we must also advise the call to `f` in `n3`'s body. But the existing context $\forall a.a \rightarrow a$ is not sufficiently specific for `f` since `n1`'s type is more specific. Coherent advice cannot be enforced here.

To circumvent this problem, we suggest placing a stricter sufficiently specific context check in rule (VAR-A). This new check does not only check advices defined on the current looked up function, but also traces all the functions which are called by those advices and checks the current context against advices defined on these functions.

Using this stricter rule, we check not only the call context of `g` against the type of advice `n3`, but also those of `n1` and `n2`. This is because the call to `f` in `n3` might be advised by `n1` and/or `n2`. Thus, in the definition of `h` where the call context for `g` is $\forall a.a \rightarrow a$, we know that the call to `f` in `n3` is of type $\forall a.a \rightarrow a$. The check fails here because `n1` has a more specific type. Consequently, as per normal, we resort to applying rule (PRED) and abstracting the advice to this call. The actual chaining is only performed in the main expression where the context $Int \rightarrow Int$ is sufficiently specific for all the three advices.

The translated code is

```
let n1 = \arg -> e1 in
let n2 = \arg -> e2 in
let n3 = \arg -> f arg in
let g x = x in
let f x = x in
let h dg x = dg x in
h <g,{(let n3 = \arg -> <f,{n1,n2}> arg in n3)}> 2
```

In this program, the call to `g` is advised by `n3` which is in turn advised by `n1` and `n2`. In the case when `n1` and/or `n2` again call some functions with advised types, these functions also need to be advised. The sufficiently specific context check introduced above guarantees the success of this releasing of advices.

6. Related Works

Since the introduction of the aspect-oriented paradigm [13], researchers have been developing semantic foundations for it. Most of the works in this area were done in object-oriented context in which type inference, higher-order functions and parametric polymorphism are of little concern. Instead, they have been focusing on modelling the nature of pointcuts and the effects of executing the associated advices [4, 21, 9]. As a result, the semantics of aspect weaving is conveniently expressed through some mechanisms of dynamic semantics, and there has been either no definition of static semantics or not a concern for static semantics. Two recent proposals [16, 5] made pioneering attempts in incorporating aspect-oriented features into strongly typed functional languages. Although both emphasize the polymorphic aspect of pointcuts and advices, none of them is able to offer a complete solution to all those concerns.

Based on a polymorphic calculus with first-class join points, PolyAML [5] allows programmers to define polymorphic advices using type-annotated pointcuts. They designed a conservative extension to the Hindley-Milner type inference algorithm with a form of local type inference based on the required annotation of pointcuts. To support non-parametric polymorphic advice, they also introduced case-advices which are subsumed by our type-scoped advices. Weaving was done by a translation into the typed core calculus and dynamic type checking is employed to decide on the triggering of case-advices. PolyAML is a first-order language that does not support $around$ advice, so it does not address many of the issues we discussed in this paper.

Aspectual Caml [16], 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 advices hinged on the pointcut are consistent with the pointcut's type. 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 pointcut's type during type inference. Type safety of advice application is considered later in the weaving process. After type inference, their 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, it relies on a very strong assumption that there will not be any renaming or even nested polymorphic calls in the program. This is particularly impractical in a higher-order language.

In [16], Masuhara *et al.* propose a convenient way of simplifying curried pointcuts into non-curried ones. We adopt this technique but in a more expressive manner in our translation, as described in Section 4.1.3.

Our idea of advice chaining is partly inspired by the chain expression in [4] and the wrapping of advices in [21]. On the other hand, these techniques are described in dynamic weaving setting, which are vastly different from our approach.

Another closely related work is the dictionary translation of Haskell type classes [19]. Our notion of advised types and type-directed translation are directly inspired by it. The main technical difference here is the adaptation of the mechanism to a new context in a coherent way. In type classes, higher-order functions are not a concern and overlapping instances are precluded, whereas in aspect-oriented programming, in which advices with overlapping pointcuts are predominant, higher-order functions complicate the translation. Therefore, we must take substantially different typing and translation approaches to handle overlapping advices, with the objective of ensuring coherent translation in the presence of higher-order functions.

7. Conclusion

We propose a novel technique for type inference of aspect-oriented functional programs, featuring higher-order functions, curried pointcuts and overlapping type-scoped advices. Our type inference system also supports static weaving of advices into programs. This is accomplished by a source-level program translation.

We believe that aspect-oriented programming is a promising paradigm for constructing functional programs, because it has the potential to turn some of the program crafting techniques into systematic program development. For example, type-scoped advices give a new perspective to the existing work in modifying part of a function's definition based on types. Specifically, in [14, 15], the authors describe a Haskell system support to enable type-preserving change to function definitions. The problem handled there can be conveniently framed in the aspect-oriented perspective, where such changes to a function can be defined as a typed-scoped advice to the function. Looking forward, we envisage the use of aspects, through the help of type-scoped advices, to model domain-specific embedded languages [8, 6, 11, 1].

On the technical perspective, implementation of the type-inference system is current in progress. In addition, we would like to extend the core language described here to admit other forms of language constructs. We do not see any technical difficulty in handling aspects with more complicated pointcut designators. On the other hand, we believe that the combination of overloaded functions (*a.k.a.*, type classes) and aspects with type-scoped advices can be a powerful tool for program construction, even though it may overwhelm programmers with too many subtle type issues.

Acknowledgments

We would like to thank the anonymous referees for their insightful comments. We also thank the members of the PLS-II lab for their valuable comments. This research is partially supported by an NUS research grant R-252-000-138-112. The first author would like to specially thank Dr. Martin Sulzmann for bringing him to the world of type inference and translation.

References

[1] S. Anand, W.-N. Chin, and S.-C. Khoo. Charting patterns on price history. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 134–145, New York, NY, USA, 2001. ACM Press.

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

[3] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *Proc. Concur*. Springer-Verlag, 2004.

[4] C. Clifton and G. Leavens. Minimao: Investigating the semantics of proceed. In *Proceedings of the Foundations of Aspect-Oriented Languages*, 2005.

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

[6] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[7] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[8] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[9] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings of the 2003 European Conference on Object Oriented Programming*, pages 54–73. Springer, 2003.

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

[11] S. Peyton Jones, J. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM Press.

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

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. 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.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[15] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255, New York, NY, USA, 2004. ACM Press.

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

[17] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

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

[19] P. Wadler and S. 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.

[20] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.

[21] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, September 2004.

A. A Sample Derivation

In this section, we present the typing/translation derivation of the program in Example 4. The code is reproduced below.

```
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)
```

We use I as a short hand for Int to save space. Some obvious details are also omitted.

The derivation of the definition of f is:

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

$$\text{(Pred)}\ \dfrac{\text{(Abs)}\ \dfrac{\text{(App)}\ \dfrac{\text{(Var)}\ \dfrac{h : t \to t \leadsto dh \in \Gamma_2}{\Gamma_2 \vdash_{\leadsto} h : t \to t \leadsto dh} \qquad \text{(Var)}\ \dfrac{x : t \leadsto x \in \Gamma_2}{\Gamma_2 \vdash_{\leadsto} x : t \leadsto x}}{\Gamma_2 = \Gamma_1, x : t \leadsto x \vdash_{\leadsto} (h\, x) : t \leadsto (dh\, x)}}{\Gamma_1 = \Gamma, h : t \to t \leadsto dh \vdash_{\leadsto} \lambda x.(h\, x) : t \to t \leadsto \lambda x.(dh\, x)}}{\Gamma \vdash_{\leadsto} \lambda x.(h\, x) : (h : t \to t).t \to t \leadsto \lambda dh.\lambda x.(dh\, x)}$$

The derivation of the main expression is:

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

$$\text{(App)}\ \dfrac{\text{(Rel)}\ \dfrac{\text{(Var)}\ \dfrac{f : \forall a.(h : a \to a).a \to a \leadsto f \in \Gamma_3}{\Gamma_3 \vdash_{\leadsto} f : (h : I \to I).I \to I \leadsto f} \qquad ⓐ}{\Gamma_3 \vdash_{\leadsto} f : I \to I \leadsto (f\ \langle h\ , \{n_1, n_2\}\rangle)} \qquad \ldots}{\Gamma_3 \vdash_{\leadsto} (f\ 1) : I \leadsto (f\ \langle h\ , \{n_1, n_2\}\rangle\ 1)}$$

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

The translated result is displayed below :

```
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)
```