Type-Directed Weaving of Aspects for Polymorphically Typed Functional Languages

Kung Chen[a], Shu-Chun Weng[b], Meng Wang[c], Siau-Cheng Khoo[d], Chung-Hsin Chen[a]

[a] *National Chengchi University*
[b] *National Taiwan University*
[c] *Oxford University*
[d] *National University of Singapore*

Abstract

Incorporating aspect-oriented paradigm to a polymorphically typed functional language enables the declaration of *type-scoped advice*, in which the effect of an aspect can be harnessed by introducing possibly polymorphic type constraints to the aspect. The amalgamation of aspect orientation and functional programming enables quick behavioral adaption of functions, clear separation of concerns and expressive type-directed programming. However, proper *static weaving* of aspects in polymorphic languages with a type-erasure semantics remains a challenge. In this paper, we describe a type-directed static weaving strategy, as well as its implementation, that supports static type inference and static weaving of programs written in an aspect-oriented polymorphically typed functional language, AspectFun. We show examples of type-scoped advice, identify the challenges faced with compile-time weaving in the presence of type-scoped advice, and demonstrate how various advanced aspect features can be handled by our techniques. Lastly, we prove the correctness of the static weaving strategy with respect to the operational semantics of AspectFun.

Key words: Aspect-oriented programming, Type-scoped advice, Static weaving, Polymorphically-typed functional language

1. Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut components of a software system[11].

In AOP, a program consists of many functional modules and some *aspects* that encapsulate the crosscutting concerns. An aspect provides two kinds of specification: *pointcut*, comprising a set of functions, designates when and where to crosscut other modules; and *advice*, which is a piece of code, that will be *triggered* for execution when the corresponding pointcut is reached during runtime. The complete program behavior is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. Such composing activity can be done at compile-time or runtime, and is referred to as *weaving* in AOP. Weaving results in the behavior of those functional modules impacted by aspects being modified accordingly.

While majority of the developments of AOP have been based on the object-oriented (OO) paradigm, there has been increasing awareness that the idea of AOP, if not the exact mechanism developed in the OO setting, is able to offer distinguished benefit to conventional functional languages in terms of modularity [23, 22]. To start with, let's consider a simple example of sorting a list. Assuming we already have a function `sort :: [a] -> [a]` that implements the quicksort algorithm and picks the pivot from the head. For specific application domains, it is generally very useful if we can augment the algorithm with some domain knowledge in a modular fashion. For example, in an application of predominately nearly sorted lists, the following aspect provides a special case for already sorted lists. (Our aspect language employs a syntax very similar to that of Haskell. Detailed syntax will be presented in the following section.)

```
opt@advice around {sort} (arg) =
    if isSorted arg then arg else proceed arg
```

This piece of code defines an aspect with the name `opt`, which designates `sort` as the *pointcut*. Effectively, this aspect watches function `sort` and executes its advice body when `sort` is called with an input that binds to `arg`. Since quicksort calls itself internally, nearly sorted lists also benefit from this aspect by having more efficient recursive invocations. The special function `proceed`, which may be called inside the body of *around* advice, is bound to a function that represents "the rest of the computation at the advised function"; specifically, it enables control to be reverted to the advised function, such as `sort`. An important difference between calling `proceed` and the actual function, say `sort`, is that `proceed` does not trigger the same advice again.

The same predicate, `isSorted`, can be used to impose *contracts* that are separated modularly from the main functional concern.

```
corr@advice around {sort} (arg) =
    let res = proceed arg
    in if isSorted res then res else error "Not Sorted!"
```

This contract aspect performs the computation first by calling `proceed`; it then takes over the returned result and checks for its sortedness. Note that the `error` function is a built-in function of Haskell whose type is `String->a`.

Function `sort` is polymorphic, which works uniformly on all input lists with comparable elements. From time to time, we may want to adapt this generic behavior for some specific (set of) types. For example, suppose later in the development, we add into the system some 32 bits binary numbers encoded as records for constant access to each digit. Pair-wise comparison on them is thus expected to be expensive. We can then switch to the more suitable radixsort algorithm.

```
radix@advice around {sort} (arg::[Binary]) = radixSort arg
```

This aspect includes a type constraint `[Binary]` on its pointcut which limits the scope of its impact through type scoping on its argument; this is called a *type-scoped* advice. This means that execution of `radix` will be triggered only when `sort` is invoked with an argument of such list elements.

The advantage of using aspects is evident. Improvement to the existing program can be done modularly with easy deployment and retraction of aspects. Since multiple pieces of advice can be attached to the same point and executed in sequence, the aspects above can be picked and matched freely.

Though small, the sort example gives a glimpse of three important applications of AOP in functional programming that are summarized below.

1. **Behavioral Adaptation** : Aspect `opt` makes function `sort` behave differently for the special case of sorted list based on the inherited recursive structure of function `sort`.
2. **Separation of Concerns** : Aspect `corr` allows contracts to be imposed on function `sort` separately from the functional component.
3. **Type-directed Programming** : Aspect `radix` augments function `sort` with type specific behaviors.

Given these distinct benefits, it is attractive to introduce AOP into functional languages. Indeed, notable proposals of AOP extensions have been made for ML [5, 15]. However, proper *static weaving* of aspects in languages with type-erasure semantics such as Haskell remains a challenge. Specifically, it is difficult to determine statically the exact type context of an invocation of a polymorphic function in order to ensure proper weaving, at compile time, of aspects with type scopes. For example, consider the following program

```
sortcat l = concat ((map sort) l)
```

When compiling `sortcat`, it is not clear whether aspect `radix` should be triggered as the element type of parameter `l` is not known.

Not only does this problem exist in the functional setting, it also exists in any AOP language with type-erasure semantics and parametric polymorphism. For example, as pointed out in [8], correct static weaving of aspects are threatened by the introduction of *generics* in Java. The following Java code is taken from [8]:

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

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

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

However, for those invocations of `max` that occur inside another polymorphic method, we shall run into the same difficulty of static weaving as described above. Furthermore, due to the type-erasure semantics of Java, runtime type test of the list element type is not feasible.

In this paper, we present a type-directed aspect weaving scheme for polymorphically typed functional languages that can solve this problem with static weaving. In particular, the mature technology of compile-time dictionary translations makes Haskell a perfect environment for our proposal. We

consolidate our past research in this field [21, 20, 2] and makes significant revisions and extensions to several dimensions of our research. Moreover, we illustrate our scheme with an experimental language, AspectFun, and provide the following:

1. A complete treatment of static and consistent weaving for the core features of AspectFun, including type-scoped advice and nested advice (whose body is also advised).
2. A full formulation of the correctness of static weaving *wrt* the operational semantics of AspectFun and its proof.
3. A complete implementation of our static weaving scheme which turns aspect-oriented functional programs into executable Haskell code without aspects[1].

The outline of the paper is as follows: Section 2 presents our experimental language AspectFun, highlighting various aspect-oriented features our scheme supports through examples in AspectFun. Section 3 defines an operational semantics for AspectFun. In Section 4, we describe our type inference system and the corresponding type-directed static weaving process. Next, we formulate the correctness of static weaving with respect to the semantics of AspectFun. Finally, we discuss related work in Section 6 and conclude in Section 7. Appendix A provides the detailed proof of the correctness of static weaving.

2. AspectFun: The Aspect Language

This section introduces the aspect-oriented functional language, AspectFun, for our investigation. We shall first describe the core features of AspectFun, and outline the compilation process we employ to implement it. Then we shall present some example applications of AspectFun.

2.1. Language Features

Figure 1 presents the language syntax[2]. We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables etc) and fv(o)

[1]The implementation is available at http://of.openfoundry.org/projects/801/

[2]To simplify the presentation, we leave out type annotations, user-defined data types, if expressions, patterns and sequencings (;), but may make use of them in examples.

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

Figure 1: Syntax of the AspectFun Language

as the set of free variables in o. We assume that $\bar{o}$ and o, when used together, denote unrelated objects.

In AspectFun, top-level definitions include global variables and function definitions, as well as aspects. An *aspect* is an advice declaration which includes a piece of advice and its target *pointcuts*. The prefix part, $n@$, of an advice declaration simply names the advice under n. Pointcuts are denoted by $\{\overline{pc}\}\ (arg)$, where pc stands for either a *primitive pointcut*, represented by ppc, or a *composite pointcut*. Pointcuts specify certain *join points* in the program in which advice is triggered when program execution reaches there. Here, we focus on join points at function invocations. Thus a primitive pointcut, ppc, specifies a function or advice name the invocations of which, either directly or indirectly via functional arguments, will be advised. Furthermore, the applicability of a piece of advice is bounded by its pointcut as well as its optional type scope, which is specified as part of the arg component, namely $x :: t$.

Advice is a function-like expression that may be executed *before*, *after*, or *around* a join point. An *around* advice is executed in place of the indicated join point, allowing the advised pointcut to be replaced. A special keyword `proceed` may be used inside the body of around advice. It is bound to

the function that represents "the rest of the computation" at the advised pointcut. As both *before* advice and *after* advice can be simulated by *around* advice that uses `proceed`, we only need to consider *around* advice in this paper.

A primitive pointcut can also be a catch-all keyword any. When used, the corresponding advice will be triggered whenever a named function is invoked. For example, the pointcut $\mathsf{any}\backslash[f, g]$ will select all named functions except f and g. Besides, since advice is also named, we allow advice to advise other advice. A sequence of pointcuts, $\{\overline{pc}\}$, indicates the union of all the sets of join points selected by the pc_i's. The argument variable *arg* is bound to the actual argument of the named function call and it may contain a type scope.

Note that, since our pointcuts are name-based, invocations of anonymous functions are not considered as join points, even when any is used. Besides, only global functions and advice are subject to advising. Although our weaving scheme can also handle local functions, we choose not to do so for it will make the base program not oblivious to the alpha conversion. On the other hand, because of this decision, we need to apply alpha renaming to local declarations beforehand so as to avoid name clashes.

In passing, we note two other features of primitive pointcuts. First, in AspectFun, advice is named and their names can appear in a pointcut. Thus we allow advice to be developed to advise other advice. We refer to such advice as *second-order advice*. Second, the function name in a primitive pointcut can be followed by an optional sequence of arguments to support advising on partially applied functions. Following [15], we refer to such pointcuts as *curried pointcuts*,

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

Following AspectJ, our aspect language also provides two kinds of pointcut designators for specifying control flow restrictions. The first one is expressed as $\mathsf{cflow}(f)$, and it captures all the join points in the control flow from the the specific application to function f, including that specific f-

application. The second one is expressed as cflowbelow(f), and it captures all the join points in the control flow from the specific application to f, but excluding that specific f-application.

Lastly, the expressions in AspectFun are pretty standard. As to the types, we introduce a conservative extension of the standard Hindley-Milner type schemes which includes the so-called *advice predicates* to form *advised types*, ρ. This construct is inspired by the *predicated types* [18] used in Haskell's type classes. Advised type augments common type scheme with *advice predicates*, $(f : t)$, to capture the need of static advice weaving based on type context. We shall explain them in detail in Section 4.1.

Before ending this part, we outline the implementation scheme we employ for compiling AspectFun programs. The target language is Haskell [6]. The overall compilation process of AspectFun can be divided into ten steps, as outlined in Figure 2. Briefly, the process comprises the following five major components: (1) Syntactic processing and dependency analysis of an AspectFun program; (2) Static type inference to add type information to the abstract syntax tree; (3) Type-directed static weaving to convert aspects to functions and produce a piece of woven code; (4) Analysis and optimization of the woven code;(5) Translation of a woven program into a Haskell program. The first component is pretty standard. The second component performs a Hindley-Milner like type inference to reconstruct type information by treating advice as normal functions with proceed calls as recursive calls. Section 4 will present the third and the last component, which are the major results of this paper. The readers are referred to [2] for the fourth component,

2.2. Examples

As outlined in the introduction, there are three major applications of functional AOP, namely behavioral adaptation, separation of non-functional concerns, and type specific behavior, which distinguish it from traditional functional programming. In the sequel of the section, we illustrate these three points in more detail with examples. The complete AspectFun programs of these examples are available in the distribution of the AspectFun compiler.

2.2.1. Behavioral adaptation

AOP enables us to adapt and reuse existing code in a modular fashion. Let's consider an example of monadic evaluators [17] for the lambda calculus.

Example 1.

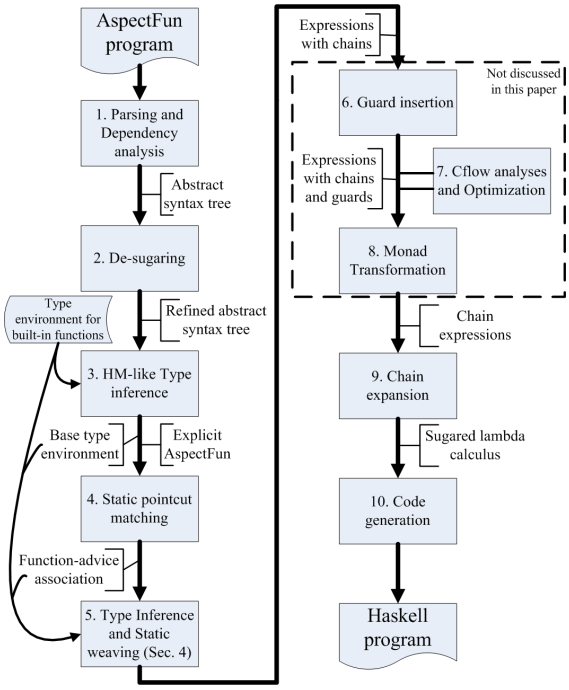

Figure 2: Compilation Process of AspectFun

```
data Term =  Var String
          |  Lam String Term
          |  App Term Term

eval :: Term -> M Term
eval (Var n)      =  return (Var n)
eval (Lam n t)    =  return (Lam n t)
eval (App t1 t2)  =  do t1' <- eval t1
                        t2' <- eval t2
                        case t1' of
                          Lam n t -> eval (subst (n,t2') t)
                          t       -> return (App t t2')
```

The evaluator, `eval`, reduces a lambda term using a monad, `M`. The default evaluation strategy above is call-by-value. A definition of a call-by-name evaluator will be very similar and only differs in the `App` case. Instead of defining two separate functions that are largely overlapping, we can treat the above definition as a 'template' function and override it later by aspects.

```
cbn@advice around {eval} (e) =
   case e of (App t1 t2) ->
        do t1' <- eval t1
           case t1' of Lam n t -> eval (subst (n,t2) t)
                       t       -> return (App t t2)
             _       -> proceed e
```

Note that, inside the body of advice `cbn`, there are two calls to the function eval, which is being advised. We call such advice, whose body is also advised, *nested advice*.

Aspects do in-place modification of the target functions, which makes the original definitions inaccessible in the same scope. In the above example, the semantics of `eval` is changed to call-by-name by `cbn`; and we loses the original call-by-value evaluator. A way to avoid this problem is to alias the template function and advise the new name as follows.

```
evalcbn = eval
cbn@advice around {eval+cflow(evalcbn)} (e) =
   case e of (App t1 t2) ->
        do t1' <- eval t1
           case t1' of Lam n t -> eval (subst (n,t2) t)
                       t       -> return (App t t2)
             _           -> proceed e
```

In this version, advice `cbn` employs a control-flow based pointcut which only applies to recursive calls to `eval` originated from `evalcbn`. We can still invoke the original call-by-value evaluator by calling `eval` directly.

Without aspects, the idiomatic way of achieving reuse of recursive pattern in functional programming is through higher-order combinators, such as *fold*. However, advanced planning is required; and programs must be written in a particular syntactic style.

2.2.2. Separation of Non-functional Concerns

The signature application of AOP is the modular tracing. There is no doubt that it is also useful in functional programming as well. Let's consider a simple example for illustration (Example 2).

The code in Example 2 defines two aspects named `n1` and `n2` respectively; it also defines a main/base program consisting of declarations of `f` and `h` and a main expression returning a triplet. The first aspect `n1` employs the all-catching pointcut, `any`, to trace the execution of all functions in the main

Example 2.

```
--Tracing Aspects
n1@advice around {any} (arg) =
   println "entering " ++ tjp;
   proceed arg in
n2@advice around {f} (arg::[Char]) =
   print " argument string ";
   println arg;
   proceed arg  in
--Base program
f x = x in
h x = f x in
(f 10, f "c", h "d")
```

```
// Execution trace
entering f
entering f
 argument string: "c"
entering h
entering f
 argument string: "d"
```

program. Inside the advice, a special run-time reflection `tjp` (standing for *this join point*) refers to the name of the advised function. Very often, for polymorphic or overloaded functions, we want to have more refined messages that reflect the type context of the executions. In aspect `n2`, there is a type-scope on the first argument. In addition to the generic trace produced by aspect `n1`, aspect `n2` prints out function `f`'s string (list of Char) inputs. The result of deploying the two aspects are shown to the right of the example code[3].

2.2.3. Type Specific Behavior

We have seen examples that exhibit type specific behavior with type-scoped advice. This kind of type-directed programming is commonplace in functional programming and the modularity brought in by AOP is highly desirable as argued in [23]. In this section, instead of showing more examples of purely static resolution of type-directed functions that readers are already accustomized to, we look at a functional idiom of *Generic Programming* [7] that generally requires some dynamic typing mechanisms and show how the extensibility of AOP plays a crucial role in constructing such an idiom [23, 22].

Type-directed programming allows us specify a case for every data type. This is fine-grained, but not very general: we cannot write reusable defini-

[3]Our compiler employs the composition of two Haskell functions, (unsafePerformIO . putStrln), to implement the "println" operation. Moreover, the sequencing construct, ";", is implemented in terms of the 'seq' facility of Haskell.

tions that explore structural similarities among types. Consequently, many *boiler-plate* codes are created [12]. Consider a function `strings` that extracts all the strings stored in a structure. With a nominal approach, we are required to define a case for every data type, which mainly specifies non-productive inductive traversals.

In contrast, generic programming is about defining functions that work for all types but that also exhibit type-specific behavior [7]. It exploits structural information of data types, and dispatches based on structural representations. For example, the *Spine* type defined below is a general and uniform way of representing elements of a data type that can support the definition of generic functions, such as `strings`.

```
data Spine a = Con (Constr a)
             | forall b. App (Spine (b -> a)) b

data Constr a = Descr a
```

If a constructor does not take any argument, it is encoded by `Con` together with information about the constructor. Otherwise, a constructor taking arguments is encoded by applying `App` to the representation of the constructor and to its arguments. The function `toSpine`, which converts a data type to its spine representation, can be defined in a type-directed manner.

```
toSpine :: a -> Spine a
toSpine x = undefined

int@advice around {toSpine} (arg::Int) = Con (Descr 0)
char@advice around {toSpine} (arg::Char) = Con (Descr 'a')
list@advice around {toSpine} (arg::[a]) =
    case arg of []     -> Con (Descr [])
                (x:xs) -> App (App (Con (Descr (:))) x) xs
...
```

Type-scoped advice is put into good use to bring in new cases of `toSpine` for the ever-growing set of datatypes. For example, `toSpine [1,2,3]` produces

```
App (App (Con (Descr (:)) 1) [2,3])
```

This shallow encoding is pushed inwards by generic functions that make use of it, as we will see shortly. Since all data types are now mapped to a single

one, `Spine`, we can easily define functions that work on this representation. For example, the following code collects strings from a structure.

```
strings :: a -> [String]
strings x = strings' (toSpine x)

strings' :: Spine a -> [String]
strings' (Con c)    = []
strings' (App f x)  = strings' f ++ strings x
```

The intention of the above program is to uniformly traverse the structures of any data types (including strings seen as lists of Chars). To be able to collect strings, we need a small exception of this generic behavior that returns a string when the input is a string. This is another type-directed operation. It is tempting to use type-scoped advice here to advise `strings`. However, we notice that the call to `strings` in the body of `strings'` is given the second argument of `App` as input, whose type is existentially quantified in the definition of `Spine` and is not available statically. This makes type-scoped advice, together with any other type-directed-programming mechanisms that relies on static resolution, not applicable.

A standard technique for handling this exception case, which can be found in the generic programming literature, is to use dynamic "type" testing based on some kinds of term encodings of types [12, 13, 7]. Independently, encoding of dynamic type casting in statically type languages [24, 3] is also available. Here, we choose to follow [7] by wrapping a value of type `a` with a type representation to form a data type, `Typed a`, and use a `cast` function that compares the type representation with a target type. As a result, the exceptional case of string inputs can be handled by the following advice.

```
n@advice around {strings} (x) =
   case cast x :: Maybe String of Just s  -> [s]
                                  Nothing -> proceed x
```

This advice intercepts all executions of `strings`. When the input is dynamically verified to be a string, we return that string in the result; otherwise, control is passed back to `strings` if there is no other intercepting advice. (We need to adapt the earlier definition of `toSpine` to accept `Typed a` as its argument type. The detail is omitted here.)

We think this ability of accepting dynamic type casting with aspects is one of the strengths of AOP, since it allows modular extensions. Suppose we

later implement a datatype of ASCII code of characters and wish to consider a list of ASCII's as a string, function `strings` can be easily extended with another special case using the following aspect.

```
n1@advice around {strings} (x) =
   case cast x :: Maybe [Ascii] of Just s  -> [s]
                                   Nothing -> proceed x
```

This modular extensibility is difficult to achieve with other type-directed approaches such as Haskell type classes. It is worth mentioning that just like encodings of dynamic typing do not obsolete static typing, the use of type casts in advices does not replace static weaving.

3. The semantics of AspectFun

This section presents an operational semantics for AspectFun. As type information is required at the triggering of advice for execution, our semantics is presented in terms of an explicitly typed version of AspectFun, referred as EA in the following discussion. Figure 3 displays the syntactic constructs of EA.

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

Figure 3: Syntax of EA

The syntactic structure of EA remains the same as that of AspectFun. The main enhancements are four type-related constructs at the expression level. The constructs of type applications and type abstractions are standard ones; they are denoted by $e\{t\}$ and $\Lambda a.e$, respectively. The type annotations for lambda parameters are also common in explicitly typed languages. The only new construct is a set of *labels* which annotate lambda expressions that can be the target of advice weaving. Essentially, a label specifies the name and the type of a function with which the lambda expression is associated via a top-level declaration. Hence, a label identifies a join point and its type context.

The types and type schemes of EA follow the convention of the Hindley-Milner type system. The semantics specification of EA includes the following notations on types. We write $t \trianglerighteq t'$, denoting that type t is more general than or equivalent to type t', iff there exists a substitution S over type variables in t such that $St = t'$, and the notation, $S = t \trianglerighteq t'$, is an abbreviation for it. We write $t \equiv t'$ iff $t \trianglerighteq t'$ and $t' \trianglerighteq t$. When $t \trianglerighteq t'$ but $t \not\equiv t'$, we say t is more general than t'. Similarly, we say a type t is more specific than a type t' if $t' \trianglerighteq t$ and $t \not\equiv t'$. Finally, the most general unifier between two types, t and t', is denoted by $mgu(t, t')$.

Conversion from AspectFun to EA is done by the standard Hindley-Milner type inference with few straightforward enhancements. First, type abstractions and type applications are made explicit. Second, when inferring the type for a piece of advice, invocation of the underlying advised function via *proceed* is treated as a recursive function call. Finally, top-level lambda expressions are annotated with labels that specify the name and type of the function they define.

3.1. Operational Semantics for EA

The operational semantics for EA is specified in terms of the judgement $\mathcal{E}; \mathcal{A} \vdash \pi \Downarrow v$, where π is an EA program, $\mathcal{E}$ is an environment that maintains the bindings of variables and functions, and $\mathcal{A}$ is a repository that keeps advice-related information derived from advice declarations in π. We shall often refer to the environment and advice store pair as the operational semantics context for brevity. The full semantic domains and the set of environment-based, big-step reduction rules to define the judgement are shown in Figure 4 and Figure 5, respectively.

Expressions	e	$::=$	$\cdots \mid th$
AdvStore	$\mathcal{A}$	$::=$	$\overline{\text{Adv}}$
Advice	Adv	$::=$	$(n : \sigma, \overline{pc}, t, e)$
Environments	$\mathcal{E}$	$::=$	$\overline{x \mapsto th}$
Values	v	$::=$	$c \mid cl$
Thunk	th	$::=$	$(\!\mid e, \mathcal{E} \mid\!) \mid cl$
Closure	cl	$::=$	$(\!\mid \lambda^l x : t.\ e, \mathcal{E} \mid\!) \mid (\!\mid \Lambda a.\ e, \mathcal{E} \mid\!)$

Figure 4: Semantic Domains for EA

We adopt call-by-name evaluation for EA[4]. Thus we add a new form of expression, called *thunk*, which is only used in the operational semantics, but not in the source expression. A thunk is a pair of an expression and an environment. When the expression in a thunk is a lambda expression or a type abstraction, the thunk is called a *closure*. Constants and closures are considered values in EA and will not be further evaluated. Finally, during the evaluation of an EA program, the environment associates a name with a thunk as its binding. We write $\mathcal{E}[y \mapsto th]$ for the environment which extends $\mathcal{E}$ by assigning thunk th to variable y, assuming that any name clash has been resolved via proper renaming.

Among the reduction rules for EA three rules, namely (OS:Decl), (OS:Adv) and (OS:Adv-An), process top-level declarations; and the rest of rules handle various forms of expressions. Rule (OS:Decl) makes a thunk out of the defining expression of a global variable or function and the current environment, and then puts it into the environment for further evaluation of the underlying program. On the other hand, rules (OS:Adv) and (OS:Adv-An) collect the set of advice declared in a program and deposit it in the advice store, $\mathcal{A}$. Both $\mathcal{A}$ and $\mathcal{E}$ are essential to the evaluation of the expression inside an EA program.

The (OS:Adv) rule simply delegates the collection task to the (OS:Adv-An) rule, making non-type-scoped advice a special case of type-scoped advice with the inferred parameter type as the scope. The (OS:Adv-An) performs the real work of advice collection: organizing a type-scoped advice into a quadruple, $(n : \sigma, \overline{pc}, t, e)$, and appending it to the advice store $\mathcal{A}$. The quadruple

[4]Note that Haskell uses call-by-need evaluation.

$$
\begin{array}{ll}
\text{(OS:Value)} & \mathcal{E};\mathcal{A} \vdash c \Downarrow c \\
& \mathcal{E};\mathcal{A} \vdash (\!|\lambda^l x : t.\ e, \mathcal{E}'|\!) \Downarrow (\!|\lambda^l x : t.\ e, \mathcal{E}'|\!) \quad \mathcal{E};\mathcal{A} \vdash (\!|\Lambda a.\ e, \mathcal{E}'|\!) \Downarrow (\!|\Lambda a.\ e, \mathcal{E}'|\!) \\
\text{(OS:Lamb)} & \mathcal{E};\mathcal{A} \vdash \lambda^l x : t_x.\ e \Downarrow (\!|\lambda^l x : t_x.\ e, \mathcal{E}|\!) \qquad \mathcal{E};\mathcal{A} \vdash \Lambda a.\ e \Downarrow (\!|\Lambda a.\ e, \mathcal{E}|\!)
\end{array}
$$

$$
\text{(OS:Thunk)} \quad \frac{\mathcal{E}';\mathcal{A} \vdash e \Downarrow v}{\mathcal{E};\mathcal{A} \vdash (\!|e, \mathcal{E}'|\!) \Downarrow v} \quad e \text{ is not an abstraction}
$$

$$
\text{(OS:App)} \quad \frac{\begin{array}{c}\mathcal{E};\mathcal{A} \vdash e_1 \Downarrow (\!|\lambda^l x : t_x.\ e_3, \mathcal{E}'|\!) \quad \mathit{Trigger}((\!|\lambda x : t_x.\ e_3, \mathcal{E}'|\!), l) = (\!|\lambda y : t_y.\ e_4, \mathcal{E}''|\!) \\ \mathcal{E}''[y \mapsto (\!|e_2, \mathcal{E}|\!)];\mathcal{A} \vdash e_4 \Downarrow v\end{array}}{\mathcal{E};\mathcal{A} \vdash e_1\ e_2 \Downarrow v}
$$

$$
\text{(OS:Ty-App)} \quad \frac{\mathcal{E};\mathcal{A} \vdash e_1 \Downarrow (\!|\Lambda a.\ e_2, \mathcal{E}'|\!) \quad \mathcal{E}';\mathcal{A} \vdash [t/a]e_2 \Downarrow v}{\mathcal{E};\mathcal{A} \vdash e_1\{t\} \Downarrow v}
$$

$$
\text{(OS:Let)} \quad \frac{\mathcal{E}[x \mapsto (\!|e_1, \mathcal{E}|\!)];\mathcal{A} \vdash e_2 \Downarrow v}{\mathcal{E};\mathcal{A} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v} \quad \text{(OS:Var)} \quad \frac{[x \mapsto e] \in \mathcal{E} \quad \mathcal{E};\mathcal{A} \vdash e \Downarrow v}{\mathcal{E};\mathcal{A} \vdash x \Downarrow v}
$$

$$
\text{(OS:Decl)} \quad \frac{\mathcal{E}[id \mapsto (\!|e_1, \mathcal{E}|\!)];\mathcal{A} \vdash \pi \Downarrow v}{\mathcal{E};\mathcal{A} \vdash id = e_1\ \mathsf{in}\ \pi \Downarrow v}
$$

$$
\text{(OS:Adv)} \quad \frac{\begin{array}{c}\forall \bar{a}.t_1 \to t_2 = \sigma \\ \mathcal{E};\mathcal{A} \vdash n :: \sigma@\mathsf{advice\ around}\ \{\overline{pc}\}\ (x :: t_1) = e_1\ \mathsf{in}\ \pi \Downarrow v\end{array}}{\mathcal{E};\mathcal{A} \vdash n :: \sigma@\mathsf{advice\ around}\ \{\overline{pc}\}\ (x) = e_1\ \mathsf{in}\ \pi \Downarrow v}
$$

$$
\text{(OS:Adv-An)} \quad \frac{\begin{array}{c}\forall \bar{a}.t_1 \to t_2 = \sigma \\ \mathcal{E};\mathcal{A}.(n : \sigma, \overline{pc}, t_1, (\!|\Lambda \bar{a}.\lambda^{n:t_1 \to t_2} x : t_1.e_1, \mathcal{E}|\!)) \vdash \pi \Downarrow v\end{array}}{\mathcal{E};\mathcal{A} \vdash n :: \sigma@\mathsf{advice\ around}\ \{\overline{pc}\}\ (x :: t_1) = e_1\ \mathsf{in}\ \pi \Downarrow v}
$$

Figure 5: Operational Semantics for EA

consists of advice name-type pair, pointcut, type scope and thunkified advice body.

As to the reduction rules for expressions of EA they mostly follow the standard ones for a typed lambda calculus with constants. The only exception is the rule for function application, (OS:App), which also handles

the triggering and weaving of advice. Specifically, the closure to apply and the associated label are passed to the function *Trigger*, which is specified in Figure 6 together with other auxiliary function declarations. The advice triggering function first chooses the set of eligible advice based on argument type, and weaves them into the function invocation – through a series of environment extension of advice closures – for execution. Note that, although we adopt the call-by-name evaluation strategy, our weaving scheme does not rely on this choice.

Three points worth mentioning here. First, in the main body of *Weave* function, the function *Trigger* is invoked again to handle any possible triggering of second-order advice. Second, among the advice matched by *JPMatch*, the function *Choose* keeps all the advice whose type scope is more general than the type passed to it, regardless of its return type. Consequently, it is likely that, during the subsequent execution of the woven advice, a runtime type error may occur and the reduction fails (unless, of course, the program has been analyzed to be safe by our type system). Third, the set of advice selected by *Choose* is kept in a list and ordered according to the sequential ordering of their declarations in the program. While we believe that the issue of chaining order is orthogonal to our study here, it is understood that advice is to be chained in a specific order during execution. Hence, we fix the order in our semantics definition, and assume that the order chosen during static weaving (Section 4) is the same.

3.2. Example

We use a contrived example to demonstrate how the semantics of AspectFun works. The AspectFun program listed in Example 3 includes three kinds of advice, namely type-scoped advice, polymorphic advice and second-order advice. They will be triggered according to the type context at different join points during the execution of the program.

Example 3.

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

$$
\begin{array}{lll}
\mathit{Trigger} & : & e \times l \to e \\
\mathit{Trigger}(e, \epsilon) & = & e \\
\mathit{Trigger}(\llparenthesis \lambda x : t_x.\ e, \mathcal{E}_f \rrparenthesis), f : t_f) & = & \mathit{Weave}(\llparenthesis \lambda x : t_x.\ e, \mathcal{E}_f \rrparenthesis), t_f, \mathit{Choose}(f, t_x))
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Weave} & : & e \times t \times \overline{\mathrm{Adv}} \to e \\
\mathit{Weave}(e, t_f, []) & = & e \\
\mathit{Weave}(e_f, t_f, \mathit{adv} : \mathit{advs}) & = & \text{Let} \quad (n : \forall \bar{a}.\ t_n, \overline{pc}, t, (\llparenthesis \Lambda \bar{a}.\ e, \mathcal{E}_n \rrparenthesis)) = \mathit{adv} \\
 & & \qquad\quad \bar{t} \text{ be types such that } [\bar{t}/\bar{a}]t_n = t_f \\
 & & \qquad\quad e_p = \mathit{Weave}(e_f, t_f, \mathit{advs}) \\
 & & \qquad\quad (\lambda^{n:t'_n} x : t_x.\ e_a) = [\bar{t}/\bar{a}]e \\
 & & \text{In} \quad \mathit{Trigger}(\llparenthesis \lambda x : t_x.\ e_a, \mathcal{E}_n.\mathit{proceed} = e_p \rrparenthesis), n : t'_n)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Choose}(f, t) = [(n_i : \sigma_i, \overline{pc_i}, t_i, e_i) \mid \ (n_i : \sigma_i, \overline{pc_i}, t_i, e_i) \in \mathcal{A},\ t_i \trianglerighteq t, \\
\qquad\qquad\qquad\qquad\qquad\qquad \exists pc \in \overline{pc_i} \ s.t.\ \mathit{JPMatch}(f, pc)] \\
\mathit{JPMatch}(f, pc) = (f \equiv pc) \vee (pc \equiv \mathsf{any}) \vee (pc \equiv \mathsf{any} \setminus [\bar{h}] \wedge f \notin \bar{h})
\end{array}
$$

Figure 6: Operational Semantics for EA: Auxiliary Function Declarations

The EA version of the above program is as follows:

```
nscope ::  (∀ a.[a]→[a])@advice around {f} (arg::[a]) =
                                          proceed (tail{a} arg) in
n      ::  (∀ a.a→t_g)@advice around {g} (arg) = proceed arg in
n2nd   ::  (∀ a.a→t_g)@advice around {n} (arg) = proceed arg in
f      = Λa.λ^{f:a→a} x:a. x in
g      = Λa.λ^{g:a→t_g} x:a. (f{a} x, f{(a, a)} (x, x), f{[a]} [x]) in
h      = Λa.λ^{h:a→t_h} x:a. g{[a]} [x] in
k      = Λa.λ^{k:a→t_g} x:a. g{a} x in
(h{Int} 1, k{Int} 2)
```

where t_g and t_h are abbreviations for `(a, (a, a), [a])` and `([a], ([a], [a]), [[a]])`, respectively.

After applying the advice collection procedure to the above program, we get the following advice store:

$$
\begin{array}{llllll}
(\texttt{nscope} & : \forall a.[a] \to [a], & \texttt{f}, & [a], & \Lambda a.\lambda^{\texttt{nscope}:[a]\to[a]}\texttt{arg} : [a].\ \texttt{proceed (tail\{a\} arg)}), \\
(\texttt{n} & : \forall ab.a \to b, & \texttt{g}, & a, & \Lambda a.\Lambda b.\lambda^{\texttt{n}:a\to b}\texttt{arg} : a.\ \texttt{proceed arg}), \\
(\texttt{n2nd} & : \forall ab.a \to b, & \texttt{n}, & a, & \Lambda a.\Lambda b.\lambda^{\texttt{n2nd}:a\to b}\texttt{arg} : a.\ \texttt{proceed arg})
\end{array}
$$

Then we apply the big-step reduction rules to evaluate the program. In

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

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

4. Static Weaving

In our compilation scheme, aspects are woven statically (Step 5 in Figure 2). Specifically, we present in this section a type inference system which guarantees type safety and, at the same time, weaves the aspects through a type-directed translation. The input to the inference and weaving system is a well-typed EA program, converted from its AspectFun version as described in Section 3. But, to ease the presentation, we often omit the type annotations in an expression in the following discussion. Moreover, we concentrate on advice with only primitive pointcuts, yet our static weaving scheme can be adequately extended to handle composite pointcuts such as `f+cflowbelow(g)`. The readers are referred to [2] for the detailed treatment.

4.1. Type directed weaving

The essential construct of our static weaving scheme is the *advised type*. As briefly mentioned in Section 2, an *advised type*, denoted as ρ, is used to capture function names and their types that may be required for advice resolution. We illustrate this concept with our tracing example given in Section 2. The relevant code snippet is repeated below for ease of presentation.

```
n1@advice around {any} (arg) = ... in
n2@advice around {f} (arg::[Char]) = ... in
f x = x in
h x = f x in
(f 10, f "c", h "d")
```

We focus on the call to function `f` in the body of `h`. The issue of static weaving here is essentially the same as we described in Section 1 for the example of function `sortcat`. If we were to naively infer that the argument `x` to function `f` in the RHS of `h`'s definition is of polymorphic type, we would be tempted to conclude that (1) advice `n1` should be triggered at the call, and (2) advice `n2` should not be triggered as its type-scope is less general than $a \rightarrow a$. As a result, only `n1` would be statically woven to the call to `f`.

Unfortunately, this naive approach would cause inconsistent behavior of `f` at run-time, as only the invocation of `f "c"` will trigger advice `n2`. By contrast, according to the operational semantics of AspectFun, both of the invocations `(f "c")` and `(f "d")` (indirectly called from `(h "d")`) should trigger `n2`. We consider such a naive approach to static weaving as *incoherent* because the two invocations of `f` would exhibit different behaviors (i.e., they would receive different sets of advice) even though they would receive arguments of the same type. More formally, a static weaving scheme is deemed as "coherent" if the static woven program evaluates to the same value as the evaluation of the original program according to its operational semantics.[5]

Our static weaving scheme resolves this problem by inferring an advised type for function `h`. Specifically, function `h` possesses the advised type $\forall a.(\texttt{f} : a \rightarrow a).a \rightarrow a$, in which $(\texttt{f} : a \rightarrow a)$ is called an *advice predicate*. It signifies that *the execution of any application of* `h` *may require triggering of the advice on* `f` *whose type can be instantiated to* $t' \rightarrow t'$*, where* t' *is an instantiation of type variable* a*.* Moreover, function `h` and the invocation `h "d"` will be translated by the weaver into the following form.

```
h df x = df x
<h, {n1}> <f, {n1, n2}> "d"
```

Here function `f` inside the definition of `h` has been turned into an *advice parameter*, `df`, which may be resolved to a woven expression. We use the notation, $\langle _ , \{\ldots\} \rangle$, to denote such woven expressions and refer to them as *chain expressions*. Intuitively, a chain expression denotes composition of advice associated with an underlying function. For instance, $\langle \texttt{f} , \{\texttt{n1}, \texttt{n2}\} \rangle$

[5]This notion of "coherence" is different from the coherence concept defined in qualified types [9] which states that different translations of an expression are semantically equivalent.

denotes chaining of two pieces of advice `n1` and `n2` to the advised function `f`. During the code generation stage, chain expressions will be expanded to ordinary Haskell expressions, as will be shown in Section 4.4.

Note that advised types are used to indicate the existence of some advice *indeterminable* at compile time. If a function contains only applications whose advice is 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 `f` in Example 2 is $\forall a.a \rightarrow a$ since it does not contain any application of advised functions in its definition.

The main set of type inference rules, as described in Figures 7 and 8, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \rho \rightsquigarrow e'$ to denote that expression e has type ρ under type environment Γ and it is translated to e'. We assume that the advice declarations are preprocessed and all the names which appear in any of the pointcuts are recorded in an initial global store A. Note that locally defined functions are not subject to being advised and not listed in A. We also assume the type information of all the functions collected in the previous step of AspectFun to EA conversion is stored in Γ_{base}. The function $|\cdot|$ returns the cardinality of a sequence of objects.

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 the name n of type σ is defined on a set of functions $\bar{x}$. We may drop the $\bowtie \bar{x}$ part if it is irrelevant to our discussion. This type σ is inferred from the body and type scope of the advice described in rules (Adv) and (Adv-an); and it is used to guard advice application in rule (Var-A). When a bound function name is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish it from the non-advised ones so that the former may appear in an advice predicate as in rule (Pred). We also use the notation $:_{(*)}$ to represent a binding which is either $:$ or $:_*$. When there are multiple bindings of the same variable in a typing environment, the newly added one always shadows previous ones.

4.2. Predicating and Releasing

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

$$(\textsc{Var})\quad \frac{x : \forall \bar{a}.\bar{p}.t \rightsquigarrow x \in \Gamma}{\Gamma \vdash x\{\bar{t}\} : [\bar{t}/\bar{a}]\bar{p}.t \rightsquigarrow x\{\bar{t}\}} \qquad (\textsc{Var-P})\quad \frac{x\{\bar{t}\} : t \rightsquigarrow dx \in \Gamma}{\Gamma \vdash x\{\bar{t}\} : t \rightsquigarrow dx}$$

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

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

$$(\textsc{Let})\quad \frac{\Gamma \vdash e_1 : \rho \rightsquigarrow e_1' \quad \Gamma.f : \forall \bar{a}.\rho \rightsquigarrow f \vdash e_2 : t \rightsquigarrow e_2'}{\Gamma \vdash \mathsf{let}\ f = \Lambda \bar{a}.e_1\ \mathsf{in}\ e_2 : t \rightsquigarrow \mathsf{let}\ f = \Lambda \bar{a}.e_1'\ \mathsf{in}\ e_2'}$$

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

Figure 7: Static Typing and Weaving Rules for Expressions

Definition 1. *Given a function f and its instantiated type $t_1 \to t_2$ under a typing environment Γ, the predicate $wv^{\Gamma}(f : t_1 \to t_2)$ holds iff the following implication holds:*

$$((\forall n.n :_{(*)} \forall \bar{a}.\bar{p}.t_1' \to t_2' \bowtie \bar{f}) \in \Gamma \wedge f \in \bar{f} \wedge t_1 \sim t_1') \Rightarrow (t_1' \to t_2' \trianglerighteq t_1 \to t_2).$$

where $t_1 \sim t_2$ means that t_1 and t_2 are unifiable.

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

$$(\textsc{Global})\quad \frac{\Gamma \vdash e : \rho \rightsquigarrow e' \quad \Gamma.id :_{(*)} \forall \bar{a}.\rho \rightsquigarrow id \vdash \pi : t \rightsquigarrow \pi'}{\Gamma \vdash \ id = \Lambda \bar{a}.e \ \mathsf{in}\ \pi : t \rightsquigarrow \ id = \Lambda \bar{a}.e' \ \mathsf{in}\ \pi'}$$

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

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

Figure 8: Static Typing and Weaving Rules for Declarations

are freshly instantiated to avoid name capturing. In this paper, we sometimes omit the typing environment part of the weavable constraint when it is clear from the context.

There are three rules for variable lookups. Rule (Var) is standard. Complementarily, in the case that variable x is advised ($x \in A$), there are two rules, (Var-A) and (Var-P), for handling it, depending on type context underlying the occurrence of x. Essentially, when the weavable condition holds, rule (Var-A) applies; otherwise, rule (Var-P) does. The details are as follows.

Rule (Var-A) will create a fresh instance t' of the type scheme bound to x in the environment. Then we check weavable condition of $(x : t')$. If the check succeeds (*i.e.*, x's input type is no more general or equivalent to those of the advice with unifiable types), x will be chained with the translated forms of all the advice defined on it, having equivalent or more general types than x has (the selection is done by $[n_i | t_i \ \trianglerighteq\ t']$). We coerce all these pieces of selected advice to have non-advised type during their translation $\Gamma \vdash n_i : t' \rightsquigarrow e_i$. This ensures correct weaving of advice advising the bodies of the selected advice. The detail will be elaborated in Section 4.6. Finally, the translated expression is *normalized* by bringing all the advice abstractions of x outside

the chain $\langle \ldots \rangle$. This ensures type compatibility between the advised call and its advice.

If the weavable condition check fails, there must exist some advice for x with more specific types, and rule (Var-A) fails to apply. Since $x \in A$ still holds, rule (Pred) can be applied, which adds an advice predicate to a type. (Note that we only allow sensible choices of t constrained by $t_x \trianglerighteq t$.) Correspondingly, its translation yields a lambda abstraction with an *advice parameter*. This advice parameter enables concrete *advice-chained functions* to be passed in at a later stage, called *releasing*, through the application of rule (Rel). Before releasing, any occurrences of an advice parameter in the context will be handled by rule (Var-P), and remains intact.

To sum up, variables that are not advised are handled by rule (Var). Advised variables whose type instantiations satisfy the weavable condition are handled by rule (Var-A). In other situations, variables are turned to an advice parameter by rule (Pred) and handled by rule (Var-P).

We illustrate the application of rules (Pred) and (Rel) by deriving the type and the woven code for the program shown in Example 2. We use C as an abbreviation for $Char$. During the derivation of the definition of h, we have:

$$\Gamma = \{\ \mathtt{f} :_* \forall a.a \to a \rightsquigarrow \mathtt{f}, \mathtt{n}_1 : \forall a.a \to a \bowtie \mathtt{f}, \mathtt{h} \rightsquigarrow \mathtt{n}_1, \mathtt{n}_2 : \forall b.[C] \to [C] \bowtie \mathtt{f} \rightsquigarrow \mathtt{n}_2\}$$

$$\text{(Pred)}\ \dfrac{\text{(Abs)}\ \dfrac{\text{(App)}\ \dfrac{\text{(Var-P)}\ \dfrac{\mathtt{f}\{t\} : t \to t \rightsquigarrow df \in \Gamma_2}{\Gamma_2 \vdash \mathtt{f}\{t\} : t \to t \rightsquigarrow df} \qquad \text{(Var)}\ \dfrac{x : t \rightsquigarrow x \in \Gamma_2}{\Gamma_2 \vdash x : t \rightsquigarrow x}}{\Gamma_2 = \Gamma_1.x : t \rightsquigarrow x \vdash (\mathtt{f}\{t\}\ x) : t \rightsquigarrow (df\ x)}}{\Gamma_1 = \Gamma, \mathtt{f}\{t\} : t \to t \rightsquigarrow df \vdash \lambda x : t.(\mathtt{f}\{t\}\ x) : t \to t \rightsquigarrow \lambda x.(df\ x)}}{\Gamma \vdash \lambda x : t.(\mathtt{f}\{t\}\ x) : (\mathtt{f}\{t\} : t \to t).t \to t \rightsquigarrow \lambda df.\lambda x.(df\ x)}$$

Next, for the derivation of the first element of the main expression, `h "d"`, we have:

$$\begin{aligned}\Gamma_3 = \{\ & \mathtt{f} :_* \forall a.a \to a \rightsquigarrow \mathtt{f}, \mathtt{n}_1 : \forall a.a \to a \bowtie \mathtt{f}, \mathtt{h} \rightsquigarrow \mathtt{n}_1, \\ & \mathtt{n}_2 : \forall b.[C] \to [C] \bowtie \mathtt{f} \rightsquigarrow \mathtt{n}_2,\ \mathtt{h} :_* \forall a.(\mathtt{f}\{a\} : a \to a).a \to a \rightsquigarrow \mathtt{h}\}\end{aligned}$$

$$\text{(App)}\ \dfrac{\text{(Rel)}\ \dfrac{A \qquad B}{\Gamma_3 \vdash \mathtt{h}\{[C]\} : [C] \to [C] \rightsquigarrow (\langle \mathtt{h}\{[C]\}, \{\mathtt{n}_1\}\rangle)\ \langle \mathtt{f}\{[C]\}, \{\mathtt{n}_1, \mathtt{n}_2\}\rangle} \quad \ldots}{\Gamma_3 \vdash (\mathtt{h}\{[C]\}\ \mathtt{"d"}) : [C] \rightsquigarrow \langle \mathtt{h}\{[C]\}, \{\mathtt{n}_1\}\rangle\ \langle \mathtt{f}\{[C]\}, \{\mathtt{n}_1, \mathtt{n}_2\}\rangle\ \mathtt{"d"}}$$

where

$$A \quad = \quad (\textsc{Var-A}) \quad \frac{\mathtt{h} :_* \forall a.(\mathtt{f}\{a\} : a \to a).a \to a \rightsquigarrow \mathtt{h} \in \Gamma_3 \quad ...}{\Gamma_3 \vdash \mathtt{h}\{[C]\} : (\mathtt{f}\{[C]\} : [C] \to [C]).[C] \to [C] \rightsquigarrow \langle \mathtt{h}\{[C]\}\,, \{\mathtt{n}_1\}\rangle}$$

and

$$B \quad = \quad (\textsc{Var-A}) \quad \frac{\mathtt{f} :_* \forall a.a \to a \rightsquigarrow \mathtt{f} \in \Gamma_3 \quad ...}{\Gamma_3 \vdash \mathtt{f}\{[C]\} : [C] \to [C] \rightsquigarrow \langle \mathtt{f}\{[C]\}\,, \{\mathtt{n}_1, \mathtt{n}_2\}\rangle}$$

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

Rules (Pred) and (Rel) introduce and eliminate advice predicates respectively. Rule (Pred) adds an advice predicate to a type. Correspondingly, its translation yields a lambda abstraction with an advice parameter. At a later stage, rule (Rel) is applied to release (*i.e.*, remove) an advice predicate from a type. Its translation generates a function application with an advised expression as argument.

4.3. Handling Advice

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

Rule (Global) is very similar to rule (Let) with the tiny difference that (Global) will bind *id* with : when it is not in A; and with $:_*$ otherwise. It is a rule shared by both function and non-function declarations.

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

After type inference of the advice, we ensure that the advice's type is more general than or equivalent to all functions' in the pointcut. Note that

the type information of all the functions is stored in Γ_{base}. Then, this advice is added to the environment. It does not appear in the translated program, however, as it is translated into a function awaiting participation in advice chaining. The last step in translating the advice declarations is to turn the keyword `proceed` into an additional parameter, representing the rest of computation (i.e., continuation). This is done by the *addProceed* function shown in Figure 9.

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

$$\begin{aligned} &\mathit{addProceed} && : (e, t) \longrightarrow e \\ &\mathit{addProceed}\ (\lambda \overline{df : t_f}.\lambda x : t_x.e_1, t) && = \lambda \overline{df : t_f}.\lambda \texttt{proceed} : t.\lambda x : t_x.e_1 \end{aligned}$$

Figure 9: Proceed lifting

In passing, we note that these two rules can be merged but it makes the rule rather complicated. Hence we keep them separated. In addition, as all the advice has a function type, attempts to advise a non-function type expression will be rejected by the type system.

4.4. Translating Chain Expressions

The last step of AspectFun compilation is to expand chain-expressions produced after static weaving to standard expressions in AspectFun, which are called *expanded expressions*. It is in fact separated into two steps: *chain expansion* and *typeErase*, as shown in Figure 10. Expansion of chain expressions is defined by an expansion operator $[\![\cdot]\!]$. It is applied compositionally on expressions, with the help of an auxiliary function *proceedApply* to substitute proper function for the `proceed` parameter. Moreover, *proceedApply* also handles expansion of second-order advice. Function *typeErase* simply removes any type annotations from its input expression.

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

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

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

$$
\begin{array}{lll}
\mathit{typeErase} & : & \text{Expanded expression} \longrightarrow \text{Implicit } \mathsf{AspectFun} \\
\mathit{typeErase}(x \ = \ e_1 \ \mathsf{in} \ e_2) & = & x \ = \ \mathit{typeErase}(e_1) \ \mathsf{in} \ \mathit{typeErase}(e_2) \\
\mathit{typeErase}(\mathsf{let} \ x \ = \ e_1 \ \mathsf{in} \ e_2) & = & \mathsf{let} \ x \ = \ \mathit{typeErase}(e_1) \ \mathsf{in} \ \mathit{typeErase}(e_2) \\
\mathit{typeErase}(\lambda x : t.e) & = & \lambda x.\mathit{typeErase}(e) \\
\mathit{typeErase}(e_1 \ e_2) & = & \mathit{typeErase}(e_1) \ \mathit{typeErase}(e_2) \\
\mathit{typeErase}(\Lambda a.e) & = & \mathit{typeErase}(e) \\
\mathit{typeErase}(e\{t\}) & = & \mathit{typeErase}(e) \\
\mathit{typeErase}(x) & = & x \\
\mathit{typeErase}(\texttt{proceed}) & = & \texttt{proceed}
\end{array}
$$

Figure 10: Definition of Chain Expansion

Admittedly, the chain expansion step is rather straightforward. One may suggest that the step should be integrated into the weaving step, thus eliminating the need of generating programs in the intermediate form. However, we argue that a staged translation process with chain expression as an intermediate form opens a wide scope of opportunities for optimizing the translated code. For instance, it is obvious that some advice will never invoke

`proceed`. For such pieces of advice, all subsequent advice chained after any of them is considered dead code and should be eliminated. We can therefore prune such chains by performing *dead-code elimination* analysis on the woven code. We have also presented an optimization of control-flow based pointcuts by taking advantage of the explicit intermediate form [2].

Looking at the compilation of AspectFun program in Example 3 (Section 3) again, the intermediate result produced by static weaving is as follows:

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

After applying chain expansion and type erasing, the final result is the following AspectFun expression:

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

4.5. Advising Recursive Functions

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

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

simply returned when their input parameter is empty[6]. We can capture this pattern using a piece of advice. Here we focus on the `reverse` function to illustrate our scheme. The dummy advice `n1` in the following program is necessary for demonstrating the issue involved.

```
n@advice around {reverse, append, setUnion} (arg) =
     \y -> if (null arg) then y else (proceed arg) y in
n1@advice around {reverse} (arg::[Bool]) = proceed arg in
reverse :: [a]->[a]->[a]
reverse x accum = reverse (tail x) (cons (head x) accum) in
reverse [1,2] []
```

After conducting type inference of advice `n`, `n1` and function `reverse`, we obtain the following result (we omit the irrelevant translation part for the moment). We write t_r as an abbreviation of $[a] \to [a] \to [a]$.

$$\Gamma = \{\ \texttt{n} : \forall ab.[a] \to b \to b, \texttt{n1} : \forall a.[Bool] \to a, \texttt{reverse} :_* \forall a.(reverse : t_r).t_r\}$$

Due to the presence of advice `n1`, the type context inside the body of `reverse` is not specific enough to statically weave advice `n` with the recursive call of `reverse`. Hence we give `reverse` an advised type which has a predicate on `reverse` itself. Subsequently, when typing the main expression, `reverse [1,2] []`, we need to release the predicate (`reverse`: $[Int] \to [Int]$) using the (Rel) rule. However, this will lead to the following infinite releasing process because the advised type has a predicate that is the same as the base type.

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

[6]The second input of `append` can be seen as an accumulator parameter. A similar argument applies to `union`

Our solution is to break the looping of (Rel) applications by devising a different releasing rule for recursive functions which predicate on themselves.

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

Rule (Rel-F) spots the pattern that a function is predicated by itself and stops further releasing of that predicate. This is correct since the releasing of the predicate is expected to generate identical advised expression as the translation of `f`. Therefore, we can use a fixed point combinator to self-apply the translation result recursively. For example, as a result of rule (Rel-F), the main expression in the reverse example above is translated to

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

Note that the sub-expression `\y-> <reverse y,{n}>` is derived by rule (Var-A) from typing `reverse {[Int]}`. The combinator `F` enables `reverse` to carry the advice `n` alongside its recursive invocations.

Moreover, mutually recursive functions can be expressed as a tuple of functions, and handled by extending rule (Rel-F). Specifically, when releasing the predicate of any component of mutual recursive functions, the static weaver will introduce a tuple of mutually recursive fixed point combinators accordingly.

4.6. Advising Advice Bodies

In AspectFun, we can write advice that advises other advice, either directly or indirectly. As advice is named, the so-called second-order advice simply includes names of other advice to advise them. Alternatively, inside the body of an advice definition, there may be calls to other functions that are advised by other advice. As mentioned before, we call such advice nested advice. For example, advice `n3` in the following program is a piece of nested advice, as `n3` calls `f` which is in turn being advised by `n1` and `n2`.

```
n1@advice around {f} (arg::Int) = proceed arg in
n2@advice around {f} (arg) = proceed arg in
f x = x in
n3@advice around {g} (arg) = f arg in
g x = x in
```

```
h x = g x in
h 1
```

As mentioned earlier, to handle nested advice properly, the rules (ADV) and (ADV-AN) make an attempt to translate advice bodies, too. Concretely, when a call to `g` is chained with advice `n3`, the body of `n3` must also be advised. Moreover, the choice of advice must be coherent.

However, just like the translation of function bodies, the local type contexts may not be specific enough to satisfy the weavable condition. We illustrate this with the call of `f` inside advice `n3`. Specifically, at the time when the declaration of `n3` is processed, the body of the advice is translated. Since the current type context is not sufficiently specific, an advised type, $\forall a.(\texttt{f} : a \to a).a \to a$ is given to `n3`.

This in turn affects the translation of function `h`. Recall that when the translation attempts to chain advice using Rule (VAR-A), the judgment $\Gamma \vdash n_i : t' \rightsquigarrow e_i$ in the premise forces the advice to have a non-advised type. This is to ensure that all the advice abstractions are fully released so that chaining can take effect. In the case that this derivation fails, it signifies that the current context is not sufficiently specific for advising some of the calls in this advice's body, and chaining has to be delayed.

Now consider the call to `g` in the body of `h`'s definition. The type context for `g` is $a \to a$, which is proper for weaving its advice `n3`. Consequently, the call to `f` inside the body of `n3` is also of type $a \to a$. However, this type is not sufficiently specific for advising `f`. As a result, we have to give `h` an advised type with `g:a->a` as the predicate. The program is then translated as follows.

```
n1 proceed arg:Int = proceed arg in
n2 proceed arg     = proceed arg in
f  x = x in
n3 df proceed arg  = df arg in
g  x = x in
h  dg x = dg x in
h  <g,{n3 <f,{n1,n2}>}> 1
```

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

Nested advice that applies to the execution of its own body merits further discussion because such advice becomes mutually dependent on the functions

it advises. On the one hand, we can employ this dependency to achieve modular and adaptive code reuse for recursive functions. Recall the `eval` function and its advice `cbn` presented in Example 1 (Section 2). Inside `cbn`, function `eval` is invoked, which in turn will trigger `cbn` to ensure the evaluation strategy is changed to call-by-name completely.

On the other hand, such a piece of advice must be handled with care because it may make the weaver non-terminating as the case of recursive functions. Consider the following program with a list function `f` and two pieces of advice, `n` and `n1`, on `f`.

```
n@advice around {f}(arg) = if null arg then arg else f (tail arg) in
n1@advice around {f}(arg::[Int]) = proceed arg in
f (x:xs) = xs in
f [1,2,3]
```

Here advice `n` on function `f` invokes `f` inside its body, thus forming a cycle between `f` and itself. In other words, function `f` and advice `n` are just like two mutually recursive functions. Hence we can translate the example using the same technique of (Rel-F) rule as follows.

```
n df proceed arg = (if (null arg) then arg
                    else (df (tail arg))) in
n1  proceed arg = (proceed arg) in
f (x:xs) = xs in
((let VF = <f,{n VF, n1}> in VF) [1,2,3]
```

However, if we modify the body of advice `n` by supplying a different type of argument, say `[arg]`, to the call to `f`, then the static weaver will run into an infinite releasing loop when handling the main expression. The reason is obvious: the static weaving of a piece of advice requires releasing of some predicate, $(f : t)$, which in turn, directly or indirectly, calls for releasing of another predicate on identical advisee, f, but with a structurally increasing type, say $[t]$. Such vicious circular advice that crashes the static weaver will also cause the program to loop even when weaving is done at runtime. Therefore, we choose to reject such advice statically by enhancing the rules (Adv) and (Adv-an) with a sanity check that identifies such predicate cycles. Specifically, after translating an advice declaration, for each predicate of the advice, the static weaver will compute the set of predicates that will be released when the underlying advice is woven. If there exist multiple

predicates on the same advisee with structurally increasing types in the set, the weaver will reject the program.

Lastly, we note that, besides the cycles formed from cyclical calls of advice and functions, there is one more kind of cycles that can be created, namely cycles formed through triggering of two or more pieces of advice. This is possible because of the presence of second-order advice. However, we do not see any practical value of having such a circular set of advice and insist on a stratified approach to declaring advice. In this approach, programs with circular set of advice will be spotted by our dependency analysis step and rejected.

4.7. Unresolved Advice Predicates

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

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

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

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

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

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

5. Correctness of Static Weaving

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

5.1. Proof Overview

Given an EA program, $\pi \equiv (ds, e)$, our static weaver converts it into a different form in which advice declarations in ds are turned into function declarations and all expressions in π are woven with applicable advice. Therefore, a woven EA program will be evaluated under a different operational semantics context from that of its source version. Hence the basis for our proof is a general definition of equivalence between EA expressions, e and e^*, under different operational semantics contexts, as denoted by $(\mathcal{E}^*, \mathcal{A}^*); (\mathcal{E}, \mathcal{A}) \vdash e^* \simeq e$[7].

Obviously, our proof does not concern the equivalence relation in general, but only the equivalence of a statically woven expression and its source version under two operational semantics contexts that are related by the static weaver. Thus we must define some kind of consistency between a context, $(\mathcal{E}, \mathcal{A})$, and its woven version, $(\mathcal{E}^*, \mathcal{A}^*)$. In particular, when static weaving is done, the advice store $\mathcal{A}^*$ will be empty. Moreover, the essential information of the static weaver is manifested in the static weaving environment, Γ. Hence we define a consistency relation of two operational semantics contexts with respect to a static weaving environment and denote it by $\mathcal{E}^* \overset{\Gamma}{\propto} (\mathcal{E}, \mathcal{A})$.

Essentially, the above consistency relates the bindings found in $\mathcal{E}^*$ and $(\mathcal{E}, \mathcal{A})$ for every identifier in the domain of Γ. Its definition is built on two other definitions. First, the two operational semantics contexts must "respect" the type bindings maintained by the static weaving environment in a specific way. We define such respect relations and denote them by $(\mathcal{E}, \mathcal{A}) \propto \Gamma$ and $\mathcal{E}^* \propto \Gamma$. Besides types, we also need a "consistency" relation at the expressions level. Specifically, we define the consistency of expression relation

[7] As a convention, we put a superscript star to an expression, an environment and an advice store to indicate that they are derived after static weaving.

between a woven EA expression, e^*, and its source expression, e, under type-consistent contexts, and denote it by $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e^* \overset{\Gamma}{\infty} e$.

Based on these definitions, the correctness result is derived from two theorems. First, we prove that, under consistent contexts and the static weaving environment derived from an EA program, any expressions produced by the static weaver will be consistent with their corresponding source expressions. Then we prove that the static weaver maintains the consistency of contexts when processing every form of top-level declarations, thus leading to the equivalence, $\simeq$, between the chain-expanded woven program and the original program.

5.2. Proof Structure

We begin with the definition of an equivalence between two EA expressions of the same type. It is mutually dependent on the equivalence between two values of EA. Moreover, since our ultimate goal is to relate a woven EA expression and its source version, and the woven expression will be evaluated in a different context from its source version, we define the equivalence relation with respect to two operational semantic contexts.

Definition 2 ($\simeq$)**.** *Let e^* and e be two EA expressions with type σ. We define an equivalence relationship between the expressions under the two pairs of operational semantics context, $(\mathcal{E}^*, \mathcal{A}^*)$, and $(\mathcal{E}, \mathcal{A})$, written as*

$$(\mathcal{E}^*, \mathcal{A}^*); (\mathcal{E}, \mathcal{A}) \vdash e^* \simeq e : \sigma$$

if

$$\mathcal{E}^*; \mathcal{A}^* \vdash e^* \Downarrow v^* \text{ iff } \mathcal{E}; \mathcal{A} \vdash e \Downarrow v \text{ and } \mathcal{A}^*; \mathcal{A} \vdash v^* \cong v : \sigma$$

where $\mathcal{A}^*; \mathcal{A} \vdash v^* \cong v : \sigma$ is defined by:

$$\begin{array}{rcl}
\mathcal{A}^*; \mathcal{A} \vdash c & \cong & c : \sigma \\
\mathcal{A}^*; \mathcal{A} \vdash (\!|\Lambda b^*.\ e_1^*, \mathcal{E}_1^*|\!) & \cong & (\!|\Lambda b.\ e_1, \mathcal{E}_1|\!) : \forall a.\sigma_1 \\
 & \text{iff} & \forall t, (\mathcal{E}_1^*, \mathcal{A}^*); (\mathcal{E}_1, \mathcal{A}) \vdash [t/b^*]e_1^* \simeq [t/b]e_1 : \sigma_1 \\
\mathcal{A}^*; \mathcal{A} \vdash (\!|e_1^*, \mathcal{E}_1^*|\!) & \cong & (\!|e_1, \mathcal{E}_1|\!) : t_1 \to t_2 \\
 & & \text{where } e_1^* \text{ and } e_1 \text{ are both lambda expressions} \\
 & \text{iff} & (\mathcal{E}_2, \mathcal{A}^*); (\mathcal{E}_3, \mathcal{A}) \vdash e_2 \simeq e_3 : t_1 \text{ implies} \\
 & & (\mathcal{E}_2, \mathcal{A}^*); (\mathcal{E}_3, \mathcal{A}) \vdash (\!|e_1^*, \mathcal{E}_1^*|\!)\ e_2 \simeq (\!|e_1, \mathcal{E}_1|\!)\ e_3 : t_2
\end{array}$$

We shall omit the type scheme σ when it is obvious from the context.

Note that we can also extend the above equivalence to relate two *open* EA programs since all top-level declarations of an EA program will be turned into a thunk and put into the environment or advice store for evaluating the main expression.

As an example of the equivalence relation, consider the definition of `g` in Example 3 (Section 3.2): `g x = (f x, f(x, x), f [x])`. In particular, we focus on the part in which function `f` is applied to a list argument `[x]`, namely `f {[a]} [x]` in the explicitly typed version. As described in Section 4.4, after static weaving, this occurrence of function `f` is woven with advice `nscope` as `<f {[a]}, {nscope {a}}>` and is subsequently expanded to `nscope {a} (f {[a]})`. It is easy to show that

$$(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash \texttt{nscope}\ \{\mathsf{a}\}\ (\texttt{f}\ \{[\mathsf{a}]\}) \simeq \texttt{f}\ \{[\mathsf{a}]\}$$

where $\mathcal{E}^*$ is the operational semantics environment with static weaving done while $(\mathcal{E}, \mathcal{A})$ is the original operational semantics context.

First, the left-hand side expression, (`nscope {a} (f {[a]})`), will be evaluated to a closure of the lambda expression `\arg:[a].proceed (tail {a} arg)` and an environment in which `proceed` is bound to the value of `f {[a]}`. On the other hand, when the right-hand side expression, `f {[a]}` is evaluated and applied to `[x]`, by the definition of *Trigger*, the advice `nscope` will be triggered and woven to `f {[a]}`, resulting in the same closure. Hence, by the definition of $\cong$, the two expressions are equivalent.

Next, we give a definition which ensures that the type bindings in a static weaving environment are consistent with those of the expressions kept by an operational semantics context. Note that, as the static weaver will eventually convert all advice declarations into normal function declarations, the woven expression will be evaluated in an operational semantics context with an empty advice store. Therefore, we provide two definitions for specifying consistency of type bindings: one for the general operation semantics contexts and the other for the specialized contexts in which the advice store is empty, thus relating only the operational semantics environment. Both are specified in terms of the three forms of bindings which may occur in a static weaving environment. Essentially, the general definition of consistency states the condition before performing static weaving while the specialized one for the case after static weaving is done. For brevity, we refer to both definitions of consistency as *respect of environment*.

Definition 3 (Respect of Environment)**.** *An operational semantics environment $\mathcal{E}$ and an advice store $\mathcal{A}$ are said to respect a static weaving environment Γ, written as $(\mathcal{E}, \mathcal{A}) \propto \Gamma$, if the domains of $\mathcal{E}$ and $\mathcal{A}$ are disjoint and the union of them are as large as the domain of Γ, and for every x in the domain of Γ,*

1. *if $x : \forall \bar{a}.\rho \rightsquigarrow x \in \Gamma$ then $[x \mapsto (\!| e_x, \mathcal{E}' |\!)] \in \mathcal{E}$, and $\Gamma' \vdash e_x\{\bar{a}\} : \rho$ for any Γ' satisfying $(\mathcal{E}', \mathcal{A}) \propto \Gamma'$.*
2. *if $x\{\bar{t}\} : \rho \rightsquigarrow dx \in \Gamma$ then $[x \mapsto (\!| e_x, \mathcal{E}' |\!)] \in \mathcal{E}$, and $\Gamma' \vdash e_x : \rho$ for any Γ' satisfying $(\mathcal{E}', \mathcal{A}) \propto \Gamma'$.*
3. *if $x : \forall \bar{a}.\bar{p}.t_y \to t_x \bowtie f \rightsquigarrow x \in \Gamma$ then $(x : \forall \bar{a}.t_y \to t_x, \overline{pc}, t_y, (\!| e_x, \mathcal{E}' |\!)) \in \mathcal{A}$, and $\Gamma' \vdash e_x\{\bar{a}\} : \bar{p}.t_y \to t_x$ for any Γ' satisfying $(\mathcal{E}', \mathcal{A}) \propto \Gamma'$.*

The specialized version of respect of environment relation holds without the advice store. Specifically, an operational semantics environment $\mathcal{E}^*$ is said to respect a static weaving environment Γ, written as $\mathcal{E}^* \propto \Gamma$, if the domain of $\mathcal{E}^*$ is as large as that of Γ and for every x in the domain of Γ,

1. *if $x : \forall \bar{a}.\rho \rightsquigarrow x \in \Gamma$ then $[x \mapsto (\!| e^*_x, \mathcal{E}' |\!)] \in \mathcal{E}^*$, and $\Gamma' \vdash e^*_x\{\bar{a}\} : \rho$ for any Γ' satisfying $\mathcal{E}' \propto \Gamma'$.*
2. *if $x\{\bar{t}\} : \rho \rightsquigarrow dx \in \Gamma$ then $[dx \mapsto (\!| e^*_x, \mathcal{E}' |\!)] \in \mathcal{E}^*$, and $\Gamma' \vdash e^*_x : \rho$ for any Γ' satisfying $\mathcal{E}' \propto \Gamma'$.*
3. *if $x : \forall \bar{a}.\rho \bowtie f \rightsquigarrow x \in \Gamma$ then $[x \mapsto (\!| e^*_x, \mathcal{E}' |\!)] \in \mathcal{E}^*$, and $\Gamma' \vdash e^*_x\{\bar{a}\} : \rho$ for any Γ' satisfying $\mathcal{E}' \propto \Gamma'$.*

Besides consistency of type bindings, we need to define the consistency of binding definitions common in two operational semantics contexts. Yet, as advice predicates may appear in the binding definitions produced by static weaving, we cannot apply the $\simeq$ directly to relate them to those in the operational semantics context of the source program. Hence, we need to provide a conditional form of equivalence which matches an EA expression with advice predicates to a pure EA expression in a way that is compliant with the $\simeq$ relation and satisfies the underlying advice predicates. First, we notice that the predicates created during static weaving can be realized at run-time through functions – and their associated advice – of appropriate types. This is captured by the notion of *feasibility*.

Definition 4 (Feasibility to predicates)**.** *Given $\Gamma, \mathcal{E}^*, \mathcal{E}$, and $\mathcal{A}$ with $\mathcal{E}^* \propto \Gamma$ and $(\mathcal{E}, \mathcal{A}) \propto \Gamma$, an EA expression e^* is said to be* feasible *to a predicate*

$g\{\bar{t}\} : t_g$, *written as* $(\!| e^*, \mathcal{E}^* |\!) \Leftarrow g\{\bar{t}\} : t_g$, *if* $wv(g : t_g)$ *and* $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash e^* \simeq g\{\bar{t}\} : t_g$.

As an example of predicate feasibility, consider the definition of `h` in Example 3 (Section 3.2): `h x = g [x]`. According to the static weaving described in Section 4.4, function `g` is typed with a predicate `f:a->a`. When it is applied to an argument of type `[b]` inside `h`, we get a more instantiated predicate `f:[b]->[b]`. As the type scope of the advice `nscope` for `f` matches the application context, the condition $wv(\mathtt{f} : [\mathtt{b}] \rightarrow [\mathtt{b}])$ holds. Hence, in this context, `f` can be statically woven with advice `nscope`. Besides, as stated before, $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash (\mathtt{nscope}\ \{\mathtt{b}\}\ (\mathtt{f}\ \{[\mathtt{b}]\})) \simeq \mathtt{f}\{[\mathtt{b}]\}$, and thus $(\!|\mathtt{nscope}\ \{\mathtt{b}\}\ (\mathtt{f}\ \{[\mathtt{b}]\}), \mathcal{E}^*|\!) \Leftarrow \mathtt{f}\{[\mathtt{b}]\} : [\mathtt{b}] \rightarrow [\mathtt{b}]$.

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

Definition 5 (Consistency of Expressions)**.** *Given* $\Gamma, \mathcal{E}^*, \mathcal{E}$, *and* $\mathcal{A}$ *with* $\mathcal{E}^* \propto \Gamma$ *and* $(\mathcal{E}, \mathcal{A}) \propto \Gamma$, *we say that an EA expression* e^* *with type* $\bar{p}.t$ *is consistent with another expression* e *under* Γ, *written as*

$$\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e^* \overset{\Gamma}{\infty} e : \bar{p}.t$$

if given some fresh variables $\overline{dp}$ *with* $|\overline{dp}| = |\bar{p}|$, *for all type substitution* S *and thunks* $\overline{(\!|e^*_p, \mathcal{E}^*|\!)}$ *such that* $\overline{(\!|e^*_p, \mathcal{E}^*|\!)} \Leftarrow S\bar{p}$, *then*

$$(\mathcal{E}^*\overline{[dp \mapsto (\!|e^*_p, \mathcal{E}^*|\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se^*\ \overline{dp}]\!] \simeq Se : St$$

holds.

Now we can define the consistency between the binding definitions of two operational semantics contexts under a specific static weaving environment. It is specified in terms of the "consistency of expressions" relation for each form of bindings which may occur in a static weaving environment derived from an EA program.

Definition 6 (Consistency of Bindings)**.** *Given* $\Gamma, \mathcal{E}^*, \mathcal{E}$, *and* $\mathcal{A}$, *the environment* $\mathcal{E}^*$ *is said to be consistent with* $(\mathcal{E}, \mathcal{A})$ *under* Γ, *written as* $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$, *if*

1. $\mathcal{E}^* \propto \Gamma$ *and* $(\mathcal{E}, \mathcal{A}) \propto \Gamma$,
2. *lambda expressions of the closures in* $\mathcal{E}^*$ *have no labels,*
3. *for every* x *in the domain of* Γ,
   (a) *if* $x : \forall \bar{a}.\bar{p}.t \rightsquigarrow x \in \Gamma$ *then*

$$\mathcal{E}_1^*; (\mathcal{E}_1, \mathcal{A}) \vdash e_x^* \overset{\Gamma}{\infty} \lfloor e_x \rfloor : \bar{p}.t$$

   where $[x \mapsto (\!| \Lambda \bar{a}.e_x^*, \mathcal{E}_1^* |\!)] \in \mathcal{E}^*$ *and* $[x \mapsto (\!| \Lambda \bar{a}.e_x, \mathcal{E}_1 |\!)] \in \mathcal{E}$.
   (b) *if* $x\{\bar{t}\} : t \rightsquigarrow dx \in \Gamma$ *then*

$$\mathcal{E}_1^*; (\mathcal{E}_1, \mathcal{A}) \vdash e_{dx}^* \overset{\Gamma}{\infty} e_x\{\bar{t}\} : t$$

   where $[dx \mapsto (\!| e_{dx}^*, \mathcal{E}_1^* |\!)] \in \mathcal{E}^*$ *and* $[x \mapsto (\!| e_x, \mathcal{E}_1 |\!)] \in \mathcal{E}$.
   (c) *if* $x : \forall \bar{a}.\bar{p}.t_y \to t_x \bowtie f \rightsquigarrow x \in \Gamma$ *then* $(x : \forall \bar{a}.t_y \to t_x, \overline{pc}, t_y, (\!| \Lambda \bar{a}.e_x, \mathcal{E}_1 |\!)) \in \mathcal{A}$, $[x \mapsto (\!| \Lambda \bar{a}.e_x^*, \mathcal{E}_1^* |\!)] \in \mathcal{E}^*$, *and*

$$\mathcal{E}_1^*; (\mathcal{E}_1.\texttt{proceed} = e_p, \mathcal{A}) \vdash \lambda \overline{dp}.(e_x^* \ \overline{dp} \ e_p^*) \overset{\Gamma}{\infty} \lfloor e_x \rfloor : \bar{p}.t_y \to t_x$$

   for all e_p^* *and* e_p *such that* $(\mathcal{E}_1^*; \emptyset); (\mathcal{E}_1, \mathcal{A}) \vdash e_p^* \simeq e_p$.

where $\lfloor \cdot \rfloor$ *removes the label from a lambda function. Specifically,* $\lfloor \lambda^l x : t_x.\ e \rfloor = \lambda x : t_x.\ e$. *Note that the binding of* `proceed` *can be seen as a special case of that of an ordinary variable and handled by 3(a).*

Based on the above definitions, we can proceed to develop the intermediate results that will lead the the correctness of static weaving.

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

Definition 7 (AdviceName and AdviceSet).

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

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

Lemma 1 (Advice Selection). *If* $(\mathcal{E}, \mathcal{A}) \propto \Gamma$, *and*

$$\Gamma \vdash f\{\bar{t}\} : \bar{p}.t_1 \rightarrow t_2 \rightsquigarrow \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle$$

then for any type substitutions S *we have* $\mathit{AdviceSet}(\lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle) = \mathit{Names}(\mathit{Choose}(f, St_1))$ *where* $\mathit{Names}(s) = \{n \mid (n : \sigma_n, \cdots) \in s\}$.

The second lemma shows that the chain expression assembled by (Var-A) rule is consistent with the source expression under the same program context.

Lemma 2 (Consistency of Chain Expressions). *Suppose that* $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$. *If*

1. $\Gamma \vdash f\{\bar{t}\} : \bar{p}.t \rightsquigarrow \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_0^*, e_1^*, \cdots, e_n^*\}\rangle$, $fv(\bar{p}) \subseteq fv(t)$, *and*
2. *for* $i = 0, 1, \cdots, n$, *let* $(\mathit{AdviceName}(e_i^*) : \sigma_i, \overline{pc}_i, (\!| e_i, \mathcal{E}_i |\!)) \in \mathcal{A}$,

   $$\mathcal{E}^*; (\mathcal{E}_i[\mathtt{proceed} \mapsto e_{pr}], \mathcal{A}) \vdash \mathit{proceedApply}(e_i^*, e_{pr}^*) \overset{\Gamma}{\infty} e_i\{\mathit{AdviceUnifiers}(e_i^*, t)\} : t,$$

   holds for all $(\mathcal{E}^*, \emptyset); (\mathcal{E}_i, \mathcal{A}) \vdash e_{pr}^* \simeq e_{pr}$,

then

$$\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle \overset{\Gamma}{\infty} f\{\bar{t}\} : \bar{p}.t$$

Given the above lemmas, we prove the following theorem which states that, under consistent contexts and the static weaving environment derived from an EA program, any expressions produced by the static weaver will be consistent with their corresponding source expressions.

Theorem 1 (Soundness of Expression Weaving). *If* $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$, $\Gamma \vdash e : \bar{p}.t \rightsquigarrow e^*$, *and* $fv(\bar{p}) \subseteq fv(t)$, *then* $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e^* \overset{\Gamma}{\infty} e : \bar{p}.t$

Finally, we prove that the static weaver maintains the consistency of contexts when processing all forms of top-level bindings of an EA program. Thus, combined with the above theorem, we establish the correctness of our static weaving scheme, as the following theorem shows.

Theorem 2 (Soundness of Static Weaving)**.** *Let π_0 be an EA program. If $\emptyset \vdash \pi_0 : t \rightsquigarrow \pi_0^*$, then*

$$\vdash [\![\pi_0^*]\!] \simeq \pi_0 : t.$$

6. Related Work

6.1. Aspect-Oriented Languages

Two works closely related to ours are AspectML [5, 4] and Aspectual Caml [15]. Both works have made many significant results in supporting polymorphic pointcuts and advice in strongly typed functional languages such as ML. While these works have introduced some expressive aspect mechanisms into the underlying functional languages, they have not successfully reconciled coherent and static weaving – two essential features for an aspect-oriented functional language we investigated in this paper.

AspectML [5, 4] advocates first-class join points for constructing generic aspect libraries. In order to support non-parametric polymorphic advice, AspectML includes case-advice which is similar to our type-scoped advice. Its type system is a conservative extension to the Hindley-Milner type inference algorithm with a form of local type inference based on some required annotations. During execution, advice is looked up through labels and *runtime type analysis* is performed to handle the matching of type-scoped pointcuts. This completely dynamic mechanism gives additional expressiveness by allowing run-time advice introduction. However, many optimization opportunities are lost as advice triggering information is not present during compilation. Lastly, advice is anonymous in AspectML and apparently not intended to be the targets of advising, *i.e.* no second-order advice.

Aspectual Caml [15], on the other hand, supports static typing and weaving. In particular, type inference on advice is carried out without consulting the types of the functions designated by the pointcuts. In addition to supporting polymorphic pointcuts, Aspectual Caml also supports monomorphic pointcuts which, when associated with a piece of advice, enable the user to express type-scoped advice. Static weaving is achieved by traversing type-annotated base program ASTs to insert advices at matched joint points.

The type of the applied advice must be more general than those of the joint points, through which type safety is guaranteed. This design has the advantage of clean separate compilation as aspects can be compiled completely independently from the base program. In our case, we value correctness and understandability of program more than the ease of separate compilation.

Aspectual Caml's lexical approach also makes it easy to advise anonymous functions. However, for polymorphic functions invoked indirectly through aliases or functional arguments, this approach cannot achieve coherent weaving results.

Formal specification of the main features of aspect-oriented languages is first given in [19]. There, a denotational semantics is developed for a miniature first-order procedural language, which is intended as a baseline semantics for correctness measurements. Some facets of our operational semantics for AspectFun follow the spirit of their work. But there are also many differences. Notably, in the dynamic context, their semantics differs from ours by having variables in advice always bound by the pointcuts. Consequently, when there are multiple pieces of applicable advice, the argument passed to `proceed` is ignored. This behavior can "lead to an intriguing discontinuity" [19]; this "means that multiple *around* advice interact in a somewhat surprising way" [19]. Moreover, in the static context, their semantics is untyped, whereas static safety and type-scoped advice are at the heart of our design.

6.2. Type-Directed Programming

Our weaving translation was originally inspired by the dictionary translation of Haskell type classes [18]. A number of subsequent applications of type classes [14, 10] also share some similarities. However, the issues discussed in this paper are unique, which make our translation substantially different from the others. Our research does not focus on designing complicated source-level type system that harnesses type-directed programming. Rather, we introduce advised types only to facilitate static weaving, and these types are not visible to users. This design follows the obliviousness principle of AOP, which dictates that aspect introduction should not cause changes to the base program's signature. At the same time, some reasoning properties, such as parametricity [16], that have been carefully preserved by pure functional languages are threatened. We leave it to future work for possible reconciliation of the two.

A more operational difference between AOP and type classes is the multiple triggering of aspects. Type classes are designed for overloading of func-

tions where one instance is always selected for each invocation. In contrast, any number of advice can be attached to the same join point and the execution of them are properly coordinated by the use of `proceed`. As we have seen in Section 2.2.3, this flexibility gives us significant expressiveness in places where type classes struggle.

Our work is not the first to explore type-directed programming with aspects. In [23], Washburn and Weirich demonstrate type-directed programming in AspectML, and the idea of using AOP for extensible generic programing is due to them. The example in Section 2.2.3 demonstrated that the expressivity of dynamic type analysis can be readily harnessed in AspectFun with separate dynamic typing mechanisms encoded in statically typed languages.

7. Conclusion

Static typing, static and coherent weaving are two main foci in this investigation of incorporating aspects into a functional languages with higher-order functions and parametric polymorphism. This paper consolidates our previous research results [21, 20, 2], and makes several significant revisions and extensions along multiple dimensions of our research. Not only do we provide a complete treatment to the core features in AspectFun, we also present an operational semantics of AspectFun. Above all, we provide a formal account of the correctness of our static typing and weaving rules with respect to the operational semantics of AspectFun.

Moreover, we have extended our static weaving scheme to handle complex pointcuts, namely curried pointcuts and control-flow based pointcuts (cflowbelow and cflow). The detail is not covered in this paper. Besides, a monadic compilation scheme for analyzing and optimizing the execution of control-flow based pointcuts has been incorporated in our implementation.

Moving ahead, we will investigate additional optimization techniques and conduct empirical experiments on performance gain. As our implementation automatically converts base program to monadic form, it is particularly promising to investigate use of aspects in capturing side-effecting computation and its monadification implementation in AspectFun[1].

On a different front, we plan to explore ways of applying our static weaving system to other language paradigms. In particular, Java 1.5 has been extended with parametric polymorphism by the introduction of *generics*. Yet, as mentioned in [8], the type-erasure semantics of Java prohibits the

use of dynamic type tests to handle type-scoped advices. We speculate our static weaving scheme could be a key to the solution of the problem.

8. Acknowledgment

The authors would like to thank the anonymous referees and Professor Oege de Moore for valuable suggestions on how to improve a previous version of this paper. This research is partially supported by the National University of Singapore under research grant "R-252-000-252-112", and by the National Science Council, Taiwan, R.O.C. under grant number "NSC 97-2221-E-004-001-MY3".

References

[1] Chen, K., Lin, J.-Y., Weng, S.-C., and Khoo, S.-C. 2009. Designing aspects for side-effect localization. In *PEPM '09: Workshop on Partial Evaluation and Program Manipulation* (2009-01-26), G. Puebla and G. Vidal, Eds. ACM Press, 189–198.

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

[3] Cheney, J. and Hinze, R. 2002. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*. ACM Press, 90–104.

[4] Dantas, D. S., Walker, D., Washburn, G., and Weirich, S. 2005. PolyAML: a polymorphic aspect-oriented functional programmming language. In *Proc. of the tenth ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

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

[6] Haskell 98. Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[7] Hinze, R. and Löh, A. 2009. Generic programming in 3D. *Science of Computer Programming 74,* 8 (June), 590–628.

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

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

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

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

[12] Lämmel, R. and Peyton Jones, S. 2003. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices 38,* 3 (Mar.), 26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[13] Lämmel, R. and Peyton Jones, S. 2005. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005).* ACM Press, 204–215.

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

[15] Masuhara, H., Tatsuzawa, H., and Yonezawa, A. 2005. Aspectual Caml: an aspect-oriented functional language. In *Proc. of the tenth ACM SIGPLAN International Conference on Functional Programming.* ACM Press, 320–330.

[16] Wadler, P. 1989. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM, New York, NY, USA, 347–359.

[17] Wadler, P. 1992. The essence of functional programming. In *Proc. of the 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM*. ACM Press, 1–14.

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

[19] Wand, M., Kiczales, G., and Dutchyn, C. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems 26,* 5 (September), 890–910.

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

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

[22] Wang, M. and d. S. Oliveira, B. C. 2009. What does aspect-oriented programming mean for functional programmers? In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP'09)*, P. Jansson, Ed. ACM.

[23] Washburn, G. and Weirich, S. 2006. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*. ACM Press, New York, NY, USA, 33–44.

[24] Weirich, S. 2000. Type-safe cast: (functional pearl). In *Proc. of ICFP'00*. ACM Press, 58–67.

A. Proof of Correctness of Static Weaving

In this appendix, we give in detail the proof of correctness of static weaving. After presenting a few straightforward propositions, we shall prove the key lemmas and theorems stated in Section 5. Note that, as in the main text, e is used to stand for an EA source expression and e^* for the result of static weaving.

We begin by listing down some properties about operational semantics and basic properties about expression equivalence.

Proposition 1. *If we write* $(\mathcal{E}^*, \mathcal{A}^*); (\mathcal{E}, \mathcal{A}) \vdash e^* \simeq e$ *in a different form*

$$(e^*; \mathcal{E}^*, \mathcal{A}^*) \simeq (e; \mathcal{E}, \mathcal{A})$$

then it is an equivalence relation, that is, satisfying reflexivity, symmetry, and transitivity.

Proposition 2. *Given an* EA expression e^* *and an annotated lambda expression* $\lambda^{f:t_f} x : t_x.e$, *if* $(\mathcal{E}^*, \mathcal{A}^*); (\mathcal{E}, \mathcal{A}) \vdash e^* \simeq \mathit{Trigger}((\!|\lambda x : t_x.e, \mathcal{E}_1|\!), f : t_f)$, *then* $(\mathcal{E}^*, \mathcal{A}^*); (\mathcal{E}_1, \mathcal{A}) \vdash e^* \simeq \lambda^{f:t_f} x : t_x.e$, *and vice versa.*

Proposition 3. *If* $(\mathcal{E}, \mathcal{A}) \vdash e \Downarrow v$ *then, for any* $x \in \mathbf{dom}(\mathcal{E})$, *we always have*

$$(\mathcal{E}, \mathcal{A}) \vdash [\mathcal{E}(x)/x]e \Downarrow v$$

Lemma 1 (Advice Selection)**.** *If* $(\mathcal{E}, \mathcal{A}) \propto \Gamma$, *and*

$$\Gamma \vdash f\{\bar{t}\} : \bar{p}.t_1 \to t_2 \rightsquigarrow \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle$$

then for any type substitutions S *we have* $\mathit{AdviceSet}(\lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle) = \mathit{Names}(\mathit{Choose}(f, St_1))$ *where* $\mathit{Names}(s) = \{n \mid (n : \sigma_n, \cdots) \in s\}$.

Proof.

1. We first show that the translation in the lemma is derived via (Var-A) rule. This is proved by induction on the number of predicates in $\bar{p}$ as follows.
   If $\bar{p}$ is empty, then clearly the translation is just an instance of (Var-A) rule due to the presence of advice chain. Suppose that when $\bar{p}$ contains n predicates, it is still an instance of (Var-A). Now assume that there are $n + 1$ predicates in $\bar{p}$. If the outermost predicate is generated by applying the (Pred) rule, then we have the derivation,

$\Gamma.x\{\bar{t'}\} : t \rightsquigarrow x_t \vdash e : \rho \rightsquigarrow e_t^*$ where ρ has n predicates and t is the type instantiating the type scheme of x. By the induction hypothesis, this translation is derived via (VAR-A) rule and $e \equiv x$ is an advised function. This contradicts the condition of the (PRED) rule. Since no other rule produces a translation with the given form of predicated typed and chain expression except (VAR-A), the translation in the lemma is indeed an instance of (VAR-A), on an advised function, $f\{\bar{t}\}$.

2. We then prove the lemma in two steps:
(a) if $\Gamma \vdash f\{\bar{t}\} : \bar{p}.t \rightsquigarrow e_c^*$ and $\Gamma \vdash f\{S\bar{t}\} : S(\bar{p}.t) \rightsquigarrow e_d^*$, then $\mathit{AdviceSet}(e_c^*) = \mathit{AdviceSet}(e_d^*)$.
(b) if $\Gamma \vdash f\{S\bar{t}\} : S(\bar{p}.t_1 \rightarrow t_2) \rightsquigarrow \lambda\overline{dp}.\langle f\{S\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle$ then $\mathit{Names}(\mathit{Choose}(f, St_1)) = \mathit{AdviceSet}(\lambda\overline{dp}.\langle f\{S\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle)$
Combining these two steps yields the set equality in the lemma.

   (a) From the premise of (VAR-A), we must have $\mathit{AdviceSet}(e_c^*) \subseteq \mathit{AdviceSet}(e_d^*)$. Hence it suffices to show that $\mathit{AdviceSet}(e_d^*) \subseteq \mathit{AdviceSet}(e_c^*)$. This is done by contradiction, as follows.
   Assume that there exists an advice binding $n : \forall\bar{b}.\bar{q}.t_n \bowtie f$ such that $t_n \trianglerighteq St$ but $t_n \ntrianglerighteq t$. Let $t_1 \rightarrow t_2 = t$ and $t_k \rightarrow t_{n_k} = t_n$. By $t_n \trianglerighteq St$, we have $t_k \trianglerighteq St_1$, which in turn implies that t_k and t_1 are unifiable. So, by the condition $wv(f : t_1 \rightarrow t_2)$, we have $t_n \trianglerighteq t$. This contradicts the assumption.
   Since no such advice exists, $\mathit{AdviceSet}(e_d^*) \subseteq \mathit{AdviceSet}(e_c^*)$.

   (b) Let $\mathit{Aset} = \mathit{AdviceSet}(\lambda\overline{dp}.\langle f\ \overline{dp}\ , \{\bar{e}^*\}\rangle)$, $\mathit{Cset} = \mathit{Names}(\mathit{Choose}(f, St_1))$. By $(\mathcal{E}, \mathcal{A}) \propto \Gamma$,

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

   Let us consider the advice selection criteria for both Aset and Cset. For advice n_i to be selected in Aset, (VAR-A) requires that $t_i \rightarrow t_{n_i} \trianglerighteq t_1 \rightarrow t_2$. By contrast, Cset requires that $t_i \trianglerighteq t_1$ according to $\mathit{Choose}(f, t_1)$.
   First, it is easy to see that $\mathit{Aset} \subseteq \mathit{Cset}$ since $t_i \rightarrow t_{n_i} \trianglerighteq t_1 \rightarrow t_2$ implies $t_i \trianglerighteq t_1$.
   Second, we show that $\mathit{Cset} \subseteq \mathit{Aset}$ by assuming otherwise and get a contradiction due to $(\mathcal{E}, \mathcal{A}) \propto \Gamma$. If there exists an advice n_k with type $\forall\bar{b}.\bar{p}'.t_k \rightarrow t_{n_k}$ such that $n_k \in \mathit{Cset}$ but not Aset. Then, $t_k \trianglerighteq t_1$ but $S_1 t_{n_k} \ntrianglerighteq S_1 t_2$ where $t_1 = S_1 t_k$. Let $\forall\bar{a}.\bar{p}.t_{f_1} \rightarrow t_{f_2} = \Gamma(f)$,

then $(t_1 \to t_2) = [S\bar{t}/\bar{a}](t_{f_1} \to t_{f_2})$. Consider the kinds of advice binding for n_k.
(i) (Adv): By the condition of (Adv), $t_k \to t_{n_k} \trianglerighteq t_{f_1} \to t_{f_2}$, which in turn implies that $t_{n_k} \trianglerighteq t_2$ since $t_{f_1} \to t_{f_2} \trianglerighteq t_1 \to t_2$. This contradicts the assumption.
(ii) (Adv-An): By the condition of (Adv-An), there exists a substitution S_0 such that

$$\begin{array}{rrcl} & t_k & = & S_0 t_{f_1} \\ \text{and} & t_{n_k} & \trianglerighteq & S_0 t_{f_2} \end{array} \tag{1}$$

Then $t_1 = S_1 S_0 t_{f_1}$, i.e. $S_1 S_0 = [S\bar{t}/\bar{a}]$. Hence by (1),

$$S_1 t_{n_k} \trianglerighteq S_1 S_0 t_{f_2} = [S\bar{t}/\bar{a}]t_{f_2} = t_2$$

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

□

Lemma 2 (Consistency of Chain Expressions)**.** *Suppose that* $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$. *If* $\Gamma \vdash f\{\bar{t}\} : \bar{p}.t \rightsquigarrow \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_0^*, e_1^*, \cdots, e_n^*\}\rangle$, $fv(\bar{p}) \subseteq fv(t)$, *and for* $i = 0, 1, \cdots, n$, $(AdviceName(e_i^*) : \sigma_i, \overline{pc}_i, (\!| e_i, \mathcal{E}_i |\!)) \in \mathcal{A}$,

$$\mathcal{E}^*; (\mathcal{E}_i[\texttt{proceed} \mapsto e_{pr}], \mathcal{A}) \vdash proceedApply(e_i^*, e_{pr}^*) \overset{\Gamma}{\infty} e_i\{AdviceUnifiers(e_i^*, t)\} : t,$$

holds for all $(\mathcal{E}^*, \emptyset); (\mathcal{E}_i, \mathcal{A}) \vdash e_{pr}^* \simeq e_{pr}$, *then*

$$\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash \lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle \overset{\Gamma}{\infty} f\{\bar{t}\} : \bar{p}.t$$

Proof. We prove the lemma according to the requirements stated in Definition 5 (consistency of expressions). Specifically, given S and $\overline{(\!| e_p^*, \mathcal{E}^* |\!)}$ such that $\overline{(\!| e_p^*, \mathcal{E}^* |\!)} \eqcirc S\bar{p}$ as in Definition 5, we need to prove that

$$(\mathcal{E}^*\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}^* |\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![(\lambda\overline{dp}.S\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle)\ \overline{dp}]\!] \simeq S(f\{\bar{t}\}) = f\{S\bar{t}\}$$

We then β-reduce left hand side cancelling out the lambda and application. Writing $\mathcal{E}_0^* = \mathcal{E}^*\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}^* |\!)]}$ and $(\!| \Lambda\bar{a}.\lambda^{f:t_f} x : t_x.\ e_f, \mathcal{E}_1 |\!) = \mathcal{E}(f)$, by Proposition 3, it is equivalent to show that

$$(\mathcal{E}_0^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![S\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle]\!] \simeq (\!| [S\bar{t}/\bar{a}](\lambda^{f:t_f} x : t_x.\ e_f), \mathcal{E}_1 |\!) : St$$

Let $S' = [S\bar{t}/\bar{a}]$. By applying Proposition 2 on the RHS, it suffices to show that

$$(\mathcal{E}_0^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![S(\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle)]\!] \simeq \\ \mathit{Trigger}((\!|\lambda x : S't_x.\ S'e_f, \mathcal{E}_1|\!), f : S't_f)$$

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

$$\mathit{Weave}((\!|\lambda x : S't_x.\ S'e_f, \mathcal{E}_1|\!), S't_f, \mathit{Choose}(f, S't_x))$$

According to Lemma 1, $\mathit{AdviceSet}(\lambda\overline{dp}.\langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle) = \mathit{Names}(\mathit{Choose}(f, S't_x))$. Thus, it suffices to show that

$$(\mathcal{E}_0^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_c^*]\!] \simeq \\ \mathit{Weave}((\!|\lambda x : S't_x.\ S'e_f, \mathcal{E}_1|\!), S't_f, \{\mathcal{A}(n_i) \mid n_i \leftarrow \mathit{AdviceSet}(e_c^*)\}) \quad (2)$$

where $e_c^* \equiv \langle f\{\bar{t}\}\ \overline{dp}\ , \{\bar{e}^*\}\rangle$. Besides, we strengthen the equation by allowing f to be advice and prove (2) by mathematical induction on the length of $\bar{e}^*$.

When f is advice, $(f \cdots, (\!|\lambda x : S't_x.\ S'e_f, \mathcal{E}_1|\!)) \in \mathcal{A}$, e_c^* is defined as $\langle f\{\bar{t}\}\ \overline{dp}\ e_{fp}^*\ , \{\bar{e}^*\}\rangle$, and $\mathcal{E}_1$ in (2) is replaced by $\mathcal{E}_1[\texttt{proceed} \mapsto e_{fp}]$ where e_{fp}^* and e_{fp} are equivalent under $(\mathcal{E}^*, \emptyset)$ and $(\mathcal{E}_1, \mathcal{A})$.

Induction basis for the length: $|\bar{e}^*| = 0$:

Here, $e_c^* \equiv \langle f\{\bar{t}\}\ \overline{dp}\ , \{\}\rangle$. We proceed by reducing both sides of (2).

$$\begin{array}{rcll}
\text{LHS} & \equiv & [\![S\langle f\{\bar{t}\}\ \overline{dp}\ , \{\}\rangle]\!] & \\
& = & Sf\{\bar{t}\}\ \overline{dp} & ; \mathcal{E}_0^*, \emptyset \\
& & \text{by proposition 3 and let } (f \mapsto (\!|\Lambda\bar{a}.\lambda\overline{dp}.e_f^*, \mathcal{E}_1^*|\!)) \in \mathcal{E}^* & \\
& \simeq & [S\bar{t}/\bar{a}]e_f^* & ; \mathcal{E}_1^*\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}^*|\!)]}, \emptyset \\
\text{RHS} & \equiv & \mathit{Weave}((\!|\lambda x : S't_x.\ S'e_f, \mathcal{E}_1|\!), S't_f, \{\}) & \\
& = & (\!|S'\lambda x : t_x.\ e_f, \mathcal{E}_1|\!) & ; \mathcal{E}, \mathcal{A} \\
& \simeq & S'\lambda x : t_x.\ e_f & ; \mathcal{E}_1, \mathcal{A}
\end{array}$$

By $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$, we could expand the respect relationship of binding of f in $\mathcal{E}^*$ and $\mathcal{E}$ and get

$$(\mathcal{E}_1^*\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}^*|\!)]}, \emptyset); (\mathcal{E}_1, \mathcal{A}) \vdash [S\bar{t}/\bar{a}]e_f^* \simeq S'\lambda x : t_x.\ e_f : t$$

Hence (2) holds for function f. For f being advice, all the above remains the same except extra e_{fp} and e_{fp}^* are added, which match the case for advice in definition 6.

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

Suppose that the equivalence statement (2) above holds for all $\bar{e}^*$ with $|\bar{e}^*| < m$. When $|\bar{e}^*| = m$, let $\bar{e}^* = e_1^*, e_2^*, \cdots, e_m^*$.

We reduce the LHS of (2) as follows:

$$\begin{array}{rl} & [\![S\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_1^*, e_2^*, \cdots, e_m^*\}\rangle]\!] \\ = & \mathit{proceedApply}(Se_1^*, S\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_2^*, \cdots, e_m^*\}\rangle) \end{array}$$

Let $(n_1 : \forall\bar{b}.t_n, \overline{pc}, t_y, (\!|\Lambda\bar{b}.\lambda^{n_1:t_n}y : t_y.e_n, \mathcal{E}_n|\!)) \in \mathcal{A}$. We reduce the RHS of (2) according to the definitions of *Weave* (in Figure 6) and get

$$\begin{array}{rcll} [S\overline{t_n}/\bar{b}]t_n & = & S't_f & \\ e_p & = & \mathit{Weave}((\!|\lambda x : S't_x.S'e_f, \mathcal{E}_1|\!), S't_f, \{\mathcal{A}(n_2), \cdots, \mathcal{A}(n_m)\}) & \\ e_a & = & [S\overline{t_n}/\bar{b}]e_n & \\ \text{RHS} & = & \mathit{Trigger}((\!|\lambda y : [S\overline{t_n}/\bar{b}]t_y.e_a, \mathcal{E}_n[\texttt{proceed} \mapsto e_p]|\!), n_1 : [S\overline{t_n}/\bar{b}]t_n) & ;\mathcal{E}, \mathcal{A} \\ & & \text{by proposition 2} & \\ & \simeq & \lambda^{n_1:[S\overline{t_n}/\bar{b}]t_n}y : [S\overline{t_n}/\bar{b}]t_y.e_a & ;\mathcal{E}_n[\texttt{proceed} \mapsto e_p]; \mathcal{A} \\ & \simeq & (\Lambda\bar{b}.\lambda^{n_1:t_n}y : t_y.e_n)\{S\overline{t_n}\} & ;\mathcal{E}_n[\texttt{proceed} \mapsto e_p]; \mathcal{A} \end{array}$$

Thus, we need to show that

$$\begin{array}{l} (\mathcal{E}_0^*, \emptyset); (\mathcal{E}_n[\texttt{proceed} \mapsto \mathit{Weave}((\!|\lambda x : S't_x.S'e_f, \mathcal{E}_1|\!)], S't_f, \{\mathcal{A}(n_2), \cdots, \mathcal{A}(n_m)\}), \mathcal{A}) \\ \quad \vdash \mathit{proceedApply}(Se_1^*, S\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_2^*, \cdots, e_m^*\}\rangle) \simeq (\Lambda\bar{b}.\lambda^{n_1:t_n}y : t_y.e_n)\{S\overline{t_n}\} : S't_f (*) \end{array}$$

By the assumption of the lemma,

$$\mathcal{E}_0^*; (\mathcal{E}_n[\texttt{proceed} \mapsto e_{pr}], \mathcal{A}) \vdash \mathit{proceedApply}(e_1^*, e_{pr}^*) \overset{\Gamma}{\infty} (\Lambda\bar{b}.\lambda^{n_1:t_n}y : t_y.e_n)\{\overline{t_n}\} : t$$

Applying Definition 5 with the same S, we see that (*) holds if

$$\begin{array}{l} (\mathcal{E}_0^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash e_{pr}^* \equiv [\![S\langle f\{\bar{t}\}\ \overline{dp}\ , \{e_2^*, \cdots, e_m^*\}\rangle]\!] \simeq \\ \qquad\qquad e_{pr} \equiv \mathit{Weave}((\!|\lambda x : S't_x.S'e_f, \mathcal{E}_1|\!), S't_f, \{\mathcal{A}(n_2), \cdots, \mathcal{A}(n_m)\}) \end{array}$$

But this is immediate from the induction hypothesis for the length ($|\bar{e}^*| = m - 1$), and the lemma is proven.

□

A special case of Definition 5 occurs when there is no predicates on type. In such a case, we do not have to consider $\overline{e_p}$ in Definition 5, which yields a stronger proposition.

Proposition 4. *If* $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e^* \overset{\Gamma}{\infty} e : \bar{p}.t$ *and* $\bar{p}$ *is empty, then* $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se^*]\!] \simeq Se : St$ *for any type substitution* S.

Theorem 1 (Soundness of Expression Weaving)**.** *If* $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$, $\Gamma \vdash e : \bar{p}.t \rightsquigarrow e^*$, *and* $fv(\bar{p}) \subseteq fv(t)$, *then* $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e^* \overset{\Gamma}{\infty} e : \bar{p}.t$

Proof. We again strengthen the statement by allowing e to be an advice name with type application and prove it by induction on the height (h) of the derivation tree for $\Gamma \vdash e : \bar{p}.t \rightsquigarrow e^*$.

When e is advice $n\{\text{AdviceUnifiers}(e_i^*, t)\}$, respect relation is replaced by $\mathcal{E}^*; (\mathcal{E}_n[\texttt{proceed} \mapsto e_p], \mathcal{A}) \vdash \mathit{proceedApply}(e^*, e_p^*) \infty e_n\{\text{AdviceUnifiers}(e_i^*, t)\} : \bar{p}.t$ for all $(\mathcal{E}^*, \emptyset); (\mathcal{E}_n, \mathcal{A}) \vdash e_p^* \simeq e_p$ where $(n : \sigma_n, \overline{pc}, t_x, (\!|e_n, \mathcal{E}_n|\!)) \in \mathcal{A}$ as in lemma 2.

Induction basis ($h = 1$):

There are only two cases, namely (Var) and (Var-P), and $e \equiv x\{\bar{t}\}$ for some variable, function, or advice x that $x : \sigma \rightsquigarrow e^* \in \Gamma$ or $x\{\bar{t}\} : t \rightsquigarrow dx \in \Gamma$. Both cases are direct from $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$.

Induction step:

Suppose that the respect condition hold for all derivation trees of height less than h. We prove that, for an expression e with a derivation tree, $\Gamma \vdash e : \bar{p}.t \rightsquigarrow e^*$, of height h. Consider the last step of the derivation:

case (Pred) :

We have a derivation of the form:

$$\frac{x :_* \forall \bar{a}.\bar{p}_x.t_x \in \Gamma \quad [\bar{t}/\bar{a}]t_x \trianglerighteq t_1 \qquad \Gamma.x\{\bar{t}\} : t_1 \rightsquigarrow dx \vdash e : \bar{q}.t \rightsquigarrow e_t^*}{\Gamma \vdash e : (x\{\bar{t}\} : t_1).\bar{q}.t \rightsquigarrow \lambda dx.e_t^*}$$

Hence, $\bar{p} \equiv (x\{\bar{t}\} : t_1).\bar{q}$ and $e^* \equiv \lambda dx.e_t^*$. We need to show that

$$\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash \lambda dx.e_t^* \overset{\Gamma}{\infty} e : (x\{\bar{t}\} : t_1).\bar{q}.t$$

According to Definition 5, assuming $\overline{dp}$ are fresh, when given S and $\overline{(\!|e_p^*, \mathcal{E}^*|\!)} \mathrel{\hat{=}} S(x\{\bar{t}\} : t_1).S\bar{q}$, we show that

$$\mathcal{E}^*\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}^*|\!)]}; (\mathcal{E}, \mathcal{A}) \vdash [\![Se_t^* \ \overline{dp}]\!] \simeq Se : St \tag{3}$$

Since $\bar{p} \equiv (x\{\bar{t}\} : t_1).\bar{q}$, we can write $\overline{(\!| e_p^*, \mathcal{E}^* |\!)}$ as $(\!| e_x^*, \mathcal{E}^* |\!).\overline{(\!| e_q^*, \mathcal{E}^* |\!)}$, $dp = dx.\overline{dq}$ such that $(\!| e_x^*, \mathcal{E}^* |\!) \eqsim S(x\{\bar{t}\} : t_1)$.
Then

$$\begin{array}{rcll} \text{LHS of (3)} & = & [\![Se_t^* \ \overline{dp}]\!] & ; \mathcal{E}^*\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}^* |\!)]}, \emptyset \\ & = & [\![Se_t^* \ \overline{dp}]\!] & ; \mathcal{E}^*[dx \mapsto (\!| e_x^*, \mathcal{E}^* |\!)]\overline{[dq \mapsto (\!| e_q^*, \mathcal{E}^* |\!)]}, \emptyset \\ & = & [\![Se_t^* \ \overline{dp}]\!] & ; \mathcal{E}_2^*\overline{[dq \mapsto (\!| e_q^*, \mathcal{E}^* |\!)]}, \emptyset \end{array}$$

where $\mathcal{E}_2^* = \mathcal{E}^*[dx \mapsto (\!| e_x^*, \mathcal{E}^* |\!)]$. By $(\!| e_x^*, \mathcal{E}^* |\!) \eqsim S(x\{\bar{t}\} : t_1)$, $\mathcal{E}_2^*$ is consistent with $(\mathcal{E}, \mathcal{A})$ under $\Gamma.x\{\bar{t}\} : t_1 \rightsquigarrow dx$. Hence we can apply induction on, $\Gamma.x\{\bar{t}\} : t_1 \rightsquigarrow dx \vdash e : \bar{q}.t \rightsquigarrow e_t^*$ to get

$$\begin{array}{rl} & \mathcal{E}_2^*; (\mathcal{E}, \mathcal{A}) \vdash e_t^* \overset{\Gamma.x\{\bar{t}\}:t_1 \rightsquigarrow dx}{\infty} e : \bar{q}.t \\ \Rightarrow & (\mathcal{E}_2^*\overline{[dq \mapsto (\!| e_q^*, \mathcal{E}^* |\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_t^* \ \overline{dp}]\!] \simeq Se : St \\ \Rightarrow & (\mathcal{E}^*\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}^* |\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_t^* \ \overline{dp}]\!] \simeq Se : St \end{array}$$

which shows that (3) holds.

case (Rel) :

We have a derivation of the form:

$$\frac{\Gamma \vdash e : (x\{\bar{t}\} : t_1).\rho \rightsquigarrow e_1^* \qquad \Gamma \vdash x\{\bar{t}\} : t_1 \rightsquigarrow e_2^* \quad x \neq e}{\Gamma \vdash e : \rho \rightsquigarrow e_1^* \ e_2^*}$$

Hence $\bar{p}.t \equiv \rho$ and $e^* \equiv e_1^* \ e_2^*$. By induction on the second sub-derivation, $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e_2^* \overset{\Gamma}{\infty} x\{\bar{t}\} : t_1$, and by proposition 4,

$$(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_2]\!] \simeq x\{S\bar{t}\} : St_1$$

for any type substitution S, that is, $(\!| Se_2^*, \mathcal{E}^* |\!) \eqsim x\{S\bar{t}\} : St_1$.

On the another hand, by induction on the first sub-derivation,

$$\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e_1^* \overset{\Gamma}{\infty} e : (x\{\bar{t}\} : t_1).\rho$$

Substituting $dx.\overline{dp}$, S, and $(\!| Se_2^*, \mathcal{E}^* |\!).\overline{(\!| e_p^*, \mathcal{E}_1^* |\!)} \eqsim (x\{S\bar{t}\} : St_1).S\bar{p}$ into Definition 5, we get

$$(\mathcal{E}^*[dx \mapsto (\!| Se_2^*, \mathcal{E}^* |\!)]\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}_1^* |\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_1^* \ dx \ \overline{dp}]\!] \simeq Se \tag{4}$$

By proposition 3, substituting dx with $(\!| Se_2^*, \mathcal{E}^* |\!)$ turns (4) into

$$(\mathcal{E}^*\overline{[dp \mapsto (\!| e_p^*, \mathcal{E}_1^* |\!)]}, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![Se_1^* \ Se_2^* \ \overline{dp}]\!] \simeq Se \tag{5}$$

which proves $\mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash e_1^* \ e_2^* \overset{\Gamma}{\infty} e$.

case (VAR-A) :

In this case, x must be an advised function or advice. From the premise of the rule, we have $\Gamma \vdash n_i\{\bar{t}\} : t \rightsquigarrow e_i^*$. By the induction hypotheses of these sub-derivations, those e_i such that $(n_i : \sigma_i, \overline{pc}_i, t_{xi}, (\!| e_i, \mathcal{E}_i |\!)) \in \mathcal{A}$ satisfies $\mathcal{E}^*; (\mathcal{E}_i[\texttt{proceed} \mapsto e_p], \mathcal{A}) \vdash \mathit{proceedApply}(e_i^*, e_p^*) \infty e_i\{\bar{t}\} : t$.

Since the type t has no predicate, by proposition 4,

$$\mathcal{E}^*; (\mathcal{E}_i[\texttt{proceed} \mapsto e_p], \mathcal{A}) \vdash \mathit{proceedApply}(e_i^*, e_p^*) \infty e_i\{\bar{t}\} : t$$

We can then apply Lemma 2 for function x to derive the proof.

For the case that x is bound to a piece of advice, it is just the strengthened proposition in the proof of lemma 2, and it is also proved.

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

$$\frac{\Gamma \vdash e_1 : \rho \rightsquigarrow e_1^* \quad \Gamma.f : \forall \bar{a}.\rho \rightsquigarrow f \vdash e_2 : t \rightsquigarrow e_2^*}{\Gamma \vdash \mathsf{let}\ f = \Lambda \bar{a}.e_1\ \mathsf{in}\ e_2 : t \rightsquigarrow \mathsf{let}\ f = \Lambda \bar{a}.e_1^*\ \mathsf{in}\ e_2^*}$$

Here $e^* \equiv \mathsf{let}\ f = \Lambda \bar{a}.e_1^*\ \mathsf{in}\ e_2^*$.

By induction on the first sub-derivation,

$$\mathcal{E}^*; \mathcal{E}, \mathcal{A} \vdash e_1^* \infty e_1 : \rho$$

Thus

$$\mathcal{E}^*\ [f \mapsto (\!| [\![\Lambda \bar{a}.\ e_1^*]\!], \mathcal{E}^* |\!)] \overset{\Gamma.f:\forall \bar{a}.\rho \rightsquigarrow f}{\infty} (\mathcal{E}\ [f \mapsto (\!| \Lambda \bar{a}.\ e_1, \mathcal{E} |\!)], \mathcal{A})$$

By induction on the second sub-derivation with the enlarged environments,

$$\mathcal{E}^*\ [f \mapsto (\!| [\![\Lambda \bar{a}.\ e_1^*]\!], \mathcal{E}^* |\!)]; (\mathcal{E}\ [f \mapsto (\!| \Lambda \bar{a}.\ e_1, \mathcal{E} |\!)], \mathcal{A}) \vdash e_2^* \overset{\Gamma.f:\forall \bar{a}.\rho \rightsquigarrow f}{\infty} e_2 : t$$

Finally applying the operational semantics (OS:APP) rule proves the case.

case (ABS) :

We have a derivation of the form:

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

Induction hypothesis gives that for all $\mathcal{E}_1^* \overset{\Gamma.x:t_1 \rightsquigarrow x}{\infty} (\mathcal{E}_1, \mathcal{A})$, $\mathcal{E}_1^*; (\mathcal{E}_1, \mathcal{A}) \vdash e_b^* \infty e_b : t_2$.

Given any pair of expressions e_x and e_x^* with $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash e_x^* \simeq e_x : t_1$, it's obvious that

$$\mathcal{E}^*[x \mapsto (\!| e_x^*, \mathcal{E}^* |\!)] \overset{\Gamma.x:t_1 \rightsquigarrow x}{\infty} (\mathcal{E}[x \mapsto (\!| e_x, \mathcal{E} |\!)], \mathcal{A})$$

So we can assign the left hand side to $\mathcal{E}_1^*$, the right hand side to $(\mathcal{E}_1, \mathcal{A}_1)$, and apply proposition 4 getting

$$(\mathcal{E}_1^*, \emptyset); (\mathcal{E}_1, \mathcal{A}) \vdash Se_b^* \simeq Se_b : St_2$$

for all type substitution S. Thus

$$\begin{array}{rc} & (\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash S\lambda x : t_1.e_b^* \simeq S\lambda x : t_1.e_b : S(t_1 \to t_2) \\ \Rightarrow & \mathcal{E}^*; (\mathcal{E}, \mathcal{A}) \vdash \lambda x : t_1.e_b^* \overset{\Gamma}{\infty} \lambda x : t_1.e_b : t_1 \to t_2 \end{array}$$

case (App) :

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

□

Theorem 2 (Soundness of Static Weaving)**.** *Let π_0 be an EA program. If $\emptyset \vdash \pi_0 : t \rightsquigarrow \pi_0^*$, then*

$$\vdash [\![\pi_0^*]\!] \simeq \pi_0 : t.$$

Proof. We use a stronger proposition to prove it. Suppose $\mathcal{E}^* \overset{\Gamma}{\infty} (\mathcal{E}, \mathcal{A})$. If $\Gamma \vdash \pi : t \rightsquigarrow \pi^*$ for a sub-program π of π_0, i.e. $\pi_0 \equiv \overline{d_0}.\pi$, then $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![\pi^*]\!] \simeq \pi$. Afterwards, the original result of the theorem can be obtained by assigning $\emptyset$ to Γ, $\mathcal{E}^*$, $\mathcal{E}$, $\mathcal{A}$ and $\overline{d_0}$.

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

Induction basis:

$|\bar{d}| = 0$: we have $\pi \equiv e$. Since the type of a program is restricted to non-predicated type, this case is a direct consequence of Theorem 1 and Proposition 4.

Induction step:

When the proposition holds for π with length$(\bar{d}) = k$, we shall prove it for π_1 with length$(\bar{d}) = k+1$. Let $\pi_1 \equiv d.\pi$. The Induction step to prove is that if $\mathcal{E}_1^* \overset{\Gamma_1}{\infty} (\mathcal{E}_1, \mathcal{A}_1)$ and $\Gamma_1 \vdash \pi_1 : t_1 \rightsquigarrow \pi_1^*$ then $(\mathcal{E}_1^*, \emptyset); (\mathcal{E}_1, \mathcal{A}_1) \vdash [\![\pi_1^*]\!] \simeq \pi_1$. We prove it by a case analysis on d and induction on the derivation for $\Gamma_1 \vdash d.\pi : t_1 \rightsquigarrow \pi_1^*$:

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

$$\frac{\Gamma_1 \vdash e_x : \bar{p}.t_x \rightsquigarrow e_x^* \quad \Gamma_1.x : \forall\bar{a}.\bar{p}.t_x \rightsquigarrow x \vdash \pi : t_1 \rightsquigarrow \pi^*}{\Gamma_1 \vdash \ x = \Lambda\bar{a}.\ e_x \ \mathsf{in}\ \pi : t_1 \rightsquigarrow \ x = \Lambda\bar{a}.\ e_x^* \ \mathsf{in}\ \pi^*}$$

Given $\Gamma_1 \vdash e_x : \bar{p}.t_x \rightsquigarrow e_x^*$, by Theorem 1, $\mathcal{E}_1^*; (\mathcal{E}_1, \mathcal{A}_1) \vdash e_x^* \overset{\Gamma_1}{\infty} e_x : \bar{p}.t_x$. And by $\mathcal{E}_1^* \overset{\Gamma_1}{\infty} (\mathcal{E}_1, \mathcal{A}_1)$ and Definition 6,

$$\begin{aligned}
&\Gamma \equiv \Gamma_1.\ x : \forall\bar{a}.\bar{p}.t_x \rightsquigarrow x \\
&\mathcal{E}^* \equiv \mathcal{E}_1^*\ [x \mapsto (\!|[\![\Lambda\bar{a}.\ e_x^*]\!], \mathcal{E}_1^*|\!)] \quad \overset{\Gamma}{\infty} \quad (\mathcal{E}_1\ [x \mapsto (\!|\Lambda\bar{a}.\ e_x, \mathcal{E}_1|\!)], \mathcal{A}_1) \equiv (\mathcal{E}, \mathcal{A})
\end{aligned}$$

Thus, by the induction hypothesis of the second premise, $\Gamma \vdash \pi : t_1 \rightsquigarrow \pi^*$, we have

$$(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![\pi^*]\!] \simeq \pi : t_1$$

Combining with

$$\begin{array}{lllllllll}
(\mathcal{E}_1^*, \emptyset); & (\mathcal{E}^*, \emptyset) & \vdash & [\![\pi_1^*]\!] & \equiv & x = [\![\Lambda\bar{a}.\ e_x^*]\!] \ \mathsf{in}\ [\![\pi^*]\!] & \simeq & [\![\pi^*]\!] & : t_1 \\
(\mathcal{E}_1, \mathcal{A}_1); & (\mathcal{E}, \mathcal{A}) & \vdash & \pi_1 & \equiv & x = \Lambda\bar{a}.\ e_x \ \mathsf{in}\ \pi & \simeq & \pi & : t_1
\end{array}$$

implies $(\mathcal{E}_1^*, \emptyset); (\mathcal{E}_1, \mathcal{A}_1) \vdash [\![\pi_1^*]\!] \simeq \pi_1 : t_1$.

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

$$\frac{\begin{array}{c}\Gamma_1.\mathit{proceed} : t_n \rightsquigarrow \mathit{proceed} \vdash \lambda x : t_x.e_a : \bar{p}.t_n \rightsquigarrow e_a^* \\ f_i : \forall\bar{b}.t_i \in \Gamma_{base} \quad t_n \trianglerighteq [\bar{t}/\bar{b}]t_i \quad \Gamma_1.n : \forall\bar{a}.\bar{p}.t_n \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t_1 \rightsquigarrow \pi^*\end{array}}{\begin{array}{l}\Gamma_1 \vdash n :: \forall\bar{a}.t_n @\mathsf{advice\ around}\ \{\bar{f}\}\ (x) = e_a \ \mathsf{in}\ \pi : t_1 \\ \qquad\qquad \rightsquigarrow n = \Lambda\bar{a}.\mathit{addProceed}(e_a^*, t_n) \ \mathsf{in}\ \pi^*\end{array}}$$

with t_x being the argument part of t_n.
First, we apply Theorem 1 to the first sub-derivation and get

$$\mathcal{E}_1^*[\mathtt{proceed} \mapsto e_{pr}^*]; (\mathcal{E}_1[\mathtt{proceed} \mapsto e_{pr}], \mathcal{A}_1) \vdash e_a^* \overset{\Gamma_1.\mathtt{proceed}:t_n \rightsquigarrow \mathtt{proceed}}{\infty} \lambda x.e_a : \bar{p}.t_n$$

for any equivalent e_{pr}^* and e_{pr}. This implies that for all $\overline{(\!|e_p^*, \mathcal{E}_1^*|\!)} \eqcirc \bar{p}$ and type substitution $[\bar{t}/\bar{a}]$,

$$\begin{aligned}
&(\mathcal{E}_1^*[\mathtt{proceed} \mapsto e_{pr}^*]\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}_1^*|\!)]}, \emptyset); (\mathcal{E}_1[\mathtt{proceed} \mapsto e_{pr}], \mathcal{A}_1) \vdash \\
&\quad [\bar{t}/\bar{a}][\![e_a^*\ \overline{dp}]\!] \simeq [\bar{t}/\bar{a}]\lambda x.e_a : [\bar{t}/\bar{a}]t_n
\end{aligned} \tag{6}$$

where $e_a^* \equiv \lambda\overline{dx}.e_n^*$. Then we are ready to prove that

$$\Gamma \equiv \Gamma_1.n : \forall\bar{a}.\bar{p}.t_n \bowtie \bar{f} \rightsquigarrow n$$
$$\mathcal{E}^* \equiv \mathcal{E}_1^*.n = (\!|\Lambda\bar{a}.addProceed(e_a^*, t_n), \mathcal{E}_1^*|\!) \overset{\Gamma}{\infty} (\mathcal{E}_1, \mathcal{A}_1.(n : \forall\bar{a}.t_n, \overline{pc}, t_x, (\!|\Lambda\bar{a}.\lambda^{n:t_n}x : t_x.e_a, \mathcal{E}_1|\!))) \equiv (\mathcal{E}, \mathcal{A})$$

This can be shown by proving

$$\mathcal{E}_1^*; (\mathcal{E}_1[\texttt{proceed} \mapsto e_{pr}], \mathcal{A}_1) \vdash \lambda\overline{dp}.(addProceed(e_a^*, t_n)\ \overline{dp}\ e_{pr}^*) \overset{\Gamma_1.\texttt{proceed}:t_n \rightsquigarrow \texttt{proceed}}{\infty} \lfloor\lambda^{n:t_n}x.e_a\rfloor : \bar{p}.t$$

Applying Definition 5 to the above statement:

$$\begin{array}{rcll}
[\bar{t}/\bar{a}]\text{LHS} & = & [\bar{t}/\bar{a}][\![addProceed(e_a^*, t_n)]\!]\ \overline{dp}\ e_{pr}^* & ; \mathcal{E}_1^*\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}_1^*|\!)]}, \emptyset \\
 & \simeq & [\bar{t}/\bar{a}][\![\lambda\overline{dx}.\lambda\texttt{proceed}.e_n^*]\!]\ \overline{dp}\ e_{pr}^* & ; \mathcal{E}_1^*\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}_1^*|\!)]}, \emptyset \\
 & \simeq & [\bar{t}/\bar{a}][\![e_n^*]\!] & ; \mathcal{E}_1^*[\texttt{proceed} \mapsto e_{pr}^*]\overline{[dx \mapsto dp][dp \mapsto (\!|e_p^*, \mathcal{E}_1^*|\!)]}, \emptyset \\
 & \simeq & [\bar{t}/\bar{a}][\![(\lambda\overline{dx}.e_n^*)\ \overline{dp}]\!] & ; \mathcal{E}_1^*[\texttt{proceed} \mapsto e_{pr}^*]\overline{[dp \mapsto (\!|e_p^*, \mathcal{E}_1^*|\!)]}, \emptyset \\
 & & \text{by (6)} & \\
 & \simeq & [\bar{t}/\bar{a}]\lambda x.e_a & ; \mathcal{E}_1.\texttt{proceed} = e_{pr}, \mathcal{A} \\
 & \simeq & [\bar{t}/\bar{a}]\text{RHS} &
\end{array}$$

Thus, by the induction hypothesis, $(\mathcal{E}^*, \emptyset); (\mathcal{E}, \mathcal{A}) \vdash [\![\pi^*]\!] \simeq \pi : t_1$. And

$$\begin{array}{lllllllll}
(\mathcal{E}_1^*, \emptyset); & (\mathcal{E}^*, \emptyset) & \vdash & [\![\pi_1^*]\!] & \equiv & n = [\![\Lambda\bar{a}.\ addProceed(e_a^*, t_n)]\!] \text{ in } [\![\pi^*]\!] & \simeq & [\![\pi^*]\!] & : t_1 \\
(\mathcal{E}_1, \mathcal{A}_1); & (\mathcal{E}, \mathcal{A}) & \vdash & \pi_1 & \equiv & n :: \forall\bar{a}.t_n @\text{advice around } \{\bar{f}\}\ (x) = e_a \text{ in } \pi & \simeq & \pi & : t_1
\end{array}$$

implies $(\mathcal{E}_1^*, \emptyset); (\mathcal{E}_1, \mathcal{A}_1) \vdash [\![\pi_1^*]\!] \simeq \pi_1 : t_1$.

Note that the equivalence between π_1 and π is obtained by the rule (OS:Adv), which delegates the reduction to (OS:Adv-An) with t_x, the argument part of t_n, as the type scope of the new advice tuple.

case (Adv-An) : We have a derivation of the form:

$$\frac{\begin{array}{c}\Gamma.\texttt{proceed} : t_x \to t \vdash \lambda x : t_x.e_a : \bar{p}.t_x \to t \rightsquigarrow e_a^* \\ f_i : \forall\bar{a}.t_i \to t_i' \in \Gamma_{base} \quad S = [\bar{t}/\bar{a}]t_i \trianglerighteq t_x \quad t \trianglerighteq S[\bar{t}/\bar{a}]t_i' \\ \Gamma.n : \forall\bar{a}.\bar{p}.t_x \to t \bowtie \bar{f} \rightsquigarrow n \vdash \pi : t_1 \rightsquigarrow \pi^*\end{array}}{\begin{array}{c}\Gamma \vdash n :: \forall\bar{a}.t_x \to t @\text{advice around } \{\bar{f}\}\ (x :: t_x) = e_a \text{in } \pi : t_1 \\ \rightsquigarrow n = \Lambda\bar{a}.addProceed(e_a^*, t_x \to t) \text{ in } \pi^*\end{array}}$$

The proof is exactly the same as the previous case. The extra premises of the weaving rule only rules out the programs we considered ill-typed and will not change anything in the proof for well-typed programs.

□