Weaving Generic Programming and Traversal Performance

Bryan Chadwick
chadwick@ccs.neu.edu

Karl Lieberherr
lieber@ccs.neu.edu

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.

ABSTRACT

Developing complex software requires that functions be implemented over a variety of recursively defined data structures. While the design (and modeling) of structures is itself difficult, complex data can require even more complex functions. In this paper, we introduce a declarative form of traversal-based generic programming that modularizes functions over a structure using function-objects. Our approach is supported by a library and set of generative tools, collectively called DemeterF, which are used to implement modular, adaptive functions. While our traversals support high-level abstractions resulting in modular and extensible functions, we retain genericity, flexibility and performance through traversal generation and inlining.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures, Patterns*

General Terms

Design, Performance

1. INTRODUCTION

The development of complex software requires the implementation of functions over a variety of recursively defined data structures. The design (and modeling) of structures can itself be difficult, but complex data can lead to even more complex functions. In Object-Oriented (OO) Programming Languages like Java and C# the *dominant decomposition* of programs is classes, which encapsulate both structure (data) and behavior (methods). This makes the implementation of certain kinds of operations over a collection of classes difficult without breaking the standard abstraction boundaries in the language. Some operations are easy, though tedious, to implement but are usually scattered throughout classes and require a fair amount of extraneous code dealing with sub-components.

AOSD '10 Rennes, St. Malo, France

In contrast to OO encapsulation, the mantra of Aspect Oriented Programming (AOP) is the *modularization of crosscutting concerns* [21], where we develop abstractions supporting the separation of program aspects that typically overlap. With aspects we can add methods to existing classes and influence a running program by executing code at dynamic join points. While AOP is very useful, it can be tricky to wield its power safely, as computation must typically be performed using mutation (side-effects), making programs difficult to reason about, even informally. This becomes increasingly important when we wish to develop concurrent programs with the advent of implicitly parallel processors.

In this paper we focus on the modularization of a particular group of crosscutting concerns dealing with data structure traversal in OO languages. We present a function-object oriented approach to traversal-based generic programming with the support of libraries and generative tools, collectively called DemeterF. Our contributions can be summarized as follows:

New Approach We introduce a new declarative form of traversal-based generic programming that uses function-objects to fold over data structures (Section 4). Function-objects encapsulate an *aspect* of flexible, extensible, traversal computation, and can be written generically to adapt to different data structures (Section 5). Functions are separated from traversals using an implicit variation of multiple-dispatch that eliminates the scattering of traditional OO methods.

Tool Support Our approach is supported by a class generator that understands Java generics (Section 6). Parametrized classes can be nested to any level, with specific and generic structure-based methods such as field getters, parsers, and printers automatically injected into class definitions.

Retargetability Our traversals support separate control specifications and contexts that allow a purely functional implementation. The elimination of side-effects enables us to replace our reflective, recursive, stack-based traversal with a number of semantically equivalent alternatives, making parallelism, inlining, and other forms of retargeting possible without modifications to function-objects.

Performance Function-objects can be type-checked against data structure definitions, allowing us to weave inlined

traversal and dispatch code (Section 7). Function-objects remain modular and resulting traversals perform as well as hand-written instance methods. Inlined function dispatch that cannot be statically determined results in dispatch residue that performs better than traditional visitors.

Our contribution is a combination of approach and implementation. Traversal-based function-objects support modular, functional, adaptive programming while eliminating some of the problems associated with operational extension in OO languages. The adaptive nature of our traversals introduces flexibility that cannot be checked statically in mainstream programming languages and, if implemented naively, can hinder performance. Our implementation provides a type checker to verify safety and code generation facilities to improve performance. Our functional approach supports fully inlined and implicitly parallel traversals, in many instances achieving execution times better than hand-coded methods. Our traversals support high-level abstractions resulting in modular and extensible functions while retaining flexibility and performance.

2. BACKGROUND

We begin by describing the problem with the help of an interesting example. Consider the definition of an OO picture library, similar to that discussed in [22]. Figure 1 contains Java classes that form the base of our example: the abstract class `Pict` has three subclasses representing `Circle`s, `Square`s, and `Offset` pictures respectively. For reference, all the code examples from this paper are available on the web [10].

```
abstract class Pict{ }

class Circle extends Pict{
  int rad;
  Circle(int r){ rad = r; }
}
class Square extends Pict{
  int size;
  Square(int s){ size = s; }
}
class Offset extends Pict{
  int dx, dy;
  Pict inner;
  Offset(int x, int y, Pict in)
  { dx = x; dy = y; inner = in; }
}
```

Figure 1: Picture Class Skeletons

Our `Pict` classes form a somewhat limited representation of pictures, and we expand them soon, but first let's write a simple `toString()` function, usually referred to as *pretty printing*. This can be somewhat difficult in Java once we separate our classes into different files, since we insert a new method into each class. Figure 2 shows the inserted code with comments describing where each method belongs. If our structures had contained other non-primitive classes we would have to be sure that `toString()` is implemented in them as well, to avoid nonsensical outputs.

Besides the fact that this code is scattered throughout our classes, this simple operational extension illustrates a few other issues that place unnecessary burden on programmers. First, OO class definitions are generally *closed*. In Java this

```
// In Pict
abstract String toString();
// In Circle
String toString(){ return "Circle("+rad+")"; }
// In Square
String toString(){ return "Square("+size+")"; }
// In Offset
String toString(){
  return "Offset("+dx+","+dy+","+
         inner.toString()+")";
}
```

Figure 2: Picture `toString` methods

is especially true for `final` classes and value types, since these cannot be subclassed, though *open classes* are supported by some languages and tools including AspectJ [1], MultiJava [15] and Ruby [7]. Second, our function follows a very typical pattern of recursion that exactly mimics the structure of the classes involved. We should be able to abstract this pattern out, and parametrize over only the interesting parts of our computation. Finally, the implementation of `toString` does not depend on anything outside the class hierarchy. `toString` is, of course, a bit of a special case, but in general, many functions can be written more generically to avoid mentioning certain details, allowing them to adapt to structural changes without programmer intervention.

This can't be the whole story though, because OO programmers rely on extensible data structures: adding cooperating functions/methods to a collection of classes may be difficult, but adding a new subclass to extend our data types is comparatively easy. This function/data centric dilemma is usually known as the *expression problem* [40, 2]. For instance, we can add a new picture subclass to represent compositions. Figure 3 contains a class, `Overlay`, that represents an overlaying of two pictures.

```
class Overlay extends Pict{
  Pict top, bot;
  Overlay(Pict t, Pict b){ top = t; bot = b; }

  String toString(){
    return "Overlay("+top.toString()+","+
                      bot.toString()+")";
  }
}
```

Figure 3: Overlay picture extension

This brings us to a crossroads: if we use a function-centric approach (like visitors), then adding to our data types is difficult, but if we use a data-centric (OO) approach then adding functions is difficult. Many abandon the function centric approach believing that it is unsafe (*e.g.*, casting [22]) or hinders performance (*e.g.*, reflection [37]). In either case we run into problems similar to those above, but is it possible to have the best of both worlds, modularizing functions while remaining general, safe, and efficient?

3. OUR SOLUTION

Our answer to this question is *yes*. We solve these problems using a traversal-based approach that encapsulates functions over a data structure into *function-objects*: instances of classes that wrap a set of methods. For our original col-

lection of picture classes, Figure 4 shows a function-class that implements `toString` using our DemeterF traversal library. Our function-class, `ToString`, extends the DemeterF class `ID`, which provides identity `combine` methods for Java's primitive types; to that we add three methods to handle the traversal of our hierarchy. In order to understand the computation involved, we simply need to *think* like a traversal. In contrast to the popular Law of Demeter: "*Talk only to your friends*", we prefer the slogan "*Listen only to your neighbors*", where neighbors refers to an object's parents and children.

```
class ToString extends ID{
  String combine(Circle c, int r)
  { return "Circle("+r+")"; }
  String combine(Square s, int sz)
  { return "Square("+sz+")"; }
  String combine(Offset o, int dx, int dy, String in)
  { return "Offset("+dx+","+dy+","+in+")"; }

  String toString(Pict p)
  { return new Traversal(this).traverse(p); }
}
```

Figure 4: `ToString` using DemeterF

In this case, a generic `Traversal` is created in the `toString` method given `this` function-object, an instance of `ToString`. Our default `Traversal` implementation reflects on the given function-object to determine the structures to which it is applicable (`Circle`s, `Square`s, `Offset`s, and primitives). The `traverse` method then recursively walks the structure of the given picture, depth-first, using reflection.

When traversal of a `Circle` or `Square` is complete, the instance and the recursive result of its field are passed to the matching `combine` method (`(Circle,int)` or `(Square,int)` respectively). Because `ToString` does not add `combine` methods to handle `int`, traversal of a `Circle` or `Square` represents an unfolding, with method parameters `r` and `sz` representing instance variables `rad` and `size`. The same is done when traversal reaches an `Offset`: the recursive field (`inner`) is traversed before the `combine` method is selected and called. The `String` resulting from the traversal of `inner` is computed and passed with other fields (`dx` and `dy`) to the matching method for `Offset`.

Our `toString` functionality is now nicely modularized in a single function-class, which can be considered an *aspect* of traversal computation. An added benefit of using function-*objects* with a reflective traversal is that extending user function-classes is no different than extending data types: when our picture classes are extended with `Overlay`, we can subclass `ToString` to handle the new case. The resulting extension is shown in Figure 5. At the top level we need to use our new `ToStringOverlay` for `Pict`s that may contain overlays, but we eliminate the need for casting found in other functional visitor solutions [22].

```
class ToStringOverlay extends ToString{
  String combine(Overlay o, String t, String b)
  { return "Overlay("+t+","+b+")"; }
}
```

Figure 5: `ToString` extended for `Overlay`

An easier way of creating this particular print-like function is to use the structure of our picture classes, and DemeterF, to generate the function-class automatically. The generator component of DemeterF accepts a textual representation of class structures called a *class dictionary* (CD) [26], which looks like a mix of BNF and algebraic data types, similar to those found in Haskell [20] and ML [31]. While DemeterF includes a reflective tool to infer CDs from hand-written Java classes, instead, we typically write CDs first, in order to generate source files. The CD for our `Pict` classes appears in Figure 6.

```
// pict.cd
Pict    = Circle | Square | Offset | Overlay.
Circle  = <rad> int.
Square  = <size> int.
Offset  = <dx> int <dy> int <inner> Pict.
Overlay= <top> Pict <bot> Pict.
```

Figure 6: Class Dictionary for `Pict`s

Our abstract class `Pict` is described by a list of variants separated by bars (|), while concrete classes list their field names (in brackets) and types.[1] Because a generic form of `ToString` is included with DemeterF, our CD can be used to generate `Pict` structures that automatically include a `toString` method:

```
>% java demeterf pict.cd --dgp:ToString
```

The `dgp` stands for *data-generic programming* [17, 28, 19], and the function-class that DemeterF generates for `ToString` is almost exactly the same as what we wrote by hand, but it can be generated for *any* CD. We can also generate other functions, like `parse`, `equals`, and `hashCode`, or write our own plug-in DGP class, but the most important generic function that can be generated is *traversal* itself. Once we have an implementation of `ToString` (generated or hand-written), we can use the CD together with the function-class definition to weave a specialized *inlined* traversal.

We use our type checker to calculate the return type of each sub-traversal and generate code that traverses each field of our classes and merges results by calling the appropriate matching `combine` method (in this case there are no overlapping methods). The result is a semantically equivalent `Traversal` class, that performs much better than our generic, reflective implementation. In many cases our inlined code can actually outperform hand-written instance methods. Figure 7 gives average performance numbers of four different implementations of `ToString` run 10 times on a very large `Pict` instance. The results are collected using *Sun's JDK 6* on a 2.2 Ghz Intel Core 2 Duo processor. The first is the DemeterF inlined version; the second is hand-coded methods from Figure 2; the third is a hand-written visitor implemented using double-dispatch; and the final one, for comparison, uses our DemeterF reflective `Traversal` implementation.

In the rest of this paper we provide the details of our traversal-based approach, and how generic programming (Section 5) and generative weaving (Sections 6 and 7) combine to provide modularity, flexibility, and performance.

[1] In fact, a CD can describe any Java class hierarchy, though we won't discuss all of our CD features in this paper.

Type	*Average Time*
INLINED	48 ms
HAND	49 ms
VISITOR	54 ms
REFLECTIVE	362 ms

Figure 7: Performance of `ToString`

4. DEMETERF TRAVERSALS

The traversal of data structures can be thought of simply as a walk over a structure that performs some work. In our case, we package our work into a function-object and let the traversal handle the selection of methods and passing of parameters (*i.e.*, *implicit invocation*). In this section we provide an overview of our traversal approach and use it as a basis for implementing different kinds of functions.

4.1 Functions to Traversals

Going back to our `Pict` structures, let's write a slightly simpler function over pictures that counts the number of `Circles` it contains. Again, we add a new method to each class, shown in Figure 8.

```
// In Pict
abstract int circles();
// In Circle
int circles(){ return 1; }
// In Square
int circles(){ return 0; }
// In Offset
int circles(){ return inner.circles(); }
// In Overlay
int circles()
{ return top.circles()+bot.circles(); }
```

Figure 8: Picture `circles` methods

Together the functions implement straight-forward structural recursion: at each point where the structure is recursive, the function is also recursive. Similar to folds, typical functional visitor approaches [8, 33] implement this sort of computation using methods that essentially replace the constructors of concrete variants. If we added the correct scaffolding for picture `Visitors`, a visitor-based function could look something like Figure 9.

```
class CircsVis extends Visitor<Integer>{
  Integer visit(Circle c){ return 1; }
  Integer visit(Square s){ return 0; }
  Integer visit(Offset o)
  { return o.inner.accept(this); }
  Integer visit(Overlay o)
  { return o.top.accept(this)+o.bot.accept(this);}
}
```

Figure 9: `circles` Visitor implementation

In order to abstract out the traversal, in DemeterF we pass the recursive (sub-)traversal results from an object's fields after the original object itself. This makes `combine` method selection/matching uniform, using a variant of type-based multiple dispatch. For example, the implementation of circles using DemeterF is shown in Figure 10.

The hand-coded, visitor, and DemeterF functions look quite similar, the major difference being that in the DemeterF

```
class Circs extends ID{
  int combine(Circle c, int rad){ return 1; }
  int combine(Square s, int siz){ return 0; }
  int combine(Offset o, int x, int y, int inCs)
  { return inCs; }
  int combine(Overlay o, int topCs, int botCs)
  { return topCs+botCs; }
}
```

Figure 10: `circles` DemeterF implementation

case, the recursion is done for us implicitly: the arguments to the combine methods have already been traversed *before* the `combine` method is called (similar to *internal* visitors [8], where the data structure is responsible for traversal). Moreover, the interesting computation involved is precisely encapsulated in our function-class, with all the boilerplate code left to the traversal implementation. As with visitors, creating or extending functions over our data structures is rather simple. For example, consider implementing a function `squares` that counts the number of `Square`s in a given picture. The DemeterF version is shown in Figure 11; since our computation is succinctly written, the more abstract, adaptive traversal provides a platform for reuse.

```
class Squares extends Circs{
  int combine(Circle c, int rad){ return 0; }
  int combine(Square s, int siz){ return 1; }
}
```

Figure 11: `squares` implementation using `Circs`

4.2 Traversal Details

The idea of abstraction is to eliminate similarities by parametrizing over differences. When abstracting traversal from computation we use a depth-first approach that treats all values as objects, *i.e.*, primitives are treated as objects without any fields. Our basic strategy is illustrated with an example traversal method for `Overlay`:

```
ID func;
<Ret,P> Ret traverse(Overlay o){
  P top = traverse(o.top);
  P bot = traverse(o.bot);
  return func.combine(o, top, bot);
}
```

The traverse method is parametrized over the types returned by the traversal of an `Overlay` and a `Pict` (`Ret` and `P` respectively). In general traversal methods of this form cannot be implicitly type checked by Java, but they suffice to show our interpretation of structural recursion: each field is traversed in turn, and the results are passed (along with the original object) to the function-object's `combine` method. The type parameters (`Ret` and `P`) signify that the traversal of different types may return different results. In this case, both `top` and `bot` are `Pict`s, so their traversals must return a common (unified) type, but multiple or mutually recursive hierarchies can be handled similarly. For primitive types and user defined classes without fields the traversal just delegates to the function-object, since sub-traversals are not needed.

```
<Ret> Ret traverse(int i){
  return func.combine(i);
}
```

Though these `traverse` methods illustrate our point, in DemeterF the `combine` method chosen by the traversal is

based on the dynamic types of all arguments, including the function-object itself. We choose the *most specific* `combine` method that is applicable to the given arguments: the current object and sub-traversal results. Our dispatch strategy is termed *asymmetric*, as we resolve ambiguities using the left-most differing argument position, similar to a lexicographical ordering using `extends` (or *subclass-of*) as less-than.

4.3 Control

Traversal that implements structural recursion everywhere throughout an object is very useful, but other strategies are certainly needed. DemeterF provides several types of control (*where to go*) as a separate aspect of the traversal. Our control is a limited form of traversal *strategies* [26, 27], based on the notion of *bypassing*. For example, consider implementing a function `topMost` that returns the top most primitive (`Circle` or `Square`) in a given picture. A traversal everywhere would be inefficient, but there's no need to hand-code the entire traversal. Instead we can bypass (or skip) the `bot` field of all `Overlay` instances; our implementation is shown in Figure 12.

```
class TopMost extends ID{
  Pict combine(Pict p, int i){ return p; }
  Pict combine(Offset o, int x, int y, Pict in)
  { return in; }
  Pict combine(Overlay o, Pict top, Pict bot)
  { return top; }

  Pict topMost(Pict s){
    return Traversal(this,
      Control.bypass("Overlay.bot")).traverse(s);
  }
}
```

Figure 12: `TopMost` using bypassing

`Control.bypass` accepts a string representing a list of fields to be bypassed, just `Overlay.bot` in this case, and returns a `Control` instance that is used by the `Traversal` to determine which parts of the structure should be traversed. The `combine` methods look similar to those before, but the overall function becomes more efficient by eliminating unneeded traversal. When an `Overlay` instance is reached the traversal will not recur on its `bot` field, instead the untouched field is passed to the matching combine method along with the recursive result from `top`. Our asymmetric multiple dispatch also allows us to abstract over multiple method cases; here the circle and square methods are abstracted into a single `combine` over `Pict`.

One form of control that is particularly useful is the special case `onestep`, corresponding to the bypassing of *all* fields. This allows programmers to efficiently implement a traversal style closer to hand-coded recursion, but leaving the type checks, casting, and call-backs to the traversal implementation. Figure 13 shows a function-class that returns the bottom most primitive picture, using a one-step traversal.

Rather than letting the traversal implicitly control our path through a picture, we make the recursive calls ourselves, one step at a time. `Traversal.onestep` returns a traversal that steps into an object and passes its fields to the function-object's matching combine method. This allows a programmer to implement more ad hoc recursion schemes,

```
class BotMost extends ID{
  Pict combine(Pict p, int i){ return p; }
  Pict combine(Offset o, int x, int y, Pict in)
  { return botMost(in); }
  Pict combine(Overlay o, Pict top, Pict bot)
  { return botMost(bot); }

  Pict botMost(Pict s){
    return Traversal.onestep(this).traverse(s);
  }
}
```

Figure 13: `BotMost` using a `onestep` `Traversal`

resulting in functionality similar to a more traditional visitor solution, achieving modular functionality without unneeded boilerplate or scaffolding.

4.4 Contexts

Traditional visitors [16] employ `void` visit methods to encapsulate computations over structures, which forces programmers to use mutation in order to communicate values between different calls. In DemeterF we have designed our traversal approach to eliminate side-effects in order to make programs compositional and simpler to optimize and parallelize, but this can limit the communication of context sensitive (top-down) information over a structure. To facilitate the passing and updating of information from a parent to a child as a separate aspect, DemeterF supports the idea of a *traversal context*. The traversal manages the context by propagating it, passing it to combine methods, and updating its value based on the function-object's declared methods.

The initial (root) context is given as an extra argument to `traverse`, and the function-object can declare `update` methods to produce a new context for children/fields of an object being traversed. The context is then passed as an optional last argument to a chosen combine method. For example, if we attempt to generate a visual representation of a `Pict` object using DemeterF, we notice that information gets lost during traversal; an `Offset` instance contains positioning information for its children. Using traversal contexts we can easily encapsulate this information into a drawing context. A simple representation is shown in Figure 14.

```
class Ctx{
  int x, y;
  Ctx(int xx, int yy){ x = xx; y = yy; }
  Ctx move(int dx, int dy)
  { return new Ctx(x+dx, y+dy); }
}
```

Figure 14: Drawing contexts

We can now implement a function to convert a `Pict` into a *Scalable Vector Graphics* (SVG) string. SVG is a popular XML format for representing visual elements that is portable and simple to generate. Figure 15 shows a function-class that implements the `Pict` conversion to SVG using our drawing context, `Ctx`. The `SVG` class encapsulates static methods that create the SVG specific formatting. The first four `combine` methods are very similar to what we have written before, except that the methods for `Circle` and `Square` include a third parameter of type `Ctx`.

When `traverse` is called we pass an initial context representing the center of the canvas, $(w/2, h/2)$. Before recursively

```
class ToSVG extends ID{
  String combine(Circle c, int r, Ctx ctx)
  { return SVG.circle(ctx.x, ctx.y, r); }
  String combine(Square s, int sz, Ctx ctx)
  { return SVG.square(ctx.x, ctx.y, sz); }
  String combine(Offset o,int dx,int dy,String in)
  { return in; }
  String combine(Overlay o, String t, String b)
  { return t+b; }

  Ctx update(Offset off, Fields.any f, Ctx c)
  { return c.move(off.getDx(), off.getDy()); }

  String toSVG(Pict p, int w, int h){
    return SVG.head(w, h)+
           new Traversal(this)
             .traverse(p, new Ctx(w/2,h/2))+
           SVG.foot();
  }
}
```

Figure 15: `Pict` to SVG using Contexts

traversing the fields of an `Offset`, the traversal will call our `update` method to produce a new context. The signature of the `update` method can be read as: *Before traversing any field of an* `Offset`*, compute a new context from the parent's.* In this case we `move` the context to include the current `Offset`. If no matching `update` method is found, then the parent's context is passed recursively to each sub-traversal unchanged. In this case, the update method's second parameter type, `Fields.any`, corresponds to a DemeterF class representing all fields. Alternatively, we can create representative field classes (*e.g.*, `Offset.inner`) to allow more fine grained context updates; classes generated using DemeterF include appropriately named inner classes that are used for this purpose.

5. GENERIC PROGRAMMING

We call the programming style of DemeterF *generic* because it generalizes the shape of the data types being traversed: functions do not necessarily rely on the specific types or names of an object's fields, but on the return types of the traversal of those fields.[2] For instance, in the `ToString` function-class (Figure 4), the traversal of an instance of `Pict` returns a `String`. Our function-class relies on this, and the fact that the traversal of an integer will return an integer.

Abstracting from the typical uses of function-classes leads us to two general cases: those which are *type unifying*, and those that are *type preserving* [25], sometimes referred to as *queries* and *transformations* [23]. The first category contains functions similar to `ToString` and `Circs`, where sub-traversals return the same type, and recursive results are combined using a single function, *e.g.*, `String` or `int` combined using `+`. The second category contains certain kinds of transformations and functional updates, where we may change interesting parts of a data structure, while reconstructing (or *copying*) others.

5.1 Type-Unifying Functions

To support generic type-unifying traversals in DemeterF we provide a special function-class that abstracts computation using two methods: a no argument `combine` method that provides a default case, and a two argument `fold` method that is used to fold together multiple recursive results into a single value. The skeleton of the `TU` class is shown in Figure 16.

```
abstract class TU<X> extends ID{
  abstract X combine();
  abstract X fold(X a, X b);

  X traverse(Object o){ /* ... */ }
}
```

Figure 16: Type-unifying function-class

How can we use this class? Figure 17 contains a new definition of our `Circs` function-class (from Figure 10) that counts the `Circle`s in a `Pict`. The first two methods implement our necessary abstract methods of `TU`, providing a default `combine`, and a `fold` that sums the resulting counts.[3] The final method describes the interesting part of our structure, `Circle`s, where we return 1.

```
class CircsTU extends TU<Integer>{
  Integer combine(){ return 0; }
  Integer fold(Integer a, Integer b){ return a+b;}

  Integer combine(Circle c){ return 1; }
}
```

Figure 17: Generic `circles` count using `TU`

In our experience, `TU` is most useful for computations that collect information over a complex data structure. This usually involves some form of library structure to collect instances (*e.g.*, `List`, `Set`, *etc.*). Figure 18 shows a typical use of `TU` with DemeterF `List`s to collect results over a generic structure. Note that we use DemeterF functional (immutable) `List`s, so `append` returns a new instance.

```
abstract class ListTU<X> extends TU<List<X>>{
  List<X> combine(){ return List.create(); }
  List<X> fold(List<X> a, List<X> b)
  { return a.append(b); }
}
```

Figure 18: Typical TU collection using lists

5.2 Type-Preserving Functions

While `TU` functions collect various results of a single type together, type-preserving functions perform recursive *transformations* over the traversal of a data structure. The basic idea is easily demonstrated by writing a *copy* function for our picture classes, shown in Figure 19.

```
class Copy extends ID{
  Circle combine(Circle c, int r)
  { return new Circle(r); }
  Square combine(Square s, int sz)
  { return new Square(sz); }
  Offset combine(Offset o,int dx,int dy, Pict in)
  { return new Offset(dx,dy,in); }
  Overlay combine(Overlay o, Pict t, Pict b)
  { return new Overlay(t,b); }
}
```

Figure 19: Copy function-class for `Picts`

[2] You could say our function-objects are *near-sighted*.

[3] Java requires that we use `Integer`, a reference type, rather than `int`, value type, though coercion is usually automatic.

For each concrete `Pict` subclass we write a combine method that has parameters with the same types as its fields and constructs a new instance with recursive results. While `Copy` is specific to `Pict`s, the completely generic version of this function is implemented in the DemeterF class `TP`, which uses reflection to dynamically call constructors of the object being traversed. We can extend the generic function with specific combine methods to implement our required transformation. Figure 20 shows a function-class that recursively scales a picture by a given factor. This function-class is completely generic and applicable to *any* data structure, though in this case we only apply it to `Pict`s to preserve its "scale" meaning.

```
class Scale extends TP{
  int scl;
  Scale(int s){ scl = s; }
  int combine(int i){ return i*scl; }

  Pict scale(Pict p)
  { return new Traversal(this).traverse(p); }
}
```

Figure 20: `Scale` transformation for `Pict`s

The benefit here is that we mention as little of our structure as possible; we only write methods for the interesting parts to be transformed. As another example, Figure 21 shows a function-class that converts all the `Circle`s in `Pict` instance into `Square`s of the same size. We only refer to the important structural elements, namely that `Circle` contains an `int` radius, or more precisely, something for which our traversal will return an `int`.

```
class Circ2Sqr extends TP{
  Square combine(Circle c, int rad)
  { return new Square(rad*2); }
}
```

Figure 21: Convert circles into squares

As a final `TP` example, Figure 22 shows a function-class that reverses the top to bottom ordering of a `Pict` instance. This example emphasizes the fact that the arguments passed to the combine method are the recursive results of applying our function-object over the traversal. The `t` and `b` arguments to our `combine` method have already been `Flip`ed once it is called.

```
class Flip extends TP{
  Overlay combine(Overlay o, Pict t, Pict b)
  { return new Overlay(b, t); }
}
```

Figure 22: Reverse top to bottom `Pict` ordering

6. DEMETERF GENERATION

DemeterF is not only a traversal library, but also a tool, similar to DemeterJ [39], for developing, generating, and managing complex class hierarchies. What we add to DemeterJ is extensive support for Java generics and parametrized classes, and a framework for data-generic programming. DemeterF class dictionary (CD) files share much of their syntax with DemeterJ. Though we include several new features for parametrized classes, our behavior (BEH) files are a simplification of DemeterJ's, supporting only a static form of open classes.

A CD file consists of class definitions describing the structure of our classes. Each definition can contain any number of subclasses, separated by bars (|), followed by field definitions and/or concrete syntax terminated by a period. For example, typical LISP style structures for a list-of-integers could be described by a CD such as:

```
// list.cd
 List = Cons | Empty.
 Cons = <head> int <tail> List.
Empty = ";".
```

This defines an abstract class `List` (since it has a non-empty list of subclass) and two concrete classes, `Cons` and `Empty`. In addition to Java class definitions, DemeterF also generates a parser using CD definitions as a modified form of BNF. Of course, specific `List` definitions are not as useful as one that is abstracted over the type of data that it stores:

```
 List(X) = Cons(X) | Empty(X).
 Cons(X) = <head> X <tail> List(X).
Empty(X) = ";".
```

Once we fully instantiate this parametrized definition, DemeterF will generate specific methods for parsing, printing, and traversing that particular instantiation.

```
IntList = <lst> List(Integer).
```

The added benefit is that we can nest our instances, and even parametrize over a new type parameter, letting DemeterF handle the complicated details.

```
D2List(Y) = <lst> List(List(Y)).
```

6.1 Data-Generic Programming

Creating more concrete versions of reflection-based DemeterF classes like `Traversal`, `TU`, and `TP` only depends on the specific structures involved. The key to overcoming performance limitations of dynamic reflection is to replace it with static information from a CD. DemeterF supports two forms of data-generic programming over the structure of data types allowing the injection of methods *per-class*, or *per-CD*. Methods like *getters* and `equals` depend only locally on a single class definition, while others like parsing, printing, and traversal rely a global view of the CD. Both forms of generation are specified as a function-class: instances of the classes chosen on the command-line are used to traverse a portion of the CD (either a single class definition, or the entire CD) and generate the necessary code.

A typical DemeterF command line looks something like the following:

```
>% java demeterf list.cd --dgp:Print --pcdgp:Getters
```

Which requests that a `print` method be generated (pretty-printing based on CD syntax) and getters (like `getHead()`) be generated for all fields.

Static classes corresponding to `TU` and `TP`, named `StaticTU` and `StaticTP`, can be generated by including them in the command-line's `dgp` list, since they require information from the entire CD. For `StaticTP`, the result is very similar to our `Copy` function from Figure 19, while `StaticTU` contains inlined calls to `TU`'s `fold` for concrete class cases. The main reason for generating these function-classes is to create type-safe versions for precise inlining and improved performance.

7. INLINING AND PERFORMANCE

Types play a central role in DemeterF traversals, both in guiding the traversal of data structures and in the selection of `combine` methods. In order for a traversal to be safe we must be sure that the selected methods over the traversal fit together correctly. An added benefit of doing this safety analysis statically is that with traversal return types in hand, we can eliminate most the overhead of multiple dispatch by generating a CD specific traversal with inlined calls to `combine` methods.

7.1 Types

DemeterF function-classes are Java classes and must conform to Java's typing rules, but things get more interesting when we interpret `combine` methods as a coordinated function over a data structure traversal. For example, consider our `Circs` function-class (Figure 10): each method returns an `int`. This means that the traversal of each subclass of `Pict` must also return an `int`. Using the CD from Figure 6 we can check that each `combine` method has the right types and number of arguments and is applicable to the expected field result types. A quick walk over the definitions in the CD tells us how many arguments to expect, and the `combine` methods tells us what types the recursive traversal will return for each. The type checker's goal is then to prove that an applicable `combine` method always exists during dispatch. The type-unifying case described above generalizes for other functions, including `TP` transformations like `Copy` (Figure 19) and more ad hoc functions like `Circ2Sqr` (Figure 21).

The basis of our type system has been formalized [12, 11] with a more algorithmic discussion here [9]. The most important cases deal with recursive types: when a recursive type use (field) is encountered, there is no way to know immediately what type the traversal will yield. As an approximation we assume that it could be *anything*, and constrain the return type based on the arguments of otherwise matching `combine` methods. For instance, the field `inner` of `Offset` is a recursive use of `Pict`; when calculating the type returned by the traversal of an `Offset`, we know that the traversal of the first two parameters is `int`, but the third is unknown. We instead look for any `combine` applicable to:

```
(Offset, int, int, *)
```

If such a method exists, then the traversal return type can be constrained based on the argument type(s) in the recursive positions. For `Offset` a constraint originates from the forth argument: when checking `Circs` it constrains the traversal of a `Pict` to return an `int`, whereas for `Copy` the traversal must return a `Pict`.

In some cases there may be more than one applicable method, which simply results in multiple constraints. For example, consider a function-class `Compress` in Figure 23, that recursively replaces nested `Offset`s with a single instance.

```
class Compress extends Copy{
  Offset combine(Offset o, int x,int y, Offset in)
  { return new Offset(in.dx+x,in.dy+y, in.inner);}
}
```

Figure 23: Compress redundant Offsets

Our function-class contains two methods that may be applied after traversing an Offset: one inherited from `Copy` and one implemented in `Compress`, which differ by their last argument. When constraining the recursive field, we choose the common supertype of `Pict` and `Offset`, `Pict`. For the traversal of abstract classes like `Pict`, the return type is a common supertype of the return types of the subclass traversals.

7.2 Inlining

If the `combine` methods satisfy all constraints, we can calculate the `combine` methods that might be called at each point during traversal. To generate a specialized traversal we insert calls to the correct `combine` method(s) at each point, weaving code to dynamically resolve method selections when needed. For example, when inlining `Compress`, after completing an `Offset` the traversal is left with a choice between two methods. The method chosen depends on the dynamic type of the recursive result for `inner`, so the DemeterF inliner produces code to disambiguate the methods like the following:

```
if(inner instanceof Offset)
  return func.combine(o, dx, dx,(Offset)inner);
return func.combine(o, dx, dy, inner)
```

Where `inner` is the recursive traversal result. Our implementation uses a algorithm to recursively partition the set of possible methods, generating `if` statements to narrow the selections until all arguments are exhausted, or only one method is applicable.

7.3 Performance

Similar to partial evaluation, the main motivation for weaving traversals is to improve performance. As a thorough performance test, we have implemented each of the functions described previously three different ways: using DemeterF function-classes, hand written instance methods, and double-dispatch visitors. Figure 24 contains the results of running each implementation of the functions on a large generated `Pict` instance. We use *Sun's JDK 6* on a 2.2 Ghz Intel Core 2 Duo processor; each time is an average of 10 runs, on a picture with approximately 80,000 nodes.

The first row of the table shows DemeterF inlined traversal results, the second is hand-coded instance methods, and the third is a double-dispatch visitor implementation. For a base comparison, the final row shows the DemeterF reflective traversal implementation, with the same function-class used for both inlined and reflective traversals. DemeterF inlined traversal performance is comparable to the hand-coded versions, actually doing better on most functions. The inlined `CircsTU` traversal has a reasonable amount of overhead due to internal method calls, but inlined `TP` based functions perform very well, without the need to write any traversal code by hand.

8. EXAMPLE: EXPRESSION COMPILER

As a more sophisticated example using DemeterF, in this section we discuss the implementation of a compiler for a simple expression language. We write function-classes to calculate the maximum local variable usage, simplify constant expressions, and convert our arithmetic language into a low level stack-based assembly language, similar to that of the Java Virtual Machine. To keep things interesting,

Type	CircsTU	ToSVG	Scale	Circ2sqr	Flip	Compress
INLINE	18 ms	489 ms	11 ms	11 ms	10 ms	11 ms
HAND	9 ms	488 ms	20 ms	19 ms	13 ms	13 ms
VISITOR	47 ms	491 ms	63 ms	62 ms	59 ms	86 ms
REFLECTIVE	651 ms	15618 ms	648 ms	645 ms	650 ms	617 ms

Figure 24: Performance of `Pict` function implementations

our source language includes arithmetic expressions, variable definitions, if expressions, and binary operations.

8.1 Structures

To build a compiler we need representations for both our source and target languages. In this case, the abstract and concrete syntax of both languages can be described with a couple CDs. Figure 25 shows a CD that defines our target language: a simple stack based assembly language with labels, subtraction, and operations for manipulating control, stack, and definitions.

```
// asm.cd
Op = Minus | Push  | Pop | Define | Undef
   | Load  | Label | Jmp | IfNZ.

Minus = "minus".
 Push = "push" <i> int.
  Pop = "pop".
Define = "def".
 Undef = "undef".
 Load = "load"  <i> int.
 Label = "label" <id> ident.
  Jmp = "jump"  <id> ident.
 IfNZ = "ifnzero" <id> ident.
```

Figure 25: Assembly structures CD

We do not show the code associated with the assembly structures, but the full code for all the examples in the paper is available on the web [10]. Figure 26 shows a CD file that describes our expression data structures.

```
// exp.cd
Exp = Ifz | Def | Bin | Var | Num.
Ifz = "ifz" <cnd> Exp "then" <thn> Exp
                      "else" <els> Exp.
Def = <id> ident "=" <e> Exp ";" <body> Exp.
Bin = "(" <op> Oper <left> Exp <right> Exp ")".
Var = <id> ident.
Num = <val> int.

Oper = Sub.
 Sub = "-".
```

Figure 26: Expression structures CD

The command we would use to generate the class definitions for our expression structures is shown below:

```
>% java DemeterF exp.cd --dgp:Print:StaticTU:StaticTP
```

A similar command would be used for the assembly structures; DemeterF uses the dgp functions to generate `print` and `parse` methods, and static versions of our generic function-classes. A simple term in this concrete expression syntax would look something like:

```
ifz (- 4 3) then 5 else 7
```

and can be parsed with the Java statement below:

```
Exp e = Exp.parse("ifz (- 4 3) then 5 else 7");
```

though for our compiler implementation we will parse expressions from a file.

8.2 Max Environment Size

A typical operation needed when compiling languages with local definitions is to calculate the maximum number of variables used by a procedure. This allows the runtime to allocate the right amount of space for procedure frames and verify that `Load` instructions are always in bounds. Figure 27 shows a function-class that calculates the maximum local definition nesting for an expression.

```
class MaxEnv extends StaticTU<Integer>{
  Integer combine(){ return 0; }
  Integer fold(Integer a, Integer b)
  { return Math.max(a,b); }

  Integer combine(Def c, int id, int e, int b)
  { return fold(e, 1+b); }
}
```

Figure 27: Maximum local environment calculation.

Variables are bound by `Def`s, so we return the maximum of (`body`+1) and the result from the binding expression. Our superclass, `StaticTU`, handles other cases like `Num` and `Bin`, so we can eventually generate inlined traversals.

8.3 Simplification

As a second example, Figure 28 shows a function-class that implements bottom up simplification of constant expressions in our mini language. We extend the generated class `StaticTP`, so we can efficiently inline the function-class later.

```
class Simplify extends StaticTP{
  class Zero extends Num{ Zero(){ super(0); } }
  Num combine(Num n,int i)
  { return (i==0) ? new Zero() : new Num(i); }

  Exp combine(Bin b,Sub p, Exp l, Zero r){return l;}
  Exp combine(Bin b,Sub p, Num l, Num r)
  { return combine(l, l.val-r.val); }

  Exp combine(Ifz f,Zero z, Exp t, Exp e){return t;}
  Exp combine(Ifz f,Num n, Exp t, Exp e){return e; }

  Exp combine(Def d,ident i, Exp e, Num b){return b;}
}
```

Figure 28: Recursive Simplification

The special cases for arithmetic expressions are each captured by our `combine` methods, while the rest of the reconstruction is handled implicitly by `StaticTP`. `Num` instances that contain zero are transformed into instances of the more specific inner class `Zero`. Subtracting `Zero` from any `Exp` yields just the left `Exp`; for subtraction consisting of only numbers we can propagate the resulting constant as a new `Num`. For `Ifz` expressions, when the condition is `Zero` or `Num`

we can simplify by returning the recursive result from `thn` or `els`, respectively. Finally, for definitions involving only numbers, we safely discard the binding.

8.4 The `Exp` Compiler

For the sake of code organization and modularity, we have split the final example into four classes; one class for each category of expression and a main, top-level entry-point. Figure 29 shows the main compiler class, `Compile`, that extends our final function-class, `Cond`.

```
// Compile an Exp File
class Compile extends Cond{
  List<Op> compile(String file){
    Exp e = Exp.parse(new FileInputStream(file));
    return new Traversal(this)
        .traverse(e, List.<ident>create());
  }
}
```

Figure 29: Main compile class

We have a single method, `compile(String)`, that parses an expression from the given file, and traverses it to produce a list of representative opcodes, `List<Op>`. When compiling, we use the traversal context to pass a stack of local variable names (`List<ident>`) for nested definitions, starting with the empty `List` as our root context. Here `List` is a functional (immutable) list implementation provided by the DemeterF library with typical methods like `create`, `append`, and `lookup`. DemeterF library/container classes such as `List` are also described by a CD file, with definitions similar to those in Section 6; our generative/weaving approach applies equally well to external and parametrized classes.

```
class Arith extends ID{
  List<Op> one(Op o){ return List.<Op>create(o); }

  List<Op> combine(Sub s){return one(new Minus());}
  List<Op> combine(Num n, int i)
  { return one(new Push(i)); }
  List<Op> combine(Bin b, List<Op> o, List<Op> l,
                                      List<Op> r)
  { return r.append(l).append(o); }
}
```

Figure 30: Compile for arithmetic `Op`s

Figure 30 shows the `combine` methods for math related operators in our expression language. The method `one(..)` simplifies the creation of single `Op` lists. As is common in stack based assembly languages we push operands onto the stack, then call an arithmetic operator. For instance, the expression `(- 4 3)` would generate the following instruction sequence:

```
push 3
push 4
minus
```

The `Defs` class in Figure 31 implements the compilation of variable definition related expressions, extending our `Arith` compiler. We generate a `Load` operation for a variable reference with the offset of its identifier in the environment, which is passed as the last argument to the `combine` method. Our `update` method adds a defined variable to the environment when traversing into the `body` of a definition, signified by the use of the field class (`Def.body`), generated by DemeterF. Once all sub-expressions have been compiled, the `body` code is wrapped in `Define`/`Undef` and appended to the binding evaluation code.

```
class Defs extends Arith{
  List<ident> update(Def d, Def.body f,List<ident> s)
  { return s.push(d.id); }
  List<Op> combine(Var v, ident id, List<ident> s)
  { return one(new Load(s.index(id))); }

  List<Op> combine(Def d, ident id, List<Op> e,
                                    List<Op> bdy){
    return e.append(new Define()).append(bdy)
            .append(new Undef());
  }
}
```

Figure 31: Compile for Variables

The final class, `Cond` shown in Figure 32, deals with conditional expressions, extending `Defs`. We use local variable (`lnum`) in the method `fresh()` to create unique `Label`s within generated code. The `IfNZ` opcode is used to branch to the `els` portion when the condition is not zero. Otherwise the code produced for `thn` will be executed, and finally we `Jmp` to the *done* label.

```
class Cond extends Defs{
  int lnum = 0;
  synchronized ident fresh(String s)
  { return new ident(s+"_"+lnum++); }

  List<Op> combine(Ifz f, List<Op> c, List<Op> t,
                                      List<Op> e){
    ident le = fresh("else"),
          ld = fresh("done");
    return c.append(new IfNZ(le)).append(t)
            .append(new Jmp(ld))
            .append(new Label(le)).append(e)
            .append(new Label(ld));
  }
}
```

Figure 32: Compile for Conditionals

8.5 Performance

To demonstrate the performance of DemeterF inlined traversals, we give timing results for three equivalent implementations of each of the functions, `MaxEnv`, `Simplify`, and `Compile`. Figure 33 contains the average results of 10 runs of each on a very large `Exp` instance with the same configuration used previously. DemeterF inlined traversals perform very competitively, beating both the hand-written and visitor implementations in the `Compile` test. As before the type-unifying case (`MaxEnv`) has a bit more overhead, but the type-preserving case (`Simplify`) is very close. The times for reflective traversal are also provided for a base comparison.

Type	`MaxEnv`	`Simplify`	`Compile`
INLINE	26 ms	25 ms	1130 ms
HAND	9 ms	21 ms	1160 ms
VISITOR	34 ms	80 ms	1187 ms
REFLECTIVE	791 ms	893 ms	2723 ms

Figure 33: Performance of compile related functions

9. RELATED WORK

The traversal-based approach of DemeterF is similar to other generic and generative programming projects. In OO programming much work has been centered around the visitor

pattern [16] and related tools, while work in functional languages focus more on new forms of polymorphism and polytypic programming. Both components of DemeterF have ties to AOP, supporting static AOP through open classes and traversal inlining, and dynamic AOP with reflective traversals and woven residual dispatch. Data-generic support in DemeterF allows methods to be injected based on the *shape* of a data structure, providing a less general, but still powerful form of advice.

Traversals in DemeterF (similar to DemeterJ [39]) also fall under a more traditional AOP model. In [29] the authors discuss the relations of several aspect oriented systems, including DemeterJ. Following their description, we can define the join point model of DemeterF as the entry (for `update` methods) and exit (for `combine` methods) of objects during the depth-first traversal of a data structure. DemeterF function-objects can be seen as parametrizable advice, while the control and `combine` method signatures are analogous to pointcuts: selecting a set of dynamic join points corresponding to the types of recursive results, which can be enhanced by the type of the traversal context. In contrast to DemeterJ, we execute only the most specific pointcut/advice at a given join point, similar to Fred [34] and Socrates [35].

Our goal is to provide a safer, functional alternative with some of the power of AOP, while maintaining its dynamic flexibility. Due to the functional nature of our traversals, execution of advice affects later join point selection, but function-classes can be checked to be sure that advice is complete, meaning method selection will never fail. Since reflection incurs steep runtime penalties, we can statically weave most of our function-object advice to regain performance.

9.1 Demeter Tools and Generators

Adaptive OO Programming [26] combines datatype descriptions with a domain specific language that selects part of an object instance over which a visitor is executed. The two major implementations of adaptive programming, DJ [36] and DemeterJ [39], are similar to DemeterF's reflective and static traversals, respectively. DemeterJ uses a similar class dictionary syntax to generate Java classes, a parser, and various default visitors. Ideas from both DemeterJ and DJ have flown into the design of DemeterF, with a purely functional flavor. DemeterF improves on those tools with safe traversals, extensive support for generics, and customizable data-generic function generation.

Other generational tools like JAXB [4] and XMLBeans [6] are used to generate Java classes and XML parsers from schemas. Though the design of the created classes enforces good programming practices, the tools seem to have little support for other generic or generative features, and do not support a notion of parametrized structures. Parser generators like JavaCC [5] and ANTLR [3] have built in support for generating code for tree based traversals. JavaCC includes a tool JJTree that provides support for writing automatic visitor methods, and ANTLR provides similar functionality with *tree parsers*.

9.2 Visitors and Multi-methods

The visitor pattern is most commonly used in OO languages to implement functions over datatypes without requiring instance checks or casts. Typical implementations employ double dispatch, though reflection has also been used [37, 36]. The visitor pattern has a sound type-theoretic background [8, 41], and has been at the center of discussions of extensible functions [22] and the expression problem [40, 32]. There is an opinion that multi-methods [15, 13] eliminate the need for the visitor pattern, but visitors can still be used to abstract traversal code, similar to the `Walkabout` visitor [37]. In DemeterF we use multiple dispatch to support both abstraction and specialization within function-classes. Type-checking of DemeterF function-classes over traversals is similar to that employed in multi-method systems [14], though we must infer more complex recursive cases.

9.3 Generic Programming

Gibbons [17] gives a comprehensive review of datatype generic programming. It is well known that higher-order functions such as `fold` [30] can be generalized [38, 18] to other datatype shapes, and the result is something similar to DemeterF traversals. Data-generic features in DemeterF are modeled after functional languages that support forms of shape polymorphism. PolyP [19] has similarities to Generic Haskell [28], both of which support the definition of functions over an abstract view of a datatype. Light-weight generic programming approaches such as Scrap Your Boilerplate [23] have been developed, making use of modular extensions provided by Haskell's typeclasses. A later paper in the series [24] presents a solution to extensible generic functions. The type checking, extensibility, control, and contexts of DemeterF function-classes set it apart from other functional approaches, though our type system and tools are not integrated into the underlying language.

10. CONCLUSION

We have introduced a new form of traversal-based generic programming that uses function-classes to define both generic and specific functions over data structures. Traversal functions employ a type-based multiple-dispatch allowing functions to be modular, flexible, and extensible. DemeterF generates classes and functions from structural descriptions of data types, including function-classes for generic programming. Function-classes can be checked and woven with data structure traversals to achieve performance that is competitive with hand written instance methods. Our traversals support high-level abstractions resulting in modular and extensible functions, while retaining flexibility and performance.

In the future we plan to use our weaving approach to implement implicitly parallel traversals that scale to multi-core architectures, further increasing performance.

11. REFERENCES

[1] The AspectJ Project. Website. `http://www.eclipse.org/aspectj/`.

[2] *Independently Extensible Solutions to the Expression Problem*. ACM, 2005.

[3] ANother Tool for Language Recognition. Website, 2008. `http://www.antlr.org/`.

[4] JAXB reference implementation. Website, 2008. https://jaxb.dev.java.net/.

[5] The Java Compiler Compiler™. Website, 2008. https://javacc.dev.java.net/.

[6] XML Beans overview. Website, 2008. http://xmlbeans.apache.org/overview.html.

[7] Ruby Programming Language. Website, 2009. http://www.ruby-lang.org/en/.

[8] P. Buchlovsky and H. Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electr. Notes Theor. Comput. Sci.*, 155:309–329, 2006.

[9] B. Chadwick. Algorithms in DemeterF. http://www.ccs.neu.edu/home/chadwick/files/algo.pdf, May 2009.

[10] B. Chadwick. AOSD-10 example code. Website, 2009. http://www.ccs.neu.edu/home/chadwick/aosd10/.

[11] B. Chadwick and K. Lieberherr. A Model of Functional Traversal-Based Generic Programming. Submitted to *Higher-Order and Symbolic Computation*, Festscrift for Mitch Wand http://www.ccs.neu.edu/home/chadwick/files/mitchfest.pdf.

[12] B. Chadwick and K. Lieberherr. A Type System for Functional Traversal-Based Aspects. In *AOSD '09, FOAL Workshop*. ACM, 2009.

[13] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92*, pages 33–56. Springer-Verlag, 1992.

[14] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *TOPLAS '95*, 17(6):805–843, November 1995.

[15] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00*, pages 130–145, 2000.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[18] R. Hinze. Efficient generalized folds. Technical Report IAI-TR-99-8, Institut für Informatik III, Universität Bonn, jun 1999.

[19] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97*, pages 470–482. ACM Press, 1997.

[20] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[22] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113, London, UK, 1998. Springer Verlag.

[23] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. volume 38, pages 26–37. ACM Press, March 2003. TLDI '03.

[24] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05*, pages 204–215. ACM Press, Sept. 2005.

[25] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *PADL '02*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[26] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[27] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[28] A. Loeh, J. J. (editors); Dave Clarke, R. Hinze, A. Rodriguez, and J. de Wit. Generic haskell user's guide. Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

[29] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP '03*, pages 2–28, 2003.

[30] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA '91*, volume 523, pages 124–144. Springer Verlag, Berlin, 1991.

[31] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[32] B. C. Oliveira. Modular visitor components. In *ECOOP '09*, pages 269–293. Springer-Verlag, 2009.

[33] B. C. D. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08*, pages 439–456, 2008.

[34] D. Orleans. Incremental programming with extensible decisions. In *AOSD '02*, pages 56–64, New York, NY, USA, 2002. ACM.

[35] D. Orleans. *Programming Language Support for Separation of Concerns*. PhD thesis, Northeastern University, June 2005.

[36] D. Orleans and K. J. Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001*, Kyoto, Japan, September 2001. Springer Verlag.

[37] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC '98*, Washington, DC, USA, 1998.

[38] T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA '93*, pages 233–242. ACM Press, New York, 1993.

[39] The Demeter Group. The DemeterJ website. http://www.ccs.neu.edu/research/demeter, 2007.

[40] M. Torgersen. The expression problem revisited. In *ECOOP '04*, pages 123–143, 2004.

[41] T. VanDrunen and J. Palsberg. Visitor-oriented programming. *FOOL '04*, January 2004.