

Good Advice for Type-directed Programming

Aspect-oriented Programming and Extensible Generic Functions

Geoffrey Washburn Stephanie Weirich

University of Pennsylvania
{geoffw,sweirich}@cis.upenn.edu

Abstract

Type-directed programming is an important idiom for software design. In type-directed programming the behavior of programs is guided by the type structure of data. It makes it possible to implement many sorts of operations, such as serialization, traversals, and queries, only once and without needing to continually revise their implementations as new data types are defined.

Type-directed programming is the basis for recent research into “scrapping” tedious boilerplate code that arises in functional programming with algebraic data types. This research has primarily focused on writing type-directed functions that are closed to extension. However, Lämmel and Peyton Jones recently developed a technique for writing openly extensible type-directed functions in Haskell by making clever use of type classes. Unfortunately, this technique has a number of limitations such as the inability to write specialized cases for existential or nested data types and function types becoming too constrained to be used as first-class functions.

We present an alternate approach to writing openly extensible type-directed functions by using the aspect-oriented programming features provided by the language AspectML. Our solution not only avoids the limitations present in Lämmel and Peyton Jones’s technique, but also allows type-directed functions to be extended at any time with cases for types that were not even known at compile-time. This capability is critical to writing programs that make use of dynamic loading or runtime type generativity.

Categories and Subject Descriptors D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features—abstract data types, patterns, polymorphism; F.3.3 [STUDIES OF PROGRAM CONSTRUCTS]: Software—[control primitives, type structure, program and recursion schemes; F.4.1 [MATHEMATICAL LOGIC AND FORMAL LANGUAGES]: Mathematical Logic—Lambda calculus and related systems

General Terms Design, Languages, Theory

Keywords type-directed programming, aspect-oriented programming, generic programming, type analysis, open extension, expression problem

1. Introduction

Type-directed programming (TDP) is an invaluable idiom for solving a wide class of programming problems. The behavior of a type-directed function is guided by the type structure of its arguments.

Serialization, traversals, and queries can all be implemented via TDP, with the benefit that they need only be written once and will continue to work even when new data types are added as the program evolves.

TDP has become a popular technique for eliminating boring “boilerplate” code that frequently arises in functional programming with algebraic data types [12, 13, 8, 7]. The goal in this research has been to develop easy-to-use libraries of type-directed combinators to “scrap” this boilerplate. For example, Lämmel and Peyton Jones’s Haskell library provides a combinator called `gmapQ` that maps a query over an arbitrary value’s children returning a list of the results. Using `gmapQ` it becomes trivial to implement a type-directed function, `gsize`, to compute the size of an arbitrary value.

```
gsize :: Data a => a -> Int
gsize x = 1 + sum (gmapQ gsize x)
```

The function `gsize` works by mapping itself over the children of its argument `x`, summing the result and incrementing for the implicit constructor in `x`.

However, this line of research focused on *closed* type-directed functions. Closed type-directed functions require that all cases be defined in advance. Consequently, if a programmer wanted `gsize` to behave in a specialized fashion for one of her own data types, she would need to edit the definition of `gmapQ`.

The alternative is to provide *open* type-directed functions. Lämmel and Peyton Jones developed a method for writing openly extensible type-directed functions, via clever use of Haskell type classes [14]. While their technique provides the extensibility that was previously missing, there are caveats:

- It only allows open extension at compile-time, preventing the programmer from indexing functions by dynamically created or loaded data types.
- It is not possible to openly extend type-directed functions with special cases for existential data types [15] because their definition must explicitly list which type classes may be used on the existential type parameters. Consequently, every time a new type-directed function is defined the data type definition will need to be updated. This quickly becomes unmanageable.
- It is not possible to write openly extensible type-directed functions for nested data types [3] because they introduce unsatisfiable type class constraints.
- Each open type-directed function used introduces a typing constraint into the environment, which for polymorphic functions is captured as part of the function’s type. This makes it difficult to use these functions as first-class values because their types may be too constrained to be used as arguments to other functions.

Because of these restrictions, Lämmel and Peyton Jones retain their old TDP mechanism as a fallback when openly extensible type-directed functions cannot be used. Providing distinct mechanisms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WGP’06 September 16, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-492-6/06/0009...\$5.00

for open and closed TDP significantly increases the complexity of TDP for the user. Additionally, it may not be easy for a user unfamiliar with type inference for type classes to understand why an extension to an open type-directed function fails to type-check.

We present an alternate approach to openly extensible TDP using the aspect-oriented programming (AOP) features provided by the language AspectML. Aspect-oriented programming languages allow programmers to specify *what* computations to perform as well as *when* to perform them. For example, AOP in AspectJ [11] makes it easy to implement a profiler that records statistics concerning the number of calls to each method. The *what* in this example is the computation that does the recording and the *when* is the instant of time just prior to execution of each method body. In AOP terminology, the specification of what to do is called *advice* and the specification of when to do it is called a *pointcut designator*. A set of pointcut designators and advice organized to perform a coherent task is called an *aspect*.

We, along with Daniel S. Dantas and David Walker, developed AspectML as part of our research on extending AOP to functional languages with parametric polymorphism [5]. In a language with parametric polymorphism, it is difficult to write advice when a function may be used at many types. It is also common to simultaneously advise functions with disparate types. Runtime type analysis, a mechanism for TDP, is used to address both of these issues.

However, as we began writing more complex examples in AspectML, we found that not only was runtime type analysis necessary for expressive advice, but that advice was necessary for expressive type-directed functions. In particular, aspects enable openly extensible TDP. We use advice to specify *what* the behavior of a specific type-directed function should be for a newly defined type, or combination of types, and *when* a type-directed function should use the case provided for the new type.

Furthermore, we believe that our entirely dynamic mechanism for AOP is critical to this expressiveness. Most other statically typed AOP languages have a compile-time step called “weaving” that fixes *which* advice is used. It may still be necessary to perform dynamic tests to determine *when* advice applies. In contrast, AspectML allows advice to be “installed” at any time. However, a compiler can still optimize away the overhead of advice dispatch in many cases. Consequently, not only does TDP in AspectML improve extensibility when writing programs, but also while the programs are running. For example, it is possible to safely patch AspectML programs at runtime and manipulate data types that did not exist at compile-time.

In the next section, we introduce AOP in AspectML via examples. In Section 3, we explain and show examples of TDP in AspectML. In Section 4, we show how AOP in AspectML can be used to openly extend type-directed functions. In Section 5, we provide a more detailed comparison with related research. Finally, in Section 6, we discuss future directions and summarize our results.

2. The AspectML Language

We begin by introducing the general design of AspectML. AspectML is a polymorphic functional, aspect-oriented language descended from the ML family of languages. The syntax for a fragment of AspectML is given Figure 1. It is most similar to Standard ML [20], but occasionally uses OCaml or Haskell syntax and conventions [16, 23].

We use over-bars to denote lists or sequences of syntactic objects: \bar{x} refers to a sequence $x_1 \dots x_n$, and x_i stands for an arbitrary member of this sequence. We use a monospaced font to indicate AspectML programs or fragments. Italicized text is used to denote meta-variables. We assume the usual conventions for variable binding and α -equivalence of types and terms.

(kinds)	$k ::= *$ $k_1 \rightarrow k_2$	kind of types function kinds
(polytypes)	$s ::= \langle \bar{a}[:k] \rangle r$	
(ρ -types)	$r ::= t$ $s \rightarrow r$ $pc \ \bar{p}t$	monotypes higher-rank function pointcut
(pointcut type)	$pt ::= \langle \bar{a}[:k] \rangle s \rightsquigarrow r$	
(monotypes)	$t ::= a$ T $Unit$ $String$ Int $t_1 \rightarrow t_2$ (t_1, \dots, t_n) $[t]$ $t_1 \ t_2$	type variables type constructors unit strings integers functions tuples lists type application
(terms)	$e ::= x$ C $()$ $" \dots "$ i $e_1 \ e_2$ $let \ d \ in \ e \ end$ $(fn \ \bar{p} \Rightarrow e)$ $case \ e \ of \ \bar{p} \Rightarrow e$ $typecase \ t \ of \ \bar{t} \Rightarrow e$ $\#f[:pt]\#$ any (e_1, \dots, e_n) $[e_1, \dots, e_n]$ $e:r$	term variables data constructors unit strings integers application let declarations abstraction pattern matching typecase pointcut set any pointcut tuples lists annotation
(patterns)	$p ::= _$ x C $()$ $" \dots "$ i $p_1 \ p_2$ $[p_1, \dots, p_n]$ (p_1, \dots, p_n) $p:s$	wildcard variable binding data constructors unit strings integers application tuples lists annotation
(trigger time)	$tm ::= before$ $after$ $around$	
(declarations)	$d ::= val \ p = e$ $fun \ f \ \langle \bar{a}[:k] \rangle (p) [: r] = e$ $datatype \ T [: k] = \bar{C}:\bar{r}$ $advice \ tm \ e_1 \ \langle \bar{a}[:k] \rangle (p_1, p_1, p_3) [: r] = e_2$ $case\text{-}advice \ tm \ e_1 \ (x:t, p_1, p_2) [: r] = e_2$	value binding recursive functions data type def advice def

Figure 1. AspectML syntax. Optional annotations are in math square brackets.

The type structure of AspectML has three layers: *polytypes*, *ρ -types*, and *monotypes*. Unless otherwise stated, when we refer to “types” in AspectML we mean monotypes. Polytypes are normally written $\langle \bar{a}[:k] \rangle r$ where $\bar{a}[:k]$ is a list of binding type variables with their kinds and r is a ρ -type. We often abbreviate $\bar{a} : *$ as \bar{a} . When $\bar{a}[:k]$ is empty, we abbreviate $\langle \rangle r$ as r . This three-way partitioning of types is used as part of AspectML’s treatment of higher-rank polymorphism [24].

The category of ρ -types include monotypes, higher-rank polymorphic functions, and $pc \ \bar{p}t$, the type of a pointcut, which in turn binds a list of type variables in a polytype and ρ -types. We explain pointcut types in more detail later.

In addition to type variables, a , simple base types like `Unit`, `String`, `Int`, and function types, the monotypes includes tuple and list types, (t_1, \dots, t_n) and $[t]$ respectively, type application for higher-kinded monotypes, and type constructors T . Tuple types are actually syntactic sugar for an infinite number of type constructors $TUPLE_n$ for n greater than two. Type variables may only be instantiated with monotypes; it is not possible to construct a list of higher-rank polymorphic functions, for example.

Type constructors can be defined using `datatype` declarations. The `datatype` declaration is essentially the same as in ML and Haskell, with the exception of support for GADTs [25]. The declarations are used to define a new type constructor, T , via a collection of data constructors. For example, lists in AspectML are definable by giving the types of the constructors.

```
datatype List = [] : <a> List a
              | (::) : <a> a -> List a -> List a
```

Note that, as in this example, it is always possible to elide the kind annotation when defining a type constructor because it can always be inferred. Because AspectML provides GADTs, we require that the programmer provide the complete types of the data constructors rather than just the argument types.

The term language of AspectML includes variables x , term constructors C , unit $()$, integers i , strings $" \dots "$, tuples, syntactic sugar for lists, anonymous functions, function application and let declarations. New recursive functions may be defined in declarations. These functions may be polymorphic, and they may or may not be annotated with their argument and result types. Furthermore, if the programmer wishes to refer to type parameters within these annotations of the body of the function, they must specify a binding set of type variables $\langle a : k \rangle$. When the type annotations are omitted, AspectML may infer them. However, inference in AspectML is limited to monotypes if no annotations are provided. AspectML does not yet support advice for curried functions, so the syntax for recursive function definitions only allows a single argument. However, curried functions may still be written by using anonymous functions.

2.1 AOP in AspectML

To support AOP, AspectML provides pointcuts and advice. Advice in AspectML is second-class and includes two parts: the body, which specifies what to do, and the *pointcut designator*, which specifies when to do it. A pointcut designator has two parts, a *trigger time*, which may be `before`, `after`, or `around`, and a *pointcut proper*, which is a set of function names and an optional pointcut type annotation. The set of function names may be written out verbatim as `#f#`, or, to indicate all functions, the `any` pointcut might be used. Note that only function names bound by function definitions may be used as part of pointcut sets; function parameters may not be advised. This is illustrated in the following example

```
fun f x = x + 1
val ptc = #f# (* allowed *)

fun h (g : Int -> Int) = #g# (* not allowed *)
```

This restriction is a consequence of the way we define the semantics of AspectML in terms of a type-directed translation. While it limits expressive power, we have not encountered compelling examples for advising first-class functions.

Informally, a pointcut type, $\langle \bar{a} \rangle s \rightsquigarrow r$, describes the I/O behavior of a pointcut. In AspectML, pointcuts are sets of functions, and s_1 and s_2 approximate the domains and ranges of those functions. For example, if there are functions f and g with types `String -> String` and `String -> Unit` respectively, the pointcut `#f,g#` has the pointcut type $pc \langle a \rangle String \rightsquigarrow a$. Because

their domains are equal, the type `String` suffices. However, they have different ranges, so we use a type variable to generalize them both. Any approximation is a valid type for a pointcut, so it would have also been fine to annotate `#f,g#` with the pointcut type $pc \langle a \ b \rangle a \rightsquigarrow b$. This latter type is the most general pointcut type, and can be the type for any pointcut, and is the type of any pointcut.

The pointcut designator `before #f#` represents the point in time immediately before executing a call to the function f . Likewise `after #f#` represents the point in time immediately after execution. The pointcut designator `around #f#` wraps around the execution of a call to the function f —the advice triggered by the pointcut controls whether the function call actually executes or not.

The most basic kind of advice has the form:

```
advice tm e1 (p1, p2, p2) = e2
```

Here, $tm\ e_1$ is the pointcut designator. When the pointcut designator dictates that it is time to execute the advice, the first pattern, p_1 , is bound either to the argument (in the case of `before` and `around` advice) or to the result of function execution (in the case of `after` advice). The second pattern, p_2 , is matched against a reification of the current call stack. Stack analysis in AspectML is interesting, but it is orthogonal to the expressiveness of TDP, so we do not discuss it in this paper. Refer to our journal article for more information on stack analysis [5]. The last pattern, p_3 , is matched against metadata describing the function that has been called, such as its name or location in the source text.

Since advice exchanges data with the designated control flow point, `before` and `after` advice must return a value with the same type as the first pattern p_1 . For `around` advice, p_1 has the type of the argument of the triggering function, and the advice must return a value with the result type of the triggering function.

A common use of AOP is to add tracing information to functions. These statements print out information when certain functions are called or return. For example, we can advise the program below to display messages before any function is called and after the functions f and g return. The trace of the program is shown on the right in comments.

```
(* code *)
fun f x = x + 1
fun g x = if x then f 1
          else f 0
fun h _ = False

(* Output trace *)
(*
(* entering g
(* entering f
(* leaving f => 2
(* leaving g => 2
(* entering h
*)

(* advice *)
advice before any (arg, _, info) =
  (print ("entering " ^ (getFunName info) ^ "\n");
   arg)

advice after #f,g# (arg, _, info) =
  (print ("leaving " ^ (getFunName info) ^
        " => " ^ (intToString arg) ^ "\n");
   arg)

val _ = h (g True)
```

Even though some of the functions in this example are monomorphic, polymorphism is essential. Because the advice can be triggered by any of these functions and they have different types, the advice must be polymorphic. Moreover, since the argument types of functions f and g have no type structure in common, the argument `arg` of the `before` advice must be completely abstract. On the other hand, the result types of f and g are identical, so we can fix the type of `arg` to be `Int` in the `after` advice.

In general, the type of the advice argument may be the most specific polytype s such that all functions referenced in the pointcut are instances of s . Inferring s is not a simple unification problem; instead, it requires *anti-unification* [26]. AspectML can often use anti-unification to compute this type. It is decidable whether it is possible to reconstruct the pointcut type via anti-unification, so the implementation warns the user if they must provide the annotation.

Finally, we present an example of around advice. Again, around advice wraps around the execution of a call to the functions in its pointcut designator. The `arg` passed to the advice is the argument that would have been passed to the function had it been called. In the body of around advice, the `proceed` function, when applied to a value, continues the execution of the advised function with that value as the new argument.

In the following example, a cache is installed “around” the `f` function. First, a cache (`fCache`) is created for `f` with the externally-defined `cacheNew` command. Then, around advice is installed such that when the `f` function is called, the argument to the function is used as a key in a cache look-up (using the externally-defined `cacheGet` function). If a corresponding entry is found in the cache, the entry is returned as the result of the function. If a corresponding entry is not found, a call to `proceed` is used to invoke the original function. The result of this call is placed in the cache (using the externally-defined `cachePut` function) and is returned as the result of the `f` function.

```
val fCache : Ref List (Int,Int) = cacheNew ()

advice around ##f (arg, _, _) =
  case cacheGet (fCache, arg)
  | Some res => res
  | None => let
    val res = proceed arg
    val _ = cachePut (fCache, arg, res)
  in
    res
  end
```

We can transform this example into a general-purpose cache inserter by wrapping the cache creation and around advice code in a function that takes a first-class pointcut as its argument as described below. Finally, we do not provide their implementations here, but `cacheGet` and `cachePut` functions are polymorphic functions that can be called on caches with many types of keys. As such, the key comparisons use a polymorphic equality function that relies on the runtime type analysis described in Section 3.

Another novel feature of AspectML is the ability to use pointcuts as first-class values. This has been possible in some dynamically typed languages, such as Smalltalk [9], but AspectML is the first statically typed language to offer this feature. AspectJ can express some of the same idioms using abstract advice, but the expressiveness power of abstract aspects is fundamentally different. This facility is extremely useful for constructing generic libraries of profiling, tracing or access control advice that can be instantiated with whatever pointcuts are useful for the application. Recall the first example in Section 2, where we constructed a logger for the `f` and `g` functions. We can instead construct an all-purpose logger that is passed the pointcut designators of the functions we intend to log with the following code, assuming the existence of a type-directed serializer `toString`.

```
fun startLogger (toLog:pc (<a b> a -> b)) =
  let
    advice before toLog (arg, _, info) =
      ((print ("before " ^ (getFunName info) ^ ":" ^
        (toString arg) ^ "\n")); arg)
    advice after toLog (res, _, info) =
      ((print ("after " ^ (getFunName info) ^ ":" ^
        (toString res) ^ "\n")); res)
  in
    ()
  end
```

The AspectML pointcut language may seem limited compared to that provided by a language such as AspectJ, but nearly all pointcuts provided by AspectJ can be expressed in AspectML. This is partly because pointcuts are first-class values, so we can write combinators that act upon pointcuts. Furthermore, we do not need special pointcuts for reading and writing to mutable storage because they are implemented in terms of regular functions that may be advised like any other function in scope. AspectML’s stack analysis capabilities, not discussed in this paper, can be used to implement all of the usual “CFLOW” and “within” style pointcuts. The only sort of pointcuts in AspectJ that we cannot express in AspectML are those that advise other advice and those that advise data type constructors. We may add support for these in the future.

3. TDP in AspectML

The tracing example, in Section 2.1, could only print the name of the function called in the general case. If we would also like to print the arguments as part of the tracing aspect we must take advantage of AspectML’s runtime type analysis. For example, consider a revised tracing aspect that can print the arguments to any function that accepts integers or booleans:

```
advice before any <a>(arg : a, _, info) =
  (print ("entering " ^ (getFunName info) ^
    "with arg " ^
    (typecase a
    | Int => (int_to_string arg) ^ "\n"
    | Bool =>
      if arg then "True\n" else "False\n"
    | _ => "<unprintable>\n"));
  arg)
```

This advice is polymorphic, and the argument type `a` is bound by the annotation `<a>`. This ability to alter control-flow based on the type of the argument means that polymorphism is not parametric in AspectML—programmers can analyze the types of values at runtime. However, without this ability we cannot implement this generalized tracing aspect and many other similar examples.

Because dispatching on the type of the argument to advice is such a common idiom, AspectML provides syntactic sugar called *case-advice*, which is triggered both by the pointcut designator and the specific type of the argument. In the code below, the first piece of advice is only triggered when the function argument is an integer, the second piece of advice is only triggered when the function argument is a boolean, and the third piece of advice is triggered by any function call. (All advice that is applicable to a program point is triggered in inverse order from their definitions—newer advice has higher precedence over older advice.)

```

case-advice before any (arg : Int, _, _) =
  (print ("with arg " ^ (intToString arg) ^ "\n");
   arg)

case-advice before any (arg : Bool, _, _) =
  (print ("with arg " ^
        (if arg then "True\n" else "False\n"));
   arg)

advice before any (arg, _, info) =
  (print ("entering " ^ (getFunName info) ^ "\n");
   arg)

```

The code below and its trace demonstrates the execution of the advice. Note that even though `h`'s argument is polymorphic, because `h` is called with an `Int`, the first advice above triggers in addition to the third.

```

(* code *)           (* Output trace *)
fun f x = x + 1      (* *)
fun g x = if x then (* entering g with arg True *)
  f 1               (* entering f with arg 1 *)
  else              (* entering h with arg 2 *)
  f 0
fun h _ = False
val _ = h (g True)

```

3.1 Writing total type-directed functions

While very important for programming in the large, data type definitions pose a problem for TDP in AspectML: A `typecase` expression can only have a finite number of cases, yet the programmer may use `datatype` to define a new type constructor at any time. For type-directed functions to be generally useful in AspectML it must be possible to write total type-directed functions that cover all possible cases. Furthermore, many type-directed functions are completely defined by the type-structure of their arguments and providing cases for each of them would introduce significant amounts of redundant code. Therefore, AspectML requires a mechanism for handling arbitrary type constructors and their associated data constructors.

We will use the term *atomic type* to refer to those monotypes in AspectML that are not defined via `datatype` and cannot be decomposed into simpler types. For example, integers, strings, and functions are atomic types in AspectML. Tuple types and type constructors are not considered atomic.

To allow programmers to write total type-directed functions, we provide a primitive `toSpine` for converting arbitrary data constructor values to *spine* abstractions, as introduced by Hinze, Löh, and Oliveira (HLO) [8, 7].

```

datatype Spine =
  | SCons : <a> a -> Spine a
  | SApp  : <a b> Spine (b -> a) -> b -> Spine a

```

Spines can be thought of “exploded” representations of a data constructor and the arguments that have been applied to it. The data constructor `SCons` is used to hold the actual constructor, and then a series of `SApp` constructors hold the arguments applied to the constructor. For example, the expression `toSpine (1::2::3::[])` produces the spine

```
SApp (SApp (SCons (op ::)) 1) (2::3::[])
```

Note that this alternate representation is only one layer deep: The two arguments of `::`, `1` and `2::3::[]`, are unchanged.

The function `toSpine` must be a primitive in AspectML, however, writing the inverse function, `fromSpine`, is straightforward.

```

fun fromSpine <a>(x : Spine a) : a =
  case x
  | SCons con => con
  | SApp spine arg => (fromSpine spine) arg

```

```

val escape : String -> String
val toString : <a> a -> String

fun toString <a>(x : a) =
  let
  fun spineHelper <a>(x : Spine a) =
    case x
    | SCons c =>
      (case (conToString c)
       | Some s => "(" ^ s ^ ")")
       | None => abort "impossible")
    | SApp spine arg =>
      "(" ^ (spineHelper spine) ^ " " ^
        (toString arg) ^ ")")
  in
  typecase a
  | Int => intToString x
  | String => "\"" ^ (escape x) ^ "\""
  | b -> c => "<fn>"
  | _ => (case (toSpine x)
          | Some spine =>
            spineHelper spine
          | None =>
            abort "impossible")
  end

```

Figure 2. `toString` in AspectML

Although the primitive `toSpine` function has type `<a> a -> Option (Spine a)`, and can therefore be used with any input, it only returns a spine when provided with a non-atomic type. This is because spines simply do not make sense for atomic types; for example, function types have no data constructor that “builds” λ -abstractions.

Finally, AspectML provides primitive implementations of equality and string conversion for integers, strings, and unapplied data constructors. Unapplied constructors are data constructors that have not been applied to any arguments. For example, `::` versus `1::[]`. The functions `eqCon` and `conToString` have the somewhat unusual types: `<a> a -> Option Bool` and `<a> a -> Option String` respectively. The reason they return `Options` is much like the reason `toSpine` returns an `Option`—the polymorphic type allows them to be applied to things other than unapplied data constructors. Finally to address the problem that there are infinitely many tuple type constructors we introduce a primitive `tupleLength` of type `<a> a -> Option Int` that if given an unapplied tuple type constructor will return an `Option` containing its length. For all other values it will return `None`.

Putting everything together, we can now write a type-directed pretty printer, `toString`, as shown in Figure 2. If `toString` is applied to an integer it uses the primitive `intToString` function to convert it to a string. If it is applied to a string it calls the `escape` function to escape any special characters and then wraps the string in quotes. For function types, `toString` simply returns the constant string `<fn>` because there is no way to inspect the structure of functions. Finally, for all other types `toString` will attempt to convert the value to a spine and then call `spineHelper`. Even though we know that `toSpine` will always succeed, because the `typecase` covers all atomic types, we must still perform the check to make sure a `Spine` was actually constructed.

The function `spineHelper` walks down the length of the spine and makes recursive calls to `toString` on the arguments and upon reaching the head converts the unapplied constructor to a string using `conToString`. As with `toSpine`, it is again necessary to check the output of `conToString` because the type system does not know that the argument of `SCons` is guaranteed to be an unapplied constructor.

```

val gmapT : <a> a -> (<b> b -> b) -> a
val gmapTspine : <a> Spine a ->
  (<b> b -> b) -> Spine a

fun gmapT <a>(x:a) = fn (f:<b> b -> b) =>
  (case (toSpine x)
   | Some y => fromSpine (gmapTspine y f)
   | None => x)

and gmapTspine <a>(x:Spine a) = fn (f:<b> b -> b) =>
  case x
  | SCons _ => x
  | SApp spn arg =>
    SApp (gmapTspine spn f) (f arg)

```

Figure 3. gmapT in AspectML

Furthermore, why did we choose to write `spineHelper` rather than just adding a branch to the `typecase` expression for the type constructor `Spine`, and then calling `toString` recursively directly? There are two problems that arise with such an implementation. Firstly, the behavior of `toString` will be incorrect if the argument is a `Spine`. For example, `toString (toSpine[1])` would produce the string `"(Cons 1 Nil)"` rather than `"(SApp (SApp (SCons Cons) 1) Nil)"`.

Secondly, such an implementation may only be given “well-formed” Spines, that is, spines that have unapplied data constructors at their head. If we recall the definition of the `Spine` data type, we see that there is nothing preventing a client from calling `toString` on spines that do not have unapplied constructors in them, like `(SCons 5)`. The implementation in Figure 2 avoids this problem by converting all non-atomic types into a spine before converting them to strings. Because `toSpine` is guaranteed to construct well-formed Spines, it is guaranteed to never fail.

3.2 Scrapping your boilerplate

It is also straightforward to use the TDP infrastructure of AspectML to implement the various combinators Lämmel and Peyton Jones developed for generic programming.

In Figure 3 is an implementation of `gmapT`, a “single-layer” generic mapping function. Given any value and a polymorphic function, it applies the function to the children the value has, if any. It is implemented in terms of a helper function for spines, `gmapTspine`. This helper is fairly straightforward. Because the `Spine` data type makes the arguments to a constructor explicit via the `SApp` constructor, it only need walk down the spine and apply the function `f` to each argument.

Finally, `gmapT` ties everything together by attempting to convert the argument to a `Spine`. If `toSpine` fails, the argument must be atomic and has no children to map over. If `toSpine` succeeds, `gmapTspine` is applied to the spine and `fromSpine` is used to convert back to the original type.

To effectively use `gmapT` we need a way to construct a polymorphic function of type ` b -> b` that is neither the identity nor the diverging function. Lämmel and Peyton Jones initially implemented this using a combinator `extT` that lifts a monomorphic function of type `t -> t`, for any type `t`, to be ` b -> b`. They do this using a primitive casting operation, but it is simple to implement in AspectML, because it is possible to use abstracted types within type patterns

```

fun extT <a>(f:a -> a) = fn <b> (x:b) =>
  typecase a of b => f x
  | _ => x

```

```

val == : <a> a -> a -> Bool

fun == <a>(x:a) = fn (y:a) =>
  let
    fun spineHelper <a>(x:Spine a, y:Spine a) =
      case (x, y)
      | (SCons c1, SCons c2) =>
        (case (eqCon c1 c2)
         | Some z => z
         | None => abort "impossible")
      | (SApp sp1 arg1, SApp sp2 arg2) =>
        (case (cast sp2, cast arg2)
         | (Some sp2', Some arg2') =>
           (sp1 == sp2') andalso (arg1 == arg2')
         | _ => False)
    in
      typecase a
      | Int => eqInt x y
      | String => eqString x y
      | _ =>
        (case (toSpine x, toSpine y)
         | (Some x', Some y') => spineHelper (x', y')
         | (None, None) => abort "equality undefined")
    end

```

Figure 4. Polymorphic structural equality in AspectML

It is then straightforward to use `gmapT` to write a generic traversal combinator

```

fun everywhere <a>(f:<b> b -> b) = fn (x:a) =>
  f (gmapT x (everywhere f))

```

While very useful, `toString`, `gmapT` and `everywhere` may only traverse over a single value at a time. Many type-directed functions need to be able to operate on multiple values simultaneously. The generic structural equality, shown in Figure 4, is the prototypical example. For atomic values that are integers or strings, the primitive equality functions are used. However, if both values have a spine representation, the spines are compared with the function `spineHelper`. If both spines are unapplied data constructors, `spineHelper` uses the primitive equality, `eqCon`. Otherwise, if both spines are application spines, the arguments and remainder of the spines are compared recursively.

Because spines are existential data types, it is necessary to use a `cast` in the case for `SApp` in order to ensure that the arguments and constituent spines have the same types. Otherwise it would not be possible to call `==` or `spineHelper` recursively, as they both require two arguments of the same type. Like `extT`, writing `cast` is trivial in AspectML.

```

fun cast <a b>(arg : a) : Option b =
  typecase a of b => Some arg
  | _ => None

```

So far we have only transliterated some standard examples of TDP into AspectML. We believe that our realization of these functions is quite elegant, but the most significant benefit comes from the dynamic open extension allowed by AspectML. We will examine this capability in the next section.

4. Open Extensibility

By default, `toString`, `gmapT`, `everywhere`, and `==` work with all AspectML monotypes, but the programmer has no control over the behavior for specific types. This is a problem because the programmer might define a new data type for which the default behavior is incorrect. This problem is easily solved in AspectML using advice.

```

datatype Queue = Q : <a> List a -> List a -> Queue a

val emptyQueue : <a> Queue a
val emptyQueue = Q [] []

val enqueue : <a> a -> Queue a -> Queue a
fun enqueue e = fn (Q l1 l2) => Q l1 (e::l2)

val dequeue : <a> Queue a -> Option (a, Queue a)
fun dequeue q =
  case q
  | Q [] [] => None
  | Q [] l2 => dequeue (Q (rev l2) [])
  | Q (h::t) l2 => Some (h, Q t l2)

```

Figure 5. Amortized constant time functional queues

Starting out simple, while `toString` from Figure 2 can produce string representation for all types, its string representations of tuples is not what we would expect. The application `toString (1, True)` will generate the string `"(Tuple2 1) True"`. We can improve the output by writing advice for `toString` to provide a specialized case for tuples.

```

advice around #toString# <a>(x : a, _, _) =
  let
    fun helper <a>(spine : Spine a) =
      case spine
      | SCons _ => ""
      | SApp spine arg =>
        (helper spine) ^ ", " ^ (toString arg)
  in
    case (tupleLength x)
    | Some i =>
      (case (toSpine x)
      | Some spine => "(" ^ (helper spine) ^ ")"
      | None => abort "impossible"
      | None => proceed x
      end
    end

```

This advice triggers on any invocation of `toString`, and checks whether the argument is a tuple using the `tupleLength` primitive we discussed in Section 3.1. If it is not a tuple, it uses `proceed` to return control to the `toString` function. Otherwise, it converts the tuple to a `Spine` and then walks down the spine producing a comma separated string representation.

While it is nice that we can use advice to specialize the behavior of `toString`, the modification was simply for aesthetic purposes. There are cases where the default behavior of a type-directed function is incorrect. Say the programmer defines a new data type and the default traversal behavior provided by `everywhere` is not correct; traversal order is especially important in the presence of effects. For example, consider the common functional implementation of queues via two lists in Figure 5. Elements are dequeued from the first list and enqueued to the second; if the dequeue list becomes empty the enqueue list is reversed to become the new dequeue list. By default, `everywhere` will traverse the dequeue list in the expected head to tail order (FIFO) but the default traversal will walk over the enqueue list in the reverse of what a user might expect.

One solution would be for the author of `Queue` to edit `everywhere` or `gmapT` to include a special case for queues. But, this solution is undesirable because their source code might not be available, and continually adding special cases would clutter them with orthogonal concerns.

If the author of the `Queue` data type knows that `everywhere` is implemented in terms of `gmapT`, can simply write the following

```

(* Abadi, Cardelli, Pierce, Plotkin style dynamics *)
datatype Dynamic = Dyn : <a> a -> Dynamic

(* Marshal arbitrary values to a string *)
val pickle : <a> a -> String

(* Unmarshal strings to dynamic values *)
val unpickle : String -> Dynamic

(* Open a file handle for I/O *)
val openFile : String -> Handle

(* Write to an open file handle *)
val writeFile : Handle -> String -> Unit

(* Read to an open file handle *)
val readFile : Handle -> String

(* Close an open file handle *)
val closeFile : Handle -> Unit

```

Figure 6. Simple primitives for dynamic loading

advice for `gmapT` to ensure that the elements are traversed in FIFO order.

```

case-advice around #gmapT# (x:Queue a,_,_) =
  fn (f:<b> b -> b) =>
    case x
    | Q l1 l2 => Q (map f (l1 @ (rev l2))) []

```

This case-advice intercepts a call to `gmapT` when the first argument is type `Queue a` for any `a`, otherwise `gmapT` is executed. The author might have also chosen to extend `everywhere`, directly, but the code would not be quite as succinct.

The author of the `Queue` data type will encounter another problem when using the type-directed structural equality function described in Section 3.2 because it distinguishes between too many values. For example, `==` does not equate the queues `Q [1] []` and `Q [] [1]` even though both of these queues are extensionally equivalent. In fact, this is an example of how structural equality breaks representation independence.

Fortunately it is again quite simple to use advice to extend `==` so that it correctly compares `Queues`.

```

case-advice around # == # (x:Queue a, _, _) =
  (fn (y : Queue a) =>
    case (x, y)
    | (Q l1 l2, Q l3 l4) =>
      (l1 @ (rev l2)) == (l3 @ (rev l4))
  )

```

This advice triggers when `==` is called on values of type `Queue a` and converts the constituent lists to a canonical form before comparing them.

The examples of open extension we have shown so far are quite useful, but are essentially the same as what is possible using Lämmel and Peyton Jones's technique, modulo its restrictions. Our approach however, extends to uses that are simply not possible with static approaches. For example, consider extending with `AspectML` dynamic loading, using the primitives defined in Figure 6. The functions `openFile`, `writeFile`, `readFile`, and `closeFile` are straightforward functions for file I/O.

The two primitives `pickle` and `unpickle` are a bit more complicated. The `pickle` primitive is similar to the function `toString` we defined, but includes support for meaningful serialization of functions. Its inverse `unpickle` returns an existential data type, `Dynamic`. This is because we cannot know the type of the value that will be produced in advance. However, thanks to TDP it is possible to

```

datatype Extension = Ext : Dynamic ->
                    (Unit -> Unit) ->
                    Extension

advice after #unpickle# (d : Dynamic, _, _) =
  case d
  | Dyn <a> x =>
    (typecase a
     | Extension =>
       (case x of (Ext d f) => (f ()); d)
     | _ => d)

```

Figure 7. Extended dynamic loading

```

val myQueue = enqueue 2 (enqueue 1 emptyQueue)

val f =
  (fn () =>
   let
     case-advice around #toString# (x:Queue a,_,_) =
       case x
       | Q l1 l2 =>
         "[|" ^
           concat " " (map toString (l1 @ (rev l2))) ^
           "|]"

     case-advice around #gmapT# (x:Queue a,_,_) =
       fn (f:<b> b -> b) =>
         case x
         | Q l1 l2 => Q (map f (l1 @ (rev l2))) []

     case-advice around # == # (x:Queue a,_,_) =
       fn (y : Queue a) =>
         case (x, y)
         | (Q l1 l2, Q l3 l4) =>
           (l1 @ (rev l2)) == (l3 @ (rev l4))
   in
     ()
   end)

val file = openFile "testing.data"
val () = writeFile file
         (pickle (Extension (Dyn myQueue) f))
val () = closeFile file

```

Figure 8. Program compiled with functional queues

still do useful things with the data, even if we have no idea what it might be. For example, our type-directed functions `toString`, `everywhere`, and `==` can still be used on the value the `Dynamic` data type is hiding.

It is even possible for a program to manipulate data types that were not defined when it was compiled. Specifically, we might have two separate programs each with some data types unique to them. For example, imagine if one of our programs used the queues described in Figure 5, but another did not. This first program would be able to `pickle` a queue, write it to a file, and then the second program could read the queue out of the file and manipulate it as it would any other dynamically packaged value.

However, this second host will encounter the exact same problems we discussed earlier with `everywhere` and `==` behaving incorrectly on functional queues. If it was only possible to extend type-directed functions at compile-time there would be nothing that could be done. However, because we might use advice to dynamically extend type-directed functions, we can package data written to

disk with the necessary extensions. We show a fairly naïve implementation of this in Figures 7 and 8.

In Figure 7, we extend the dynamic loading primitives with a notion of “extension”. The new data type `Extension` packages up a dynamic value along with a function from `Unit -> Unit` that can be used to perform any necessary configuration required for the packaged value. We then provide advice for the `unpickle` primitive to check whether it has created an `Extension` value. If so, it invokes the function contained in the `Extension` and returns the included dynamic value as the result.

Our first program, that was compiled with functional queues can now store them to disk as shown in Figure 8. Here, it creates an configuration function that installs the relevant extensions to `toString`, `gmapT`, and `==` for functional queues. It then packages the function with a queue and writes it to disk. Now any other program built with the “extensible” version of the dynamic loading primitives in Figure 7 can read the queue from the disk, and at the same time receive appropriate advice for manipulating the data in a generic fashion. For example, it can call `toString` on the value it obtains from the file `testing.data` and obtain `"(Dyn [|1 2|])"`.

5. Related-work

In this section we will examine the wide spectrum of research that this paper is built upon.

5.1 Extensible type-directed programming

Scrapping your boilerplate with class Our examples in Sections 3 and 4 were adapted from the work of Lämmel and Peyton Jones on “scrapping your boilerplate” [12, 13, 14]. In their latest paper, Lämmel and Peyton Jones have attempted to address the problem of providing openly extensible type-directed functions. This is done by associating each type-directed function with a type class. However, this technique has many limitations. Most importantly, the Haskell type class mechanism requires that all instances for these classes be defined at compile-time. Consequently, it is impossible to use their library with types that are not known until runtime. This is necessary for any applications that make significant use of dynamic loading, such as the example in the previous section.

The second major limitation of their technique is that it is not possible to write openly extensible type-directed functions that have specialized behavior on two significant classes of data types: existential data types and nested data types. It is not possible to write open type-directed functions for existential data types because of their use of the type class mechanism. Consider the following example where we attempt to define a specialized case for `Size`, an open type-directed function to generically compute the size of a value.

```

class Data Size a => Size a where
  gsize :: a -> Int

data Exists :: * where Ex :: a -> Exists

instance Size Exists where
  -- Does not type check
  gsize (Ex (ex :: b)) = 1 + gsize ex

```

Here, in order to define an instance of `Size` for `Exists`, we need to call `gsize` on the existentially packaged value `ex`. However, calling `gsize` requires that there exist an instance of `Size` for type `b`. However, because `b` could be anything there is no way of determining whether such an instance exists. The only solution is to require that the `Exists` data type package the necessary type class dictionary itself.

```

data Exists :: * where
  Ex :: Size a => a -> Exists

instance Size Exists where
  -- Type checks
  gsize (Ex (ex :: b)) = 1 + gsize ex

```

However, this is simply not scalable as every time we would like to use an open type-directed function on values of type `Exists` it will be necessary to add another constraint to the data type definition.

Nested data types, such as Okasaki’s square matrices [21], cause other difficulties with type class constraints.

```

data Sq a = Zero a | Succ Sq (a, a)

```

In order to solve type-classes constraints for Lämmel and Peyton Jones’s implementation, a form of coinductive constraint solving is necessary. Therefore, a solver attempting determine whether the type class constraint for `Size [a]` is satisfiable will go through a constraint back-chaining process that looks like the following.

```

      Size [a]
      ↓
Size [a], Data Sized [a]
      ↓
Size [a], Data Sized [a], Sat (Sized [a])

```

First, `Size [a]` requires the instance `Data Sized [a]`. And `Data Sized [a]` needs `Sat (Sized [a])`. Finally, because `Size [a]` implies the existence of `Sat (Sized [a])`, and at this point the solver reaches as fixed-point.

With nested data types a fixed-point can never be reached. For example, an instance for `Size (Sq a)` will require an instance of `Data Sized (Sq a)` which needs both `Sat (Sized a)` and `Sat (Sized (Sq (a,a)))`. While `Sat (Sized (Sq (a, a)))` can be obtained from `Size (Sq (a, a))`, it in turn needs `Data Sized (Sq (a,a))` which in requires `Sat (Sized (Sq ((a,a), (a,a))))`, and so forth ad nauseam.

Finally, because each open type-directed function used introduces a constraint into a function’s type it becomes difficult to use them as first-class functions. This is because they may become more constrained than other functions are expecting their arguments to be. For example,

```

shrink :: Shrink a => a -> [a]

data Proxy (a :: * -> *)

data ShrinkD a = ShrinkD { shrinkD :: a -> [a] }

shrinkList :: (Eq a, Shrink a) => [a] -> [a]
shrinkList xs = nub (concat (map shrink xs))

gmapQ :: Data ctx a => Proxy ctx ->
  (forall b. Data ctx b => b -> r) ->
  a -> [r]

-- Ill-typed
shrinkTwice x = gmapQ (undefined :: Proxy ShrinkD)
  (shrinkList . shrink) x

```

Here `shrinkTwice` is ill-typed because `gmapQ` is expecting a polymorphic function with the constraint `Data ctx b` for some `ctx` and some `b`. However, the composition of `shrinkList` and `shrink` has the constraint `(Eq a, Shrink a)` which resolves to `(Eq a, Data Shrink a)` which is too constrained by the need for the `Eq` type class to be a valid argument.

This problem with constraints can be alleviated by either making `shrinkList` an openly extensible type-directed function or by introducing new dummy type classes that encapsulate constraints.

The former leads to large amounts of boilerplate for otherwise simple helper functions and the latter leads to a combinatorial explosion as the number of type-directed functions grow.

Concurrent with our work, Sulzmann and Wang have shown how to use constraint handling rules to resolve some of the difficulties encountered in Lämmel and Peyton Jones’s approach [29]. By using constraint handling rules to allow type class super-classing, in addition to the usual sub-classing, they eliminate the need for abstraction over type classes and recursive instances. However, their solution does not solve the problems with existential data types or over constrained function types.

Scrapping your boilerplate with spines Our use of spines to implement type-directed functions was heavily influenced by the closely related work by Hinze, Löh, and Oliveira on understanding Lämmel and Peyton Jones’s techniques in terms of spines. In their work, they opt to implement `toSpine` directly using type representations built from GADTs. They assume the existence of some mechanical method of producing all the necessary representations and cases for `toSpine`. However, in order to provide open extension, they use type classes in a fashion inspired by Lämmel and Peyton Jones. Therefore, their approach to open extension has many of the same problems with type classes.

In their more recent work, Hinze and Löh have shown how to extend spines so that it is possible to implement type-directed functions that can not only process arbitrary data types, but build values of arbitrary type. Additionally, they show how to perform type-directed operations on higher-kinds using what they call *lifted spines*. AspectML does not presently have support for writing type-directed functions that construct arbitrary data types. We could easily adapt the modifications to spines that Hinze and Löh developed, but we will most likely pursue an approach based upon Weirich’s concurrent research [36]. We could extend AspectML with support for *lifted spines*, but we think it would be better for us to focus on developing a general solution for writing functions with polykinded types [6].

A core calculus for open and closed TDP Along with Dimitrios Vytiniotis, we developed a core calculus, $\lambda_{\mathcal{L}}$, that could be used to implement type-directed functions that could be either closed or open to extension [31]. While it is possible to extend type-directed functions in $\lambda_{\mathcal{L}}$, the programmer must manage the extensions herself, and manually close off the recursion before the functions can be used. However, as $\lambda_{\mathcal{L}}$ was designed as a core calculus rather than a high-level programming language, it is not expected to be easy to write software in $\lambda_{\mathcal{L}}$.

A novelty of $\lambda_{\mathcal{L}}$ is its use of *label-sets* to statically ensure that a type-directed function was never given an argument for which it lacked a case. However, *label-sets* are very heavy-weight and lack sufficient precision to describe whether some inputs are acceptable. In AspectML, the programmer must rely upon the initial implementation of a type-directed function to be total. Lämmel and Peyton Jones use of type classes does not have this problem as constraint solving can be relied upon to know that the input will have instances for all the appropriate type classes. However, checking that a type-directed function covers all cases statically, requires knowing all types that may be used statically.

5.2 Generic programming

The research on Generic Haskell [18] is related to our work on AspectML in many ways. However, Generic Haskell has a number of limitations in comparison to AspectML. First, in Generic Haskell, all uses of type information are resolved at compile-time, making it impossible to extend to TDP with types that are created at runtime or loaded dynamically. Secondly, Generic Haskell does not provide any direct mechanism for open extension. The latest version of Generic

Haskell has added syntactic sugar to easily extend a type-directed function with new cases. However, these new cases do not change the behavior of the original function, as advice does.

Achten and Hinze examined an extension to the Clean language [1] to allowing dynamic types and generic programming to interact cleanly [2]. This extension is quite flexible, even providing polykinded types. However, the generic functions in this language extension do not provide any kind of open extension. Cheney and Hinze developed a library with very similar capabilities in Haskell in terms of type representations [4]. However, it is not possible to provide special cases for user defined types in their library without defining new type representations, and modifying the infrastructure.

5.3 Aspect-oriented programming

There has been an increasing amount of research into AOP in the presence of parametric polymorphism. Our language, AspectML, directly builds upon the framework proposed by Walker, Zdancewic, and Ligatti [32], but extends it with polymorphic versions of functions, labels, label sets, stacks, pattern matching, advice and the auxiliary mechanisms to define the meaning of each of these constructs. We also define “around” advice and a novel type inference algorithm that is conservative over Hindley-Milner inference, which were missing from WZL’s work.

Concurrently with the development of AspectML, Masuhara, Tatsuzawa, and Yonezawa [19] developed an aspect-oriented version of core OCaml they call Aspectual Caml. Their implementation effort is impressive and deals with several features we have not considered, including curried functions and extensible data types. There are similarities between AspectML and Aspectual Caml, but there are also many differences: Pointcut designators in AspectML can only reference names that are in scope, Aspectual Caml does not check pointcut designators for well-formedness, pointcuts in Aspectual Caml are second-class, and there is no formal description of the Aspectual Caml type system or operational semantics.

Later, Wang, Chen, and Khoo began examining language design problems in combining aspects with a polymorphic functional language [33]. Their design makes fundamentally different assumptions about aspects that lead to significant differences in expressiveness: Their advice is scoped such that it is not possible to install advice that will affect functions that have already been defined, their advice is named, like Aspectual Caml their pointcuts are second-class, finally their design does not provide a mechanism for examining the call-stack or obtaining information about the specific function being advised.

Jagadeesan, Jeffrey, and Riely have shown how to extend Generic Featherweight Java with aspects [10]. However, in their work they did not emphasize the importance of type analysis in the presence of parametric polymorphism, perhaps partly because of the predominance of subtype polymorphism in idiomatic Java programming. Their aspects however do implicitly perform an operation equivalent to `subtypeof` in Type-directed Java [38]. Finally, their language extension does not provide any mechanism for installing new advice at runtime.

The only other statically-typed implementation of aspects that provides dynamically installable advice is Lippert’s modification of AspectJ [17] so installing and uninstalling plug-ins that provide advice gives the expected behavior.

Predating the research on AOP, Palsberg, Xiao, and Lieberherr showed how “wrappers” can be used to write *adaptive software* [22]. Using a special language they would write adaptive operations that generated code for generic traversal similar to what we do dynamically. Later, Seiter, Palsberg, and Lieberherr developed *context object* and traversal extensions to C++ to support behavioral evolution of software [28]. Context objects can be seen as a generalization of the method update operation.

5.4 The expression problem

Our use of advice in this paper can be seen as a solution to the “expression problem”, but at the level of types. Coined by Phil Wadler, the expression problem arises when data types and operations on these types must be extended simultaneously. We can view types as an openly extensible data type, and then use advice to simultaneously extend type-directed functions with the necessary cases.

One of Masuhara, Tatsuzawa, and Yonezawa’s primary goals in developing Aspectual Caml was as a solution to the “expression problem”. Aspectual Caml’s ability to openly extend algebraic data types with additional data constructors required some method of extending functions over these data types to support the new cases. Much like we use advice to openly extend functions with specialized cases for user defined types, they used advice to extend functions with the needed cases for newly defined data constructors.

6. Conclusions and future work

We have shown that runtime advising is critical to the expressive power of type-directed functions, just as runtime type analysis is key to the expressive power of AOP in polymorphic functional languages. Furthermore, our fully dynamic approach to advice allows the benefits of open extensibility to be realized even for types not known at compile-time. However, our solution is hardly the final word and there are many modifications or extensions we would like to explore to improve upon this work.

One problem that we did not address is that our type-directed functions cannot restrict the domain of types they may operate upon. For example, in AspectML functions cannot be compared for equality. Therefore, polymorphic structural equality in AspectML cannot provide a sensible notion of equality on functions types. However, its type, `<a> a -> a -> Bool`, does not provide any indication that it will abort execution when provided two functions. As mentioned, our language $\lambda_{\mathcal{L}}$ attempted to solve this problem using label-sets, but this solution seems too heavyweight for practical programming. Furthermore, label-sets are still too imprecise. They can be used to prevent using polymorphic structural equality on values of function type, but will also prevent the comparison of reference cells containing functions, which do have a notion of equality in AspectML. We plan to explore whether separating the kind `*` into two or more subkinds could provide an acceptable middle-ground. A more expressive solution might be to allow the programmer to use regular patterns to specify the accepted domain.

Another direction we would like to explore is the capability to temporarily advise the behavior of functions in AspectML. Currently, once advice is installed, there is no way of removing it. Therefore, it is not possible to temporarily alter the behavior of a type-directed function, even if the desired scope of the extension is limited. Therefore, we plan to consider adding an `advise-with` form that allows programmers to write advice that only applies within the scope of a specific expression. A closely related extension would be to extend AspectML with named advice. We compile AspectML into a core calculus that already has first-class advice, so there seems to be little reason not to expose the additional expressive power this provides.

Another limitation we did not discuss was that AspectML does not offer full reflexivity [30]. Because `typecase` may only analyze monotypes, and type variables may only be instantiated with monotypes, it is not possible to use type-directed operations on pointcuts and higher-rank polymorphic functions. We do not have enough experience to gauge whether this is a significant limitation in practical programming. However, we have studied the problem in other settings and believe that extending AspectML with higher-order type analysis would be feasible [35, 37].

Concurrently with this paper, Weirich developed a new approach to TDP in Haskell [36]. We expect that we will revise the primitives we have chosen for handling user defined data types and tuples in AspectML to take advantage of what her study has learned about the use of representation types.

Despite all of the benefits of TDP, another problem we did not address was that type-directed functions can break representation independence and potentially even the invariants of abstract data types. We have shown that it is possible to prove a generalization of Reynolds's parametricity [27], theorem by tracking the information content of types and terms [34]. In the near future, we hope to extend AspectML with an information-flow type and kind system to evaluate how well this solution works in practice.

Acknowledgments

This project is supported by NSF grant 0347289, CAREER: Type-Directed Programming in Object-Oriented Languages. We appreciate the insightful comments by anonymous reviewers and the contributions made by Daniel S. Dantas and David Walker in developing AspectML.

References

- [1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Programming Languages and Systems*, 13(2):237–268, Apr. 1991.
- [2] P. Achten and R. Hinze. Combining generics and dynamics. Technical Report NIII-R0206, Nijmegen Institute for Computing and Information, July 2002.
- [3] R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction (MPC)*, pages 15–17. Springer-Verlag, Marstrand, Sweden, June 1998.
- [4] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 90–104, Pittsburgh, Pennsylvania, Sept. 2002. ACM Press.
- [5] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language (extended version). *ACM Trans. Programming Languages and Systems*. To appear pending revisions. Draft available at <http://www.cis.upenn.edu/~geoffw/research/papers/aspect-toplas.pdf>.
- [6] R. Hinze. Politypic values possess polykinded types. In *Proceedings of the 5th Int. Conf. on Mathematics of Program Construction (MPC)*, pages 2–27, Ponte De Lima, Portugal, July 2000. Springer-Verlag.
- [7] R. Hinze and A. Löb. “Scrap your boilerplate” revolutions. In *Proceedings of 8th Int. Conf. on Mathematics of Program Construction (MPC)*, 2006.
- [8] R. Hinze, A. Löb, and B. C. Oliveira. “Scrap your boilerplate” reloaded. In *Proceedings of the 8th Int. Symp. on Functional and Logic Programming (FLOPS)*, pages 13–29, Fuji Susono, Japan, Apr. 2006.
- [9] R. Hirschfeld. Aspects - aspect-oriented programming with Squeak. In *Revised Papers from the Int. Conf. NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODE)*, pages 216–232, London, UK, 2003. Springer-Verlag.
- [10] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*. To appear.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conf. on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001. Springer-Verlag.
- [12] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Originally in *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*.
- [13] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 244–255, Snow Bird, UT, Sept. 2004. ACM Press.
- [14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 204–215, Tallinn, Estonia, Sept. 2005. ACM Press.
- [15] K. Läuffer and M. Odersky. An extension of ML with first-class abstract types. In *Proceedings of the SIGPLAN Workshop on ML and its Applications*, pages 78–91, June 1992.
- [16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System: Documentation and user's manual, 2000. Available from <http://caml.inria.fr>.
- [17] M. Lippert. AJEER: an AspectJ-enabled Eclipse runtime. In *Companion to the 19th ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 180–181, Vancouver, BC, Canada, Oct. 2004. ACM Press.
- [18] A. Löb, D. Clarke, and J. Jeuring. Dependency-style generic Haskell. In *Proceedings of the 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 141–152, Uppsala, Sweden, Aug. 2003. ACM Press.
- [19] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. AspectualCaml: An aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 320–330, Tallinn, Estonia, Sept. 2005. ACM Press.
- [20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.
- [21] C. Okasaki. From fast exponentiation to square matrices: an adventure in types. In *Proceedings of the 4th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 28–35, Paris, France, Sept. 1999. ACM Press.
- [22] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Trans. Program. Lang. Syst.*, 17(2):264–292, 1995.
- [23] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- [24] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*. To appear.
- [25] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. To appear in *Proceedings of the 11th ACM SIGPLAN Int. Conf. on Functional Programming*, Sept. 2006.
- [26] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
- [27] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information processing*, pages 513–523, 1983.
- [28] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of object behavior using context relations. *SIGSOFT Software Engineering Notes*, 21(6):46–57, 1996.
- [29] M. Sulzmann and M. Wang. Modular generic programming with extensible superclasses. In *Proceedings of the 2006 Workshop on Generic Programming*, Sept. 2006. In this volume.
- [30] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proceedings of the 5th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 82–93, New York, NY, USA, 2000. ACM Press.

- [31] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, Long Beach, CA, USA, Jan. 2005. 15 pages.
- [32] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN Inter. Conf. on Functional Programming (ICFP)*, Uppsala, Sweden, Aug. 2003. ACM Press.
- [33] M. Wang, K. Chen, and S.-C. Khoo. On the pursuit of staticness and coherence. In *Proceedings of the 5th Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2006.
- [34] G. Washburn and S. Weirich. Generalizing parametricity using information flow. In *Proceedings of the 20th IEEE Symp. on Logic in Computer Science (LICS 05)*, pages 62–71, Chicago, IL, June 2005. IEEE Computer Society Press.
- [35] S. Weirich. Higher-order intensional type analysis. In D. L. Métyer, editor, *Proceedings of the 11th European Symp. on Programming (ESOP)*, pages 98–114, Grenoble, France, Apr. 2002.
- [36] S. Weirich. RepLib: A library for derviable type classes. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Portland, OR, Sept. 2006. To appear.
- [37] S. Weirich. Type-safe run-time polytypic programming. *Journal of Functional Programming*, 2006. 30 pages. To appear.
- [38] S. Weirich and L. Huang. A design for type-directed Java. In V. Bono, editor, *Workshop on Object-Oriented Developments (WOOD)*, ENTCS, Aug. 2004. 20 pages.