Aspectual Caml: an Aspect-Oriented Functional Language

Hideaki Tatsuzawa
Department of Computer Science, University of Tokyo
hideaki@is.s.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org

Akinori Yonezawa
Department of Computer Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

ABSTRACT

We propose an aspect-oriented programming (AOP) language called *Aspectual Caml* based on a strongly-typed functional language Objective Caml. Aspectual Caml offers two AOP mechanisms, namely the pointcut and advice mechanism and the type extension mechanism, which gives similar functionality to the inter-type declarations in AspectJ. Those mechanisms are not simple adaptation of the similar mechanisms in existing AOP languages, but re-designed for common programming styles in functional languages such as type inference, polymorphic types, and curried functions. We implemented a prototype compiler of the language and used the language for separating crosscutting concerns in application programs, including separating a type system from a compiler of a simple language.

1. INTRODUCTION

Aspect-Oriented Programming (AOP)[7, 16] is a programming paradigm for modularizing *crosscutting concerns*, which can not be well modularized with existing module mechanisms. Although AOP would be useful to many programming languages with module mechanisms, it has been mainly studied in the contexts of object-oriented programming languages such as Java[4, 5, 14, 15], C++[18], and Smalltalk[6, 12].

In this paper, we propose an AOP language called *Aspectual Caml* based on a functional language Objective Caml. The goal of development of Aspectual Caml is twofold. First, we aim to enable *practical* AOP for development of functional programs. Since there have been developed large and complicated application programs in functional languages[13, 20], such as compilers, theorem provers[3] and software verification tools[2], AOP features should be useful to modularize crosscutting concerns also in functional languages. Second, we aim to provide Aspectual Caml as a basis of further theoretical studies on AOP features. Strongly-typed functional languages, such as ML and Haskell, offer many powerful language features based on solid theoretical foundations. Aspectual Caml, which incorporates existing AOP language features into a strongly-type functional language, would help theoretical examination of the features.

Aspectual Caml is an AOP extension to Objective Caml, a dialect of the functional language ML. We design its AOP features by adapting the AOP features in AspectJ, including the pointcut and advice mechanism and the inter-type declaration mechanism, for a functional language with polymorphic types and type inference. We also design the AOP features so that they would fit key properties of strongly-typed functional programming including type safety, type inference, and curried functions.

The language is implemented as a translator to Objective Caml by extending the parser and type checker of the Objective Caml compiler.

The rest of the paper is organized as follows. Section 2 introduces the AOP features of Aspectual Caml. Section 3 presents our current implementation. Section 4 shows case studies of modularization of crosscutting concerns in some application programs with Aspectual Caml. Section 5 presents relevant studies. Section 6 concludes the paper.

2. LANGUAGE DESIGN

This section describes the language design of Aspectual Caml. First, we overview problems in introducing AOP features into functional languages and solutions to those problems. Next, we present an example of extending a small program (which is called a *base program* in this paper) with an aspect. We then discuss the design of the AOP features, namely the pointcut and advice mechanism and the type extension mechanism with emphases on the differences from AspectJ.

2.1 Design Issues

Although the AOP features of Aspectual Caml are similar to the ones in AspectJ, the design of those features was not a trivial task. The differences of programming styles between the base languages (*i.e.,* Objective Caml and Java) such as the higher-order functions, variant records, and polymorphic types, require reconsideration of most AOP features.

Below, we briefly discuss some of the notable issues in the design of AOP features in Aspectual Caml, and our proposed solutions:

- ML (including Objective Caml) and Haskell programs

usually omit types in expressions thanks to the type inference system, whereas types are more explicitly written in Java and AspectJ program. Aspectual Caml has a type inference system for pointcut and advice descriptions.

- Strongly typed languages such as ML and Haskell also have the feature of polymorphic types. We found that type inference with polymorphic types does not fit to programmers' intuition. This is coped with two types of pointcuts, namely *polymorphic* and *monomorphic* pointcuts.
- Functional programs often use curried functions to receive more than one parameters. If the semantics of `call` pointcut were merely capture one application to functions, it would be inconvenient to identify second or later applications to curried functions. To cope with this problem, Aspectual Caml offers *curried pointcuts*.
- Although AOP features similar to the inter-type declarations in AspectJ would be useful, they should be carefully designed because functional programs usually represent structured data by using variant record types, whereas object-oriented programs do by using classes. In particular, the inter-type declarations in AspectJ relies on the type compatibility of classes with additional instance variables and methods, which is not guaranteed for the variant record types. The type extension mechanism in Aspectual Caml therefore has limited scope to preserve type compatibility.

2.2 Example: Extending Simple Interpreter

In this section, we will show an example of a simple program with an aspect. The base program is an interpreter of a small language, which merely has numbers, variables, additions and let-terms. The aspect adds a new kind of terms—subtractions—into the language. Since Aspectual Caml is an extension to Objective Caml, the interpreter is written in Objective Caml.

2.2.1 Interpreter

The interpreter definition begins with definitions for variables which are of type `id`, an identifier type:

```
type id = I of string
let get_name (I s) = s
```

A term is of variant record type `t`, which can vary over number (`Num`), variable (`Var`), addition (`Add`), or let (`Let`) terms:

```
type t = Num of int
       | Var of id
       | Add of t * t
       | Let of id * t * t
```

There are a few functions for manipulating environments, whose definitions are omitted here:

```
let extend = (* env -> id -> int -> env *)
let lookup = (* id -> env -> int *)
let empty_env = (* env *)
```

The interpreter `eval` is a recursive function that takes an environment and a term and returns its value:

```
let rec eval env t = match t with
| Num(n) -> n
| Var(id) -> lookup id env
| Add(t1, t2) ->
     let e = eval env in (e t1) + (e t2)
| Let(id, t1, t2)  ->
     eval (extend env id (eval env t1)) t2
```

For example, the following expression represents evaluation of `let x=3+4 in x+x`, which yields `14`.

```
eval empty_env (Let(I("x"), Add(Num(3),Num(4)),
                Add(Var(I("x")),Var(I("x")))))
```

2.2.2 Adding Subtraction to the Simple Language

The code fragment in Figure 1 shows an aspect definition in Aspectual Caml that extends the interpreter to support subtractions. The first line declares the beginning of an aspect named `AddSubtraction`, which spans until keyword `end`. The body of the aspect consists of an extension to the data structure and a modification to the evaluation behavior.

```
aspect AddSubtraction
  type+ t = ... | Sub of t * t
  pointcut evaluation env t = call eval env; t
  advice eval_sub = [around evaluation env t]
    match t with
    Sub(t1, t2) -> (eval env t1) - (eval env t2)
  | _ -> proceed t
end
```

Figure 1: An aspect that adds subtraction to interpreter

The second line is *type extension* that adds an additional constructor `Sub` to type `t` so that extended interpreter can handle subtraction terms. Within `AddSubtraction` aspect, the type `t` has a constructor `Sub` as well as other constructors defined in the base program. Section 2.4 explains this mechanism in detail.

The third line defines a pointcut named `evaluation` that specifies any application of an environment and a term to `eval` function. The pointcut also binds variables `env` and `t` to the parameters of `eval`. This is also an example of *curried pointcut* that can specify applications to curried functions. Section 2.3.2.4 will explain this in detail.

Lines 4–7 are an advice declaration named `eval_sub` that evaluates subtraction terms augmented above. The keyword `around` on the right hand side at the fourth line specifies that the body of the advice runs instead of a function application matching the pointcut `evaluation`. The lines 5–7 are the body of the advice, which subtracts values of two sub-terms when the term is a `Sub` constructor. Otherwise, it lets the original `eval` interpret the term by applying the term to a special variable `proceed`, which is bound to a function that represents the rest of the computation at the function application.

Note that the pointcut and the body of the advice have no type descriptions, which is similar to other function defini-

Table 1: Kinds of Join Points in Aspectual Caml and AspectJ

in Aspectual Caml	in AspectJ
function call	method call
function execution	method execution
construction of a variant	constructor call
pattern matching	field get

tions in Objective Caml. The type system infers appropriate types and guarantees type safety of the program.

2.3 Pointcut and Advice Mechanism

Aspectual Caml offers a pointcut and advice mechanism for modularizing crosscutting program behavior. The following three key elements explains the mechanism:

- *join points* are the points in program execution whose behavior can be augmented or altered by advice declarations.
- *pointcuts* are the means of identifying join points, and
- *advice declarations* are the means of effecting join points.

The design is basically similar to those in AspectJ-like AOP languages. We mainly explain the notable differences below.

2.3.1 Join Points

Similar to AspectJ-like languages, Aspectual Caml employs a dynamic join point model, in which join points are the points in program execution, rather than the points in a program text. There are four kinds of join points in Aspectual Caml, which are listed in Table 1 with their AspectJ counterparts.

Note that the correspondences between Aspectual Caml and AspectJ are rather subjective as functional programs and Java-like object-oriented programs often express similar concepts in different ways. For example, functional programs often use variant records to represent compound data while object-oriented programs use objects. Therefore we place the pattern matching (which takes field values out of a variant record) and field get join points in the same row. There are no field-set-like join points in Aspectual Caml since variant records are immutable[1].

A join point holds properties of the execution, such as the name of the function to be applied to and arguments. The names of functions are those directly appear in program text. For example, evaluation of `let lookup = List.assoc in lookup var env` generates a function call join point whose function name is `lookup`, rather than `List.assoc`. We believe that programmers give meaningful names to functions even if the higher-order functions make renaming of functions quite easy in functional programming.

[1]Many functional programming languages offer *references* for representing mutable data. The operations over references are also the candidates of join points in future version of Aspectual Caml.

Table 2: Summary of Primitive Pointcuts

syntax	matching join points
`call` N P_1 ; ...; P_n	function call
`exec` N P_1 ; ...; P_n	function execution
`new` N`(`P_1`,...,`P_n`)`	construction of a variant
`match` P	pattern matching (before selecting a variant)
`within` N	all join points within a static scope specified N

2.3.2 Pointcuts

A pointcut is a predicate over join points. It tests join points based on the kinds and properties of join points, and binds values in the join point to variables when matches.

2.3.2.1 Primitive Pointcuts

Similar to AspectJ, Aspectual Caml has a sublanguage to describe pointcuts. Table 2 lists the syntax of primitive pointcuts and kinds of join points selected by respective pointcuts. In the table, N denotes a name pattern and P_i denotes a parameter pattern.

A name pattern N is a string of alphabets, numbers, and wildcards followed by a type expression. It matches any function or constructor whose name matches the former part, and whose type matches the latter part. The type expression can be omitted for matching functions of any type.

A parameter pattern P is a pattern that used to describe a formal parameter of a function in Objective Caml[2]. It is either a variable name, or a constructor with parameter patterns, followed by a type expression. It matches any value of the specified type, or any value that is constructed with the specified constructor and the field values that match respective the parameter patterns. Again, the type expression can be omitted. For example, "`x:int`" matches any integer. "`Add(Num(x),Var(y))`" matches any `Add` term whose first and second fields are any `Num` and `Var` terms, respectively. Note that parameter patterns with constructors are basically runtime conditions. This is similar to `args`, `this` and `target` pointcuts in AspectJ which can specify runtime types of parameters.

Pointcut `within(`N`)` matches any join point that is created by an expression appearing in a lexical element (e.g., a function definition) matching N. In order to specify function definitions nested in other function definitions, the pattern N can use a path expression, which is not explained in the paper.

2.3.2.2 Parameter Binding

The parameter patterns in a primitive pointcut also bind parameters to variables. For example, when string `"abc"` is applied to function `lookup` and there is a pointcut `call lookup name`, the pointcut matches the join point and binds

[2]In Objective Caml, it is simply called a "pattern", but we refer it to as a "parameter pattern" for distinguishing from the name patterns.

the string `"abc"` to the variable `name` so that the advice body can access to the parameter values. When a pattern has an underscore character ("_") instead of a variable name, it ignores the parameter value.

2.3.2.3 Combining and Reusing Pointcuts

Aspectual Caml offers various means of combining and reusing pointcuts similar to AspectJ. There are the operators for combining pointcuts, namely `and`, `or`, `not`, and `cflow`. It also supports named pointcuts. For example, the line 3 in Figure 1 names a pointcut expression (`call eval env; t`) `evaluation`,

```
pointcut evaluation env t = call eval env; t
```

which can be used in a similar manner to primitive pointcuts in the subsequent pointcut expressions, like `evaluation env t` at line 4 in the same figure.

2.3.2.4 Pointcuts for Curried Functions

The `call` and `exec` pointcuts also support curried functions. For example, `call eval env; t` matches the second partial application to function `eval`. Therefore, when an expression `eval empty_env (Num 0)` is evaluated, the pointcut matches the application of `(Num 0)` to the function returned by the evaluation of `eval empty_env`. The pointcut matches even when the partially applied function is not immediately applied. As a result, when `let e = eval env in (e t1) + (e t2)` is evaluated, the applications of `t1` and `t2` to `e` match the above call pointcut.

The following definition gives more precise meaning to `call` pointcuts:

- `call` N P_1 matches evaluation of an expression (e_0 e_1) when the expression e_0 matches the name pattern N and the expression e_1 matches the parameter pattern P_1.
- `call` N P_1`; ...;` P_n matches evaluation of an expression (e_0 e_1) when the evaluated value of e_0 is returned from a join point matching to `call` N P_1`; ...;` $P_{(n-1)}$ and the expression e_1 matches the parameter pattern P_N.

Similarly, `exec` pointcuts support curried functions on the callee's side.

Section 3.4 presents how this advice declarations with a curried pointcut can be implemented.

2.3.2.5 Type Inference for Pointcuts

When types are omitted in a pointcut expression, they are automatically inferred from the advice body in which the pointcut is used. This fits with the programming style in Objective Caml, where types can be omitted as much as possible.

For example, the advice `eval_sub` in Figure 1 has no type expressions in the pointcut `evaluation env t`. However, it is inferred from the expressions in the advice body, that the types of the variables `env` and `t` and the return type of the function are the types `env`, `t` and `int`, respectively. As a result, the pointcut, whose definition is `call eval env; t`, matches applications to a function named `eval` and of type `env` $\rightarrow$ `t` $\rightarrow$ `int`.

The type inference gives the most general types to the variables in the pointcuts. In the following advice definition, the system gives fresh type variables α and β to variables `env` and `t`, respectively:

```
advice tracing = [around call eval env; t]
  let result = proceed t in print_int result; result
```

As a result, the pointcut matches any applications to functions whose type is more specific than $\alpha \rightarrow \beta \rightarrow$ `int`. As a result, this advice captures applications to `eval` as well as other `eval` functions that takes two parameters and returns integer values.

2.3.2.6 Polymorphic and Monomorphic Pointcuts

Aspectual Caml provides a mechanism that programmers can make the types in a named pointcut either polymorphic or monomorphic. This is useful when there are more than one advice definition that uses the same named pointcut. When a named pointcut is defined with the keyword `concrete`, it is a *monomorphic pointcut* whose type variables can not be instantiated. Otherwise, it is a *polymorphic pointcut* whose type variables are instantiated when the pointcut is used in an advice definition.

For example, the `evaluation` pointcut in Figure 1 is polymorphic. It matches any function applications `eval` of type $\forall\alpha\beta\gamma.\alpha \rightarrow \beta \rightarrow \gamma$. When `evaluation` used in advice `eval_sub`, the type system instantiates α, β, and γ and then infers the types with respect to the advice body. Therefore, another advice definition that uses `evaluation` with different types do not conflict with the previous advice definition:

```
advice tracing = [before evaluation env t]
  print_string env; print_string t
end
```

This mechanism is quite similar to the let-polymorphism in ML languages.

Although the polymorphic pointcuts are useful to define generalized pointcuts, they are sometimes inconvenient when the programmer wants to specify the same set of join points at any advice that uses the same pointcut. *Monomorphic pointcuts* are useful in such a situation. Consider the following aspect definition that prints messages at the beginning and end of any function application:

```
aspect Logging
  pointcut logged n = call ??$ n
  advice log_entry = [before logged n]
    print_string ("\nenters with "^(string_of_int n))
  advice log_exit = [after logged n]
    print_string "\nexits"
end
```

Since `logged` is a polymorphic pointcut that matches any application to functions of type $\forall\alpha\beta.\alpha \rightarrow \beta$, the first advice matches only functions that take integer values as their parameter, whereas the second matches any function. This is

because the types in the pointcut are inferred at each advice definition.

By declaring `logged` pointcut with the keyword `concrete` and type expression to the variables that are used in the advice:

```
concrete pointcut logged n = call ??$ (n:int)
```

`logged` pointcut becomes monomorphic that matches any application to functions of type `int` $\rightarrow \alpha$. With this pointcut definition, the two advice definitions are guaranteed to advise the same set of join points because the types in the pointcut will not be instantiated further.

2.3.3 Advice

Advice, defined with a pointcut, gives behaviors at, before, or after join points, these timing are decided by timing specifiers `around`, `before`, and `after` respectively, specified by the pointcut. In the body of advice, programmers can use all top-level variables, variables bound by the pointcut, and the special function `proceed` (available only in `around` advice). Since `proceed` means the replaced behavior, it restarts the original execution when it takes an argument.

For preserving type safety, the body expression of `around` advice must have the same type as returning values of specified join points. In addition, that of `before` and `after` advice must have the type `unit`. In the example of subtraction extension, the body of `eval_sub` has the type `int` that is the same type as a result value of `eval`.

2.4 Type Extension Mechanism

The type extension mechanism allows aspects to define extra fields or constructors in variant types in a base program. The former mechanism can be seen as a rough equivalent to the inter-type instance variable declarations in AspectJ.

Despite the simplicity of the mechanism, we believe that it is as crucial as the pointcut and advice mechanism. As you can observe in example programs in AspectJ, not a few crosscutting concerns contain not only behavior (which is implemented by the pointcut advice mechanism) but also data structures (which are implemented by the inter-type declarations).

2.4.1 Defining Extra Constructs

One of the abilities of the type extension mechanism is to define additional constructors to existing variant types. A variant type definition `type+` T `= ... |` C adds constructor C to existing type whose type name is T. In Figure 1, we have already seen an example that adds `Sub` constructor to the type `t`.

The constructors added to a variant type by aspects often make pattern matching non-exhaustive. In other words, a base program that originally defined the variant type usually has functions that process for each variant differently (*e.g.,* `eval` in the simple interpreter). Therefore, an aspect that added a constructor to a variant type would also need to advise such functions so as to process the case for the additional constructor. In the example Figure 1, the advice `eval_sub` processes the constructor `Sub` for the function `eval`, which otherwise reports non-exhaustiveness warnings.

2.4.2 Defining Extra Fields

The type extension mechanism can also allow to define additional fields to constructors of existing variant types. A variant type definition

$$\texttt{type+}\ T_0\ \texttt{=}\ C\ \texttt{of ...}\ \ \texttt{*}\ T_1\{e_1\}\ \texttt{*}\ \cdots\ \texttt{*}\ T_n\{e_n\}$$

adds fields of type $T_1, \ldots, T_n$ to a constructor C of type T_0. The expressions $e_1, \ldots, e_n$ in the curly brackets specify default values to the respective fields.

For example, assume we want to associate a number (*e.g.,* a line number in a source program) to each variable in the simple language of Section 2.2. A solution with the type extension mechanism is to add an integer field to `ident` type by writing the following definition:

```
type+ ident = I of ... * int{0}
```

As the base program originally defines `ident` type as `type ident = I of string`, a value created by the constructor `I` has a pair of string and integer.

Extended fields are available only in the aspects that define the extension. This means that the type of the constructor look differently inside and outside of the aspect:

- Inside the same aspect, the constructor has the extended type. Therefore, `I ("x",1)` is a correct expression in the aspect.
- Outside the aspect, the constructor retains the original type, and yields a value that has the values of default expressions in the extended fields. Therefore, `I "x"` is a correct expression outside the aspect, which yield a value that has `"x"` and `0` in its string and integer fields, respectively.

3. IMPLEMENTATION

We implemented a compiler, or a weaver of Aspectual Caml as a translator to Objective Caml. Many parts of the compiler are implemented by modifying internal data structures and functions in an Objective Caml compiler as the AOP features deeply involve with the type system.

The compiler first parses a given program to build a parse tree. Then the next five steps process the parse tree:

1. infers types in the base function definitions;
2. infers types in the aspect definitions;
3. modifies variant type definitions in the base program by processing type extensions;
4. simplify advice definitions; and
5. inserts applications to advice bodies into matching expressions.

Finally, it generates Objective Caml program by unparsing the modified parse tree.

Below, those five steps are explained by using the example in Section 2.2.

3.1 Type Inference for Base Functions

The types in the base function definitions are inferred by using the internal functions in the original Objective Caml compiler. After the type inference, all variables in the functions are annotated with types (or type schemes):

```
type id = I of string
let (get_name:id->string) = fun (I(s:string)) -> s
type t =        (* omitted *)
let extend =    (* ibid. *)
let lookup =    (* ibid. *)
let empty_env = (* ibid. *)
let rec (eval:env->t->int) =
  fun (env:env) -> fun (t:t) -> match t with
| Num(n:int) -> n
| Var(id:id) -> lookup id env
| Add((t1:t), (t2:t)) ->
    let (e:t)->int = eval env in (e t1) + (e t2)
| Let((id:id), (t1:t), (t2:t))  ->
    eval (extend env id (eval env t1)) t2
```

3.2 Type Inference for Aspects

The types in aspect definitions are inferred in a similar manner to the type inference for the base functions. Notable points are the treatments of polymorphic/monomorphic pointcuts, and scope of the variables.

The type of a pointcut is a type of join points that can match the pointcut and a type environment for the variables in the pointcut. The type of matching join points is decided by the shapes of primitive pointcuts in the pointcut and the types of the variables. The variables bound by the pointcuts have unique type variables otherwise explicitly specified. For polymorphic pointcuts, those type variables are quantified with universal quantifiers that can be instantiated at the advice definitions. Monomorphic pointcuts use the special type variables that can not be instantiated in the later processes.

For example, `evaluation` pointcut in Figure 1 has, type of $\forall\alpha\beta\gamma.\alpha \to \beta \to \gamma$ for the matching join points, and [`env` : β, `t` : γ] for variables.

Note that the type inference of pointcuts does not use the types of function names; *e.g.,* the type of `eval` in the base program. This is because the function names in pointcuts do not necessarily refer to specific functions in the base program, but they rather refer to any function that have matching name.

The type inference of an advice definition is basically similar to the type inference of a function definition, but it takes types of parameters from the types of the pointcut, and gives a type to `proceed` variable that is implicitly available in the advice body. Given an advice definition `advice` a `= [around call` p`]` e where p is a pointcut of join point type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and variable type ρ, the type of e is inferred under the global type environment extended with ρ and [`proceed` : $\alpha_n \to \beta$].

For example, type inference of `eval_sub` advice uses a global type environment extended with [`proceed` : $\beta \to \gamma$, `env` : α, `t` : β], and assigns types as follows:

```
advice eval_sub
 = [around evaluation (env:env) (t:t)]
  (* let proceed:t->int *)
    match t with
      Sub((t1:t), (t2:t)) ->
        (eval env t1) - (eval env t2)
    | _ -> proceed t
```

Note that the types of `eval` and `Sub` are taken from the global type environment, which eventually instantiates the types of other variables including those in the pointcut.

3.3 Reflect Type Modifications in Base Programs

In this phase, type extensions are reflected in the base programs. The definition of types are changed according to the aspects. Additionally, the default values are added to expressions whose fields are extended by the aspects.

3.4 Simplify Advice Definitions

The next step is to transform the advice definitions into simpler ones in order to make the later weaving process easier.

First, it transforms every `before` and `after` advice definition into `around` advice, by simply inserting an application to `proceed` at the beginning or end of the advice body.

Second, it transforms an advice declaration that uses curried pointcuts so that all `call` or `exec` pointcuts takes exactly one parameter. The next is a translated advice definition from `eval_sub` (inferred types are omitted for readability):

```
advice eval_sub = [around call eval env]
  let proceed = proceed env in
    fun t -> match t with
        Sub(t1, t2) -> (eval env t1) - (eval env t2)
      | _ -> proceed t
```

When an environment is applied to `eval`, the transformed advice runs and returns a function that runs the body of the original advice when it takes a term. In other words, `eval` is advised to return a function that runs the original advice body.

Generally, it transforms an advice definition with a curried pointcut by iteratively removing the last parameter in the curried pointcut by using the following rule that transforms an advice definition:

`advice` a `= [around call` f v_1`; ⋯;` v_n`]`
e

into the next one:

`advice` a `= [around call` f v_1`; ⋯;` v_{n-1}`]`
`  let proceed = proceed` v_{n-1} `in`
`    fun` v_n `->` e

There is a subtle problem with this approach when curried pointcuts are used with a disjunctive (`or`) operator, which is left for future research.

3.5 Weave Advice Definitions

The last step is to insert expressions that runs advice bodies at appropriate times in the base functions. It first transforms each advice definition into a function definition. It then walks through all expressions (*i.e., join point shadows*) in the function definitions, and inserts an application to an advice function when it matches the pointcut of the advice.

Given an advice definition, the first step is to simply generate a recursive function that takes `proceed` parameter followed by the parameters to the advice. For example, it generates the following function for `eval_sub` advice (again, types are omitted for readability):

```
let rec eval_sub proceed env =
  let proceed = proceed env in
    fun t -> match t with
        Sub(t1, t2) -> (eval env t1) - (eval env t2)
      | _ -> proceed t
```

The second step is to rewrite the bodies of the base functions[3] so that they call advice functions at appropriate places. By traversing the expressions in the given program, for each expression type of function application, lambda abstraction, constructor application, or pattern matching for structured values, it looks for advice definitions that have the respective kind of primitive pointcuts. When the name pattern of the pointcut matches the name in the expression, and the type of the pointcut is more general than the type of the expression, it replaces the expression with an application to the advice function.

For example, `eval` function in the base program has a sub-expression `(eval env)` where `eval:env->t->int` and `env:env`. This application sub-expression matches the `call` pointcut in `eval_sub` as the types of the join point and the pointcut are the same. In this case, it replaces the expression with (⟨*eval_sub*⟩ `eval env`) where (⟨*eval_sub*⟩ is an expression that references the advice function (explained below).

It is a little tricky to define and reference advice functions due to recursiveness introduced by advice. An advice definition has a global scope; it can advise any execution in any module and it also can use global functions defined in any module. Consequently, advice definitions can introduce recursion into non-recursive functions in the original program. For example, the following code fragment recursively computes factorial numbers by advising the non-recursive function `fact`[1]:

```
let fact n = 1
aspect Fact
  advice realize = [around exec fact n]
    if n=0 then proceed n else n*(fact (n-1))
end
```

In order to allow advice to introduce recursion, we proposed two solutions:

- Define advice functions in a *recursive module*[17] in Objective Caml. As recursive modules allow mutual recursion between functions across modules, this would directly solve the problem.
- Reference advice functions via mutable cells. In this solution, the translated program begins with definitions of mutable cells that hold advice functions. The subsequent function definitions run advice functions by dereferencing from those mutable cells. Finally, after defined advice functions, the program stores the advice functions into the mutable cells.

Although the latter solution is trickier, our current implementation uses it since the recursive modules are not available in official Objective Caml implementations as far as the authors know.

After finished above processes, the compiler generates the following translated code for the example program:

```
(* define mutable cells for advice functions *)
let eval_sub_ref = ref (fun _ -> failwith "")
(* definitions for id, t and env are omitted *)
let rec eval env t = match t with
| Num(n) -> n
| Var(id) -> lookup id env
| Add(t1, t2) ->
    let e = !eval_sub_ref eval env in
    (e t1) + (e t2)
| Let(id, t1, t2)  ->
    !eval_sub_ref
     eval
     (extend env id (!eval_sub_ref eval env t1))
     t2
(* advice function *)
let rec eval_sub proceed env =
  let proceed = proceed env in
  fun t -> match t with
      Sub(t1, t2) ->
        (!eval_sub_ref eval env t1) -
        (!eval_sub_ref eval env t2)
    | _ -> proceed t
(* store advice function into mutable cell *)
let _ = eval_sub_ref := eval_sub
```

Note that all applications to `eval` function, including those in the advice body, are replaced with applications to `!eval_sub_ref eval`. The `eval_sub_ref` is defined at the beginning of the program with a dummy value, and assigned `eval_sub` function at the end of the program.

3.6 Implementation Status

Thus far, we developed a prototype implementation[4] of Aspectual Caml. Although some of the features discussed in the paper are not available yet, it supports essential features for validating our concept, including the type extension, around advice, and most kinds of primitive pointcuts except for wildcarding. In fact, the next section introduces an example that can be compiled by our prototype implementation.

The current implementation has approximately 24000 lines of Objective Caml program, including the parser and type

[3]Precisely, the base functions also include the advice bodies. This enables to advise execution of advice as well as execution of function.

[4]Available at `http://www.yl.is.s.u-tokyo.ac.jp/~hideaki/acaml/` .

inference system that are modified from the ones in the original Objective Caml compiler. Although it would be theoretically possible to directly pass the translated parse tree to the back-end Objective Caml compiler, our compiler generates source-level program by unparsing the parse tree. This is mainly for the ease of development and for debugging.

4. APPLICATION PROGRAMS

Among several small application programs that we have written in Aspectual Caml, we briefly sketch two of them.

The one is, as we have seen thought the paper, to augment an interpreter of a simple language with additional kinds of terms, such as subtraction. Although it is a very small program, the aspect illustrates its usefulness for pluggable extension; since the aspect does not require to change the original interpreter definitions, we can easily fall back to the original language.

The second application program is larger. It extends a compiler of an untyped language to support static type system. The base part of the program define types for the parse trees of the source and intermediate languages and functions that translate the parse tree in the source language into the intermediate language called K-normal forms. The aspects extend the type of the source parse tree with type information, and modifies the transformation functions to carry out type inference during the transformation.

The aspects in the program can improve comprehensibility of the compiler implementation in particular educational purposes. Since the translation rules in the original can be complicated by the types, separating the compiler into the one for untyped language and the extension for types would clarify both the core translation rules and the interaction between translations and type system.

The second program, which consists of approximately 100 lines of base program and 100 lines of aspect definitions, is shown in Appendix A.

5. RELATED WORK

AspectJ[14, 15] is the first AOP language that offers both the pointcut and advice and inter-type declaration mechanisms. Aspectual Caml is principally designed by following those mechanisms. However, we see AspectJ-family of languages might be too complicated to theoretically study the AOP features as they primarily aim practical languages. For example, AspectJ 1.2 compiler type checks the following advice declaration:

```
Object around() : call(Integer *.*(..))
{ return new Float(0); }
```

even though it could cause a runtime error if applied to an expression like `Integer.decode("0").intValue()`. A simpler language that yet has a notion of polymorphism would help to reason about such a situation.

There are several proposals of theoretical models of AOP features. As far as the authors know, most work merely on the pointcut and advice mechanism. Aspect SandBox[22] describes a semantics of an dynamically-typed procedural language with a pointcut and advice mechanism. Tucker and Krishnamurthi presented a pointcut and advice mechanism in dynamically-typed functional languages[19]. MiniAML is a core calculus for expressing the pointcut and advice mechanism in strongly-typed functional languages[21]. Such a calculus would be suitable to describe the language design of Aspectual Caml, which is currently explained at the source language level. AspectML is an AOP extension to Standard ML with the pointcut and advice mechanism[21]. The semantics of AspectML is defined as a translation into MiniAML. TinyAspect is a model of pointcut and advice mechanism for strongly-typed languages with ML-like modules[1]. It proposes a module system for aspects so as to protect join points in a module from aspects outside the module.

There are several studies for adding fields or constructors into existing types, but not in the context of aspect-oriented programming. Type-safe update programming provides a means of extending existing data types[8], which inspired the type extension mechanism in Aspectual Caml. Polymorphic variants[10] allow to define functions that manipulate variant records without prior declaration of the variant type. This can improve code re-usability of a program when it uses polymorphic variants instead of ordinary variants[11]. Since there have been many programs that developed with ordinary variants, we believe that the polymorphic variants and our type extension mechanism would complement each other.

6. CONCLUSION

This paper presented the design and implementation of Aspectual Caml, an AOP functional language. The language design aims at developing practical applications by adapting many AOP features in existing AOP languages. In order to fit for the programming styles in strongly-typed functional languages, we reconsidered AOP features, including type inference of aspects, polymorphism in pointcuts, and type extension mechanisms. We believe that those features would serve a good basis for further theoretical development of AOP features such as type safety.

A compiler of an Aspectual Caml subset is implemented as a translator to Objective Caml. It is capable to compile nontrivial application programs in which base and aspect definitions deeply interact. Those application programs would also demonstrate that AOP is as useful in functional programming as in object-oriented programming.

We plan to work more on the design and implementation of Aspectual Caml. In particular, a module system for aspects that would nicely work with the ML module system would be needed. We also consider further polymorphism in advice bodies so as to easily define type universal aspects like tracing. One idea is to integrate the language with G'Caml[9] so that advice can use functions that can examine values in different types.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In R. L. Curtis Clifton and G. T. Leavens, editors, *FOAL2004*, Technical Report TR#04–04,

Department of Computer Science, Iowa State University, Mar. 2004.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference record of Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[4] J. Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *Proc. of AOSD '04*, pages 5–6. ACM Press, 2004. Invited Industry Paper.

[5] B. Burke and A. Brok. Aspect-oriented programming and JBoss. Published on The O'Reilly Network, May 2003. `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`.

[6] B. de Alwis and G. Kiczales. Apostle: A simple incremental weaver for a dynamic aspect language. Technical Report TR-2003-16, Dept. of Computer Science, University of British Columbia, 2003.

[7] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, Oct. 2001.

[8] M. Erwig and D. Ren. Type-safe update programming. In *ESOP 2003*, volume 2618 of *LNCS*, pages 269–283, 2003.

[9] J. Furuse. *Extensional Polymorphism: Theory and Application*. PhD thesis, Université Denis Diderot, Paris, Dec. 2002.

[10] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[11] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, Nov. 2000.

[12] R. Hirschfeld. Aspects - AOP with squeak. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Oct. 2001.

[13] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, number 1241 in LNCS, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[17] X. Leroy. A proposal for recursive modules in Objective Caml. `http://cristal.inria.fr/~xleroy/publi/recursive-modules-note.pdf`.

[18] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proc of TOOLS2002*, pages 18–21, Sydney, Australia, Feb. 2002.

[19] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proc. of AOSD2003*, pages 158–167. ACM Press, 2003.

[20] P. Wadler. Functional programming: An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, 1998.

[21] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP2003*, 2003.

[22] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In R. Cytron and G. T. Leavens, editors, *FOAL2002*, Technical Report TR#02–06, Department of Computer Science, Iowa State University, pages 1–8, Enschede, The Netherlands, Apr. 2002.

APPENDIX

A. AN EXAMPLE OF COMPILER WRITTEN IN ASPECTUAL CAML

A.1 Base Program: A Simple Compiler

```
(*type of identifiers*)
type ident = I of string
let ppI (I(x)) = x
(*type of immediate values*)
type imm =
  | Int of int
  | Float of float
(*type of terms*)
type syntax =
  | S_Let of ident * syntax * syntax
  | S_Var of ident
  | S_LetRec of s_fundef list * syntax
  | S_App of syntax * syntax list
  | S_NegInt of syntax
  | S_SubInt of syntax * syntax
  | S_IfLEInt of syntax * syntax * syntax * syntax
  | S_Imm of imm
(*mutually recursive functions*)
and s_fundef = { s_name : ident;
 s_args : ident list;
 s_body : syntax }
(*type of terms after K normalizing*)
type knormal =
  | K_Let of ident * knormal * knormal
  | K_Var of ident
  | K_LetRec of k_fundef list * knormal
  | K_App of ident * ident list
  | K_NegInt of ident
  | K_SubInt of ident * ident
  | K_IfLEInt of ident * ident * knormal * knormal
  | K_Imm of imm
and k_fundef = { k_name : ident;
 k_args : ident list;
 k_body : knormal }
(*return a fresh identifier*)
let fresh_knormal =
  let r = ref 0 in
    fun () -> (incr r;
```

```
      I(("_knormal_" ^ (string_of_int !r))))
(*K normalizing for the constructor LetRec*)
let rec knormal_letrec fundef = match fundef with
    [] -> []
  | {s_name = ident;
     s_args = ident_list;
     s_body = exp}::tl ->
     {k_name = ident;
      k_args = ident_list;
      k_body = (knormal exp)}::(knormal_letrec tl)
(*K normalizing for the constructor App*)
and knormal_app exp explist =
(*omitted for limited space*)
(*K normalizing*)
and knormal = function
    S_Var(x) -> K_Var(x)
  | S_NegInt(exp) ->
      let f x = K_NegInt(x) in
      insert_let (knormal exp) f
  | S_SubInt(exp1, exp2) ->
      insert_let (knormal exp1)
                 (fun x ->
                   insert_let (knormal exp2)
                              (fun y -> K_SubInt(x, y)))
  | S_IfLEInt(exp1, exp2, exp3, exp4) ->
      insert_let (knormal exp1)
                 (fun x ->
                   insert_let (knormal exp2)
                              (fun y ->
                               K_IfLEInt(x, y,
                                         knormal exp3,
                                         knormal exp4)))
  | S_Let(ident, exp1, exp2) ->
      K_Let(ident, knormal exp1, knormal exp2)
  | S_LetRec(fl, exp) ->
      K_LetRec(knormal_letrec fl, knormal exp)
  | S_App(exp, explist) -> knormal_app exp explist
  | S_Imm(i) -> K_Imm(i)
and insert_let e c = match e with
    K_Var(x) -> c x
  | exp -> let fresh = fresh_knormal () in
      K_Let(fresh, exp, c fresh)
(*K normalizing main*)
let knormal_main s_exp = knormal s_exp
```

A.2 Aspect: Addition of Typing

```
aspect AddType
(*type of types for expressions*)
type typ =
    Tint
  | Tfloat
  | Tvar of typ option ref
  | Tfun of typ list * typ
(*type extension of identifiers for types*)
type+ ident = I of ... * typ{Tvar(ref None)}
(*returns fresh variables with types*)
let fresh_knormal_with_type =
  let r = ref 0 in
    fun typ -> (incr r;
                I("_knormal_" ^ (string_of_int !r), typ))
(*find out concrete type to short cut constructors Tvar*)
let rec get_type = function
    Tvar({contents = Some(t)}) -> get_type t
  | Tfun(t_list, t) -> Tfun(List.map get_type t_list,
                            get_type t)
  | t -> t
(* occur check *)
let rec occur r1 = function
    Tint | Tfloat -> false
  | Tfun(t2s, t2') -> List.exists (occur r1) t2s
                      || occur r1 t2'
  | Tvar(r2) when r1 == r2 -> true
  | Tvar({ contents = None }) -> false
  | Tvar({ contents = Some(t2) }) -> occur r1 t2
exception Unify of typ * typ
(*unification of two types*)
let rec unify t1 t2 =
(*omitted for limited space*)
(*typing after K normalizing*)
let rec id_typing k_exp t_env = match k_exp with
    K_Var(I(x, typ)) -> begin try
      let typ1 = List.assoc x t_env in
        unify typ typ1; typ
    with Not_found ->
       failwith ("unbound_variable: " ^ x) end
  | K_NegInt(I(_, typ)) -> unify typ Tint; typ
  | K_SubInt(I(_, typ1), I(_, typ2)) ->
       unify typ1 Tint; unify typ2 Tint; Tint
  | K_Let(I(name, typ), k_e1, k_e2) ->
      let typ1 = id_typing k_e1 t_env in
        unify typ1 typ;
        id_typing k_e2 ((name, typ1)::t_env)
  | K_LetRec(k_fundef_list, k_e) ->
      let new_t_env =
        id_typing_letrec k_fundef_list t_env in
      id_typing k_e new_t_env
  | K_App(id, id_list) ->
      id_typing_app id id_list t_env
  | K_IfLEInt(I(_, typ1), I(_, typ2), k_e1, k_e2) ->
      unify typ1 Tint; unify typ2 Tint;
      let ke1_typ = id_typing k_e1 t_env in
      let ke2_typ = id_typing k_e2 t_env in
        unify ke1_typ ke2_typ;
        ke1_typ
  | K_Imm (Int _) -> Tint
  | K_Imm (Float _) -> Tfloat
(*typing after K normalizing
 for the constructor LetRec *)
and id_typing_letrec k_fundef_list t_env =
(*omitted for limited space*)
(*typing after K normalizing for the constructor App*)
and id_typing_app fun_id arg_ids t_env =
(*omitted for limited space*)
(*a flag to judge if a call to knormal is recursive*)
let rec flag = ref false
(*adding typing expression after K normalizing*)
advice knormal_with_typing =
  [around (call knormal s_exp)]
    if not !flag
    then
      let _ = flag := true in
      let k_exp = proceed s_exp in
      let _ = id_typing k_exp [] in
      let _ = flag := false in k_exp
    else
      let k_exp = proceed s_exp in k_exp
end
```