A Generative Approach to Traversal-based Generic Programming

Bryan Chadwick Karl Lieberherr

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{chadwick, lieber}@ccs.neu.edu

Abstract

The development of complex software requires the implementation of functions over a variety of recursively defined data structures. Much of the corresponding code is not necessarily difficult, but more tedious and/or repetitive and sometimes easy to get wrong. Data structure traversals fall into this category, particularly in object-oriented languages where traversal code is spread throughout many cooperating modules. In this paper we present a new form of generic programming using traversals that lends itself to a flexible, safe, and efficient generative implementation. We describe the approach, its relation to generic and generative programming, and our implementation and resulting performance.

1. Introduction

The development of complex software requires the implementation of functions over a variety of recursively defined data structures. The design (or modeling) of these structures can itself be difficult, but complex data can lead to even more complex functions. How much of this complexity can be handled for the programmer? Is it inherent in the problem, or is it more dependent on our choice of data organization or implementation language?

The programmer's main tool for managing complexity is abstraction: functions abstract over values, generics (also called bounded, or parametric polymorphism) abstracts over types, and various forms of polytypic programming support abstraction over the shape of data. Each of these abstractions can be considered a different kind of (datatype) *generic programming* (15), with many different incarnations in current programming languages. In Object-Oriented (OO) Languages such as Java and C#, the first two forms are quite easy to realize through methods, interfaces, and generic type parameters, but abstracting over the shape of datatypes is less conventional, and arguably not possible using typical standard constructs.

In this paper we present a new approach and set of tools, collectively called DemeterF, for generative, traversal-based generic programming. In particular:

- We introduce a new form of traversal-based generic programming that uses function objects to fold over structures (Section 4). Functions are flexible, extensible, and written independent of the traversal implementation using a variation of multiple dispatch. This provides a special form of shape polymorphism with support for both general and specialized generic functions (Section 5).
- Our approach is supported by a class generator that consumes a concise description of data types and produces Java classes along with specific instances of generic functions like `parse()`, `print()`, and `equals()` (Section 6). The generative framework is extensible, so programmers can add their own generic functions parametrized by datatype definitions.
- Function objects (specific and generic) can be type checked against a given data structure traversal to ensure safety (Section 7). A number of different traversal organizations can be generated for specific data structures including recursive, context passing, and even implicitly parallel versions. Type-correct function objects can then be inlined in generated traversals to reach the performance of specialized, hand-written code.

Our contribution is a combination of approach and implementation. Traversal-based function classes support a function-centric design, which eliminates the problems generally associated with operation extensions in OO languages. But, functions are just classes and are likewise extensible. This dual extension of functions and data 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, sometimes achieving numbers better than hand-coded methods. Overall we retain flexibility, extensibility, and efficiency.

2. Background

We begin by thoroughly describing the problem with an interesting example. Consider the definition of an OO picture library, similar to that discussed in (20). Figure 1 contains Java classes that form the base of the example: the superclass `Pict` has three subclasses representing `Circle`s, `Square`s, and `Offset` pictures respectively. Of course, all the code from the paper is available on the web (8).

The `Pict` classes are somewhat limited now, and we can fix that soon, but first let's write a simple `toString()` function, usually referred to as *pretty printing*. As you might have guessed, this can be difficult in Java, especially once we separate our classes into different files, since we must insert a new method into each class. Figure 2 shows the inserted code with comments describing

```
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

where each method belongs; the recursive call in `Offset` is made explicit, but it should otherwise be familiar. If our classes contained other non-primitive classes we must be sure that `toString()` is implemented in them as well, to avoid nonsensical outputs.

```
// 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

This simple operation extension illustrates a few issues that place unneeded burden on programmers. First, OO class definitions are generally[1] *closed*; in Java this is especially true for `final` classes and value types, since these cannot be subclassed. This is not necessarily a bad thing because it conserves modularity, but it certainly makes programs difficult to evolve and maintain. 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 different and interesting parts of the computation. Finally, the `toString` function does not depend on anything intrinsic to the problem, only on the the names and structures within the class hierarchy. `ToString` is, of course, a special case, but in general there are many functions that can be written directly from data type descriptions, without the need for programmer specialization. To our knowledge, such forms of generic and meta programming have not previously been thoroughly explored in OO languages such as Java.

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 relatively easy. To demonstrate we can add a new picture subclass that allows us to represent compositions. Figure 3 contains a new class, `Overlay`, that represents a simple overlaying of two pictures.

This brings us to a crossroads: if we use the 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 due to its lack

[1] We say *generally* because open classes are available in some dynamic and hybrid OO languages including MultiJava (12) and Ruby (3).

```
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

of safety (casting (20)) and/or performance issues (reflection (28)). In either case we run into problems similar to those above, but is it possible to have the best of both worlds, 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 collection of picture classes (Figure 1), the function class that implements `toString` is shown in Figure 4. To understand the computation involved, we simply need to *think* like a traversal.

```
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 a traversal

In this case, the generic `Traversal` (constructed in the `toString` method) walks the structure of a given picture. When the walk reaches a `Circle` or a `Square`, the fields are expanded and passed to the matching `combine` method (`(Circle,int)` or `(Square,int)` respectively). The same is done when traversing an `Offset`, but the recursive field (`inner`) is traversed *before* a `combine` method is selected and called. In this case the `String` resulting from the traversal of `inner` is computed and passed to the (one and only) matching `Offset` method.

This is similar to generalized folds (29) with an object oriented flavor. The base class for function classes in DemeterF is `ID`, which contains `combine` methods for Java's primitive types. The benefit of function classes is that extending user defined function classes is no different than extending data types: when our picture classes are extended with `Overlay`, we simply subclass `ToString` to handle the new case. The resulting extension is shown in Figure 5.

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

Figure 5. `ToString` extended for `Overlay`

Perhaps a better way of creating this particular function is to describe the structure of our picture classes, and use it generate the function automatically. DemeterF accepts a textual representation of the class structures called a *class dictionary* (CD), which looks like a mix of BNF and algebraic data types in Haskell (26). The CD for our Pict classes appears in Figure 6.

```
// pict.cd: class dictionary for pictures
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. CD for Pict classes

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[2]. The CD can also include concrete syntax strings for printing and parsing, but with our CD in hand, we can generate the necessary toString functionality with a call to DemeterF:

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

The code generated for ToString is almost exactly the same as what we wrote by hand, but it can be generated for *any* data structure described by a CD. We also get other functions for free, like parse() and hashCode, but the most important generic function is traversal itself. Because we've written our ToString function without explicit traversal, we can use the picture CD together with the ToString class definition to produce a specialized *inlined* traversal.

Using our type checker we calculate the return value of each traversal and produce code that traverses the each of the classes, calling the appropriate combine methods. In many instances our inlined code can actually perform *better* than hand-written instance methods. Figure 7 gives average performance numbers of three different implementations of ToString run 10 times on a very large Pict instance with over 5000 nodes. The first is the DemeterF inlined version; the second is hand-coded methods directly from Figure 2; and the final one is a hand-written visitor using double-dispatch, for comparison.

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

Figure 7. Performance of various ToString implementations

In the rest of this paper we provide the details of our traversal-based approach, and how generic and generative programming fit in to provide flexibility, extensibility, and performance.

4. Traversals and Computation

The traversal of data structures can be thought of simply as a higher-order function; a function that takes a function as an argument. In functional languages, such as Scheme (19) and Haskell (18), lists are central data structures. These languages provide several useful abstractions for processing lists, like foldr, map, *etc*.. Traversal is one such function that generalizes to other kinds of data structures, and can be used to implement both specific functions (like print) and generic functions, like foldr. In this section we provide a background and overview of our traversal approach as a basis for writing other functions.

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

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; the hand-coded methods are 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

We can think of this function as implementing straight-forward structural recursion: at each point where the structure is recursive, the function is also recursive. Similar to folds, typical functional visitor approaches (6; 13) implement this sort of computation using methods that essentially replace the constructors of concrete variants. If we added the correct scaffolding for picture visitors, the function would 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. A visitor implementation of circles

In order to abstract out the traversal, in DemeterF place the recursive (sub-)traversal results from the object's fields after the original object itself. This allows the combine method selection to be uniform, with a variant of multiple dispatch. The DemeterF implementation of circles is shown in Figure 10.

```
class CircsDemF 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

The hand-coded, visitor, and DemeterF functions all look similar, the major difference being that in the DemeterF case the recursion is implicitly done for us: the arguments to the combine methods have already been traversed *before* the combine method is called. Moreover, the interesting computation involved is precisely encapsulated in our function class, with boilerplate code left to the traversal implementation. Creating creating new, or extending existing, functions over the data structures is rather simple. For example, consider implementing a new function, squares that counts the number of Squares in a given picture; the DemeterF version is shown in Figure 11. Since our computation is succinctly written, the abstract traversal provides a platform for reuse.

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

Figure 11. `squares` implementation using `CircsDemF`

4.2 Traversal

The idea of abstraction is to eliminate similarities by parametrizing over only what is different. When abstracting the traversal from computation we use a depth-first traversal approach that treats all values as objects, *i.e.*, primitives are treated as objects without any fields. Assuming similar implementations for each of our types, the basic traversal strategy is illustrated with a simple 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);
}
```

This method cannot, in general, be implicitly type checked by Java, but it shows our interpretation of structural recursion: each field is traversed in turn, and the results are passed (along with the originally traversed object) to the function object's `combine` method. The type parameters (`Ret`, `P`) signify that the traversal of different types may return different results. Here both `top` and `bot` are `Pict`s, so their traversals must return a unified type.

The situation is exactly the same for primitive types and user defined classes without fields: the traversal simply delegates to the function object, since there are no other fields to traverse.

```
<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. Since Java is a single dispatch language, the function object dispatch and type checking become slightly more involved. We will get back to these when we discuss type checking and inlining in Section 7.

4.3 Traversal Flexibility

Though a traversal that implements structural recursion everywhere throughout an object is very useful, sometimes other strategies are needed. One that is particularly useful is `onestep` (21). In DemeterF we provide other types of control (not discussed here) of which the `onestep` traversal is a special case. This allows programmers to efficiently implement a traversal style closer to hand-coded recursion. Figure 12 shows a function class that returns the topmost primitive picture (`Circle` or `Square`) in a given `Pict` instance.

Rather than letting the traversal completely control our path through a picture, we can control the recursion ourselves, one step at a time, similar to a classic visitor solution. `Traversal.onestep()` returns a traversal that steps into an object and passes its fields to the function object's matching combine method. The multiple dispatch also gives use the added benefit of being able to abstract over multiple method cases. Here the circle and square methods are abstracted into a single `combine` over `Pict`.

4.4 Contexts

Traditional visitors (14) 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 clear and simple to optimize, but this limits 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, DemeterF supports the idea of a *traversal context*. The initial (root) context is given by passing an extra argument to the `traverse` method and the traversal automatically passes the context around. The function object can then use `update` methods to modify the context for children/fields of an object being traversed.

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

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

Figure 12. `TopMost` using a `onestep` `Traversal`

For example, if we attempt to generate a visual representation of a `Pict` object, we notice that information gets lost during the generic traversal; an `Offset` instance contains all the positioning information for its children. Using traversal contexts we can easily encapsulate this information into a drawing context. A simple `Ctx` class representation is shown in Figure 13.

```
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 13. Drawing context offset

To show the power of contexts, we'll implement a function to convert a `Pict` into a *Scalable Vector Graphics* (SVG) string. SVG is a popular XML format for representing visual elements, which is very portable and simple to generate. Figure 14 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 the `traverse` method is called we pass an extra argument that becomes our root context pointing to the center of the canvas, $(w/2,\ h/2)$. Before recursively traversing the fields of an `Offset`, the traversal will call a matching `update` method to produce a new context. In this case, the update method's second parameter type, `Fields.any`, corresponds to a DemeterF class representing all fields; more complex uses of `update` methods will be discussed in Section 6, where we generate representative field classes for each class.

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 child traversal unchanged.

```
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.dx, off.dy); }

  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 14. `Pict` to SVG format using a traversal context

5. Generic Programming

Now that we have a handle on the programming style of DemeterF, we can delve into the details of the more generic forms of traversal-based programming. We call the programming style of DemeterF *generic* because it generalizes the shape of the data types being traversed: functions do not rely on the specific types of fields, but on the return types of the traversal of those fields[3]. For instance, in the `ToString` function class (Figure 4), the traversal of an instance of a concrete `Pict` class 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*, sometimes referred to as *queries* and *transformations* (21). The first category contains functions similar to `ToString` and `Circs`, where each sub-traversal returns the same type, with recursive results combined using the same function, *e.g.*, `String` or `int` combined using $+$. The second category contains certain kinds of transformations and functional updates, where we change interesting parts of the data structure and reconstruct the rest.

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 results into a single value. The skeleton of the `TU` class is shown in Figure 15.

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

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

Figure 15. Abstract class for type-unifying computations

How can we use this class? Figure 16 contains a new definition of our `CircsDemF` function class (from Figure 10) that counts the `Circles` 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. The final method describes the interesting part of the structure, `Circle`, where we return a 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 16. Generic `circles` count using `TU`

In our experience, `TU` is most useful for computations that collect information over a complex data structure, which usually involves some form of library structures to collect instances. Figure 17 shows a typical use of `TU` with DemeterF `Lists` to collect results over a generic structure. Note that we use DemeterF functional (immutable) `Lists`, so `append` returns a new `List` instance.

```
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 17. Typical TU collection into `Lists`

5.2 Type-Preserving Functions

While `TU` functions collect various results of a single type together, type-preserving functions perform recursive *transformations* over a data structure. The basic idea is easily demonstrated by writing a *copy* function class for `Picts`, shown in Figure 18.

```
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 18. Copy function class for `Picts`

We write a combine method for each `Pict` subclass, which takes parameters with the same types as its fields and constructs a new instance with the recursive results. While `Copy` is specific to `Picts`, the completely generic version of this function is implemented in the DemeterF class `Bc` (the *building constructor*). When implementing transformations we can extend the generic function with specific combine methods; Figure 19 shows a function class that scales a picture by a given factor. This function class is completely generic and applicable to *any* data structure, though sometimes this kind of function can be too general. It is usually a good idea to somehow restrict its use, in this case we only apply it to `Picts` to preserve its "scale" meaning.

The benefit here is that we mention as little of our structure as possible; we only need to write methods for the interesting parts to be transformed. As another example, Figure 20 shows a function class that converts all the `Circles` in `Pict` instance into `Squares` of the same size. We only refer to the classes to be transformed, namely that `Circle` contains an `int` radius, or more precisely, something for which our traversal will return an `int`.

As a final `Bc` example, Figure 21 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

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

```
class Scale extends Bc{
  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 19. `Scale` transformation for `Pict`s

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

Figure 20. Convert circles into squares

method are the recursive results of our function object over the traversal; the `t` and `b` arguments have already been `Flip`ed once our `combine` is called.

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

Figure 21. Reverse top to bottom `Pict` ordering

6. Generative Programming

Specialized versions of the completely generic DemeterF `Traversal`, `TU`, and `Bc` classes depend only on the specific structures involved. In our library these classes are implemented using reflection, which severely inhibits performance. The key to overcoming this limitation is the idea that dynamic structural reflection can be replaced by static information from a class dictionary (CD). In this section we describe the generative possibilities of CDs, focusing on the generic classes we provide in DemeterF and the specialization of traversal-based generic functions.

6.1 Data-generics in DemeterF

We start with an overview of data-generic facilities and a few typical data-generic functions: equality, parsing and printing. The DemeterF class generator has methods that read in a CD, resolving any includes, and creates a list of class descriptions. There are some functions, like equality, that deserve special mention, but most other generic functions can be generated over a traversal of a CD. Users can choose a number of functions to be generated over the class descriptions, but, while many useful functions are provided, a key feature of DemeterF is that users can implement their own function classes, to be used to generate specific code.

A typical command-line use of DemeterF to generate the related classes for `pict.cd` would look like:

```
>% java DemeterF pict.cd --dgp:Print
```

Where after `--dgp:` is a colon separated list of function classes that describe *data generic programming* functions. Implicit in this command is the generation of the Java classes, a canonical `equals` method, and parser generator input for JavaCC (4); though each can be suppressed with `--nogen`, `--noequals`, and `--noparse` respectively. The use of `Print` here introduces a `print()` method into each class that triggers a traversal using a generated function class. A CD file usually includes concrete syntax strings in class definitions, which makes its way into parsing and printing code.

6.2 Special Cases

A few structure-based methods deserve special cases within our class generator, mainly because they are not easy to write generically, or they require the traversal of more than one data structure simultaneously. For instance, the class generator introduces a canonical `equals(Object)` method into each concrete class, which implements deep (extensional) equality. The method generated for our `Overlay` class is shown in Figure 22. Although equality could be implemented using our traversal library, it remains a special case to enhance both performance and code clarity.

```
/** Is the given object Equal to this Overlay? */
public boolean equals(Object o){
    if(o == this)return true;
    if(!(o instanceof Overlay))return false;
    Overlay oo = (Overlay)o;
    return (top.equals(oo.top) && bot.equals(oo.bot));
}
```

Figure 22. Generated `equals` method for `Overlay`

The other special case of the generator is *field classes*, which are used represent fields, used with update methods. Inner class definitions are added to the generated classes, and are passed to matching update methods prior to the traversal of the corresponding field. For example, the field classes generated for `Overlay` would be:

```
static class top extends Field.any{}
static class bot extends Field.any{}
```

which allows us to use the type `Overlay.top` in update methods to change the context only for the `top` field. We will see an example use in Section 8.4.

6.3 DGP Functions

DemeterF supports a generative form of meta-programming over the structure of data types, an idea similar to PolyP (17). Each dgp function adds a method to each class, which by default is a lower-case version of its class name. The built in functions generate a method body that calls a static stub method; Figure 23 shows a snippet of the generated `Print` class including the static method to be called by specific classes. The main goal of dgp functions like `Print` is to generate function classes that compute their results over a traversal.

```
class Print extends ID{
  /** Static stub method for calling print */
  public static String PrintM(Object o){
     return new Traversal(new Print()).traverse(o);
  }
     /* ... combine methods ... */
}
```

Figure 23. Generated `Print` function class

`Print` computes a string representation based on the syntax found in the CD, but as seen in Section 3, other print-like functions are available. `ToStr` returns a nested constructor-like description of an object, and `Display` returns an indented view of an object notated with types and field names. Each print based dgp function has a similar class that injects the canonical `toString()` method instead of its default, so the function can be used for automatic string conversion. These are aptly named `ToString`, `PrintToString`, and `DisplayToString` respectively.

6.4 Static `TU` and `Bc`

DemeterF's generic function classes, TU and Bc, are also quite easily specialize for a given CD. We call the corresponding function

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.

classes `StaticTU` and `StaticBc`, and they can be generated by including them in the command-line dgp list. The result is something quite similar to the `Copy` function from Figure 18. The main benefit of generating these functions is to create type-safe (non-reflective) versions for precise inlining and improved performance. We'll see more uses of these generated functions in Section 8.

7. Types, Inlining, and Performance

Types play a central role in DemeterF traversals, both in the traversal of data types and the selection of `combine` methods. In order for traversal to be safe we must be sure the functions selected over the traversal fit together correctly. As a bonus, with the traversal return types in hand, in many cases we can eliminate the overhead of multiple dispatch by generating a specific traversal with inlined calls to combine methods. In this section we give an overview of type checking in DemeterF and discuss traversal inlining and performance.

7.1 Types

In DemeterF each function class is just a Java class and must conform to Java's typing rules, but things get interesting when we interpret its `combine` methods as a function over a specific data structure. For example, consider our `CircsDemF` function class (Figure 10); each method returns an `int`, which means that the traversal of each subclass of `Pict` must return an `int`. Using the CD (Figure 6), we can check that each `combine` method has the right number and types of arguments to accept the recursive results. A quick walk over the definitions in the CD tells us how many arguments to expect, and the function class tells us what types the traversal will return for each. Our goal is to prove that we will always have an applicable `combine` method during traversal. The type-unifying case generalizes for other functions, including `Copy` (Figure 18) and more ad hoc transformations like `Circ2Sqr` (Figure 20)[4].

The basis of our type system has been formalized (9) with a more algotithmic discussion here (7), but there's one important trick involved; when the use of a type in the CD is recursive, then there's no way to know what type the traversal will yield. In this case we assume that it could be *anything*. For instance, the field `inner` of `Offset` is a recursive use of `Pict`. When calculating the combine method that will be called for `Offset`, we calculate the traversal type for the first two parameters, but the third is unknown, so we look for any `combine` applicable to:

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

In most cases this will limit us to a single function, so a constraint can be placed on the recursive type based on the matching method. For `Offset`, in the `CircsDemF` case this constrains the traversal of a `Pict` to return an `int`, whereas for `Copy` it must return a `Pict`. In some cases there may be more than one applicable method, which simply results in multiple constraints. For example, consider the function class `Compress` in Figure 25, that recursively replaces nested `Offsets` with a single instance.

Here there are two methods that may be applied after traversing an Offset, the one here and the one inherited from `Copy`, which differ only by their last argument. When constraining the recursive argument, we choose the common supertype of `Pict` and `Offset`, which is just `Pict`. Similarly for the traversal of abstract classes like `Pict`, the return type of a traversal is a common supertype of the return types of subclass traversals.

[4] `Circ2Sqr` is not strictly *type-preserving*

```
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 25. Reverse top to bottom `Pict` ordering

7.2 Inlining

As long as the `combine` methods mesh together and all constraints are satisfied, 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, adding code to dynamically resolve the method selection 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:

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

7.3 Performance

The main motivation for generating traversals is to improve performance, similar to partial evaluation. As a comprehensive performance test, we have implemented each of the functions described previously in the paper three different ways: DemeterF function classes, hand written instance methods, and double-dispatch visitors. Figure 24 contains the results of running each implementation of the functions on large generated `Pict` instances. Each time is an average of 10 runs, on a `Pict` 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, which provides a good comparison for typical implementation styles in Java. The final row is the DemeterF reflective traversal for a base comparison. The 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 method delegation, but inlined `Bc` based functions perform very well, without the need to write any traversal code by hand.

8. Example: Expression Compilation

As a more complicated example using DemeterF, in this section we discuss the implementation of a compiler for a simple expression language. We write function classes to simplify constant expressions, calculate the maximum local variable usage, and convert our arithmetic language that includes variable definitions and uses, if expressions, and binary operations, into a low level stack-based operations similar to those found in the Java Virtual Machine. We first examine our target data structures, then discuss the source structures and the different operations involved in the transformation from one to the other.

8.1 Structures

To build a compiler we need representations for both our source and target languages. The abstract and concrete syntax of both languages can be described with a few CDs. Figure 26 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 26. 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 (8). Figure 27 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 27. Expression structures CD

The command to generate all the class definitions is shown below.

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

DemeterF uses the dgp functions to generate our `print` methods, and static versions of our generic function classes. As for parsing, a simple term in this expression syntax would look something like:

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

and can be parsed with the Java statement below, though for the rest of the example we will parse expressions from file streams.

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

8.2 Max Environment

A typical operation needed when compiling languages with local definitions is to calculate the maximum number 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 28 shows a function class that calculates the maximum local definition nesting for an expression. Variables are bound by `Def`s, so we calculate return the maximum of `body+1` and the result from the expression. We extend `StaticTU`, which handles other cases like `Num` and `Bin`, and can be used to generate inlined traversals.

```
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 28. Maximum local environment calculation.

8.3 Simplification

As a second example, Figure 29 shows a function class that implements the bottom up simplification of constant expressions in our mini language. We extend the generated class `StaticBc`, so we can efficiently inline the function class later.

```
class Simplify extends StaticBc{
  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 29. Simplification function class

The special cases in our arithmetic language are each captured by a `combine` method, while the rest of the reconstruction is handled implicitly by `StaticBc`. Instances of `Num` 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 results from the *thn* or *els* fields, respectively.

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 30 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) throws Exception{
    Exp e = Exp.parse(new FileInputStream(file));
    return new Traversal(this)
        .traverse(e, List.<ident>create());
  }
}
```

Figure 30. Main compile class

We have a single method, `compile(Exp)`, that traverses the given expression to produces representative opcodes in a `List<Op>`.

List is the functional (immutable) list implementation provided in the DemeterF library with typical methods: create, append, and lookup. All our DemeterF library classes are also described by a CD file, so our generative traversal approach applies equally well. When compiling, we use the traversal context to pass the stack of local variables (List<ident>) to nested definitions, starting with an empty List<ident>.

```
class Arith extends ID{
    static List<Op> empty = List.create();
    static List<Op> one(Op o){ return empty.append(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 31. Compile for arithmetic Ops

Figure 31 shows the combine methods for math related operators. The static field empty and the method one(..) simplify 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 simple expression (- 4 3) would generate the following instruction sequence:

```
push 3
push 4
minus
```

The Defs class in Figure 32 implements the compilation of variable definition related expressions. We generate a Load operation for a variable reference, with the offset of the identifier from the environment, which is passed as the last argument of the combine. 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. Once all sub-expressions have been compiled, the body code is wrapped in Define/Undef and appended to the code for the binding evaluating.

```
  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 32. Compile for Variables

The final class, Cond shown in Figure 33 deals with conditional expressions. We use a local variable to create unique Labels within the generated code, as fresh(·) creates an new ident. The IfNZ opcode is used to branch to the *else* portion when the condition is not zero, otherwise the *then* will be executed and we Jmp to the *done* label.

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 34 contains the average results of 10 runs of each on a very large Exp instance. DemeterF inlined traversals perform very competitively,

```
class Cond extends Defs{
  int lnum = 0;
  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 33. Compile for Conditionals

beating both the hand-written and visitor implementations in the Compile test.

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

Figure 34. Performance results for 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 (14) and related tools, while work in functional languages focus more on new forms of polymorphism and polytypic programming.

9.1 Demeter Tools and Generators

Adaptive OO Programming (23) combines datatype descriptions with a domain specific language that selects a portion of an object instance, over which a visitor is executed. The two major implementations of adaptive programming, DJ (27) and DemeterJ (30), are similar to DemeterF's reflective and static traversals, respectively. DemeterJ uses a similar class dictionary syntax to describe datatypes and generate Java classes, a parser, and various default visitors. Ideas from both DemeterJ and DJ have flowed into the design of DemeterF, with a purely functional flavor. DemeterF improves on those tools with type-safe traversals, support for generics, and customizable data-generic function generation. Similar to the Law of Demeter slogan, "*talk only to your friends*", the programming style of DemeterF can be described as "*listen only to your friends*".

Other generational tools like JAXB (2) and XMLBeans (5) are used to generate verbose Java classes and parsers from XML Schemas. Thought the design of the created classes enforce good programming practices, the tools seem to have little support for other generic features, and no notion of parametrized classes. Parser generators like JavaCC (4) and ANTLR (1) have built in support for generating code for tree based traversals. JavaCC includes a tool JJTree that includes support for automatic visitor methods; 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 use a double dispatch technique, though reflection is also used (28; 27). The visitor pattern has a sound type-theoretic background (6; 31), and has been central in OO discussions of extensible functions (20). There is an opinion that multi-methods (12; 10) eliminate the need for the visitor pattern, but visitors can still be used to abstract traversals similar to the `Walkabout` (28) visitor. 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 (11).

9.3 Generic Programming

Gibbons (15) gives a comprehensive review of datatype generic programming. Higher-order functions such as `fold` (25) can be generalized (29; 16) to other datatype shapes, similar to DemeterF's traversals, which adapt to the shape of a data structures. The data generic features of DemeterF are modeled after functional languages that support forms of shape polymorphism. PolyP (17) has similarities to Generic Haskell (24), both of which support the definitions of functions that work over datatypes with different shapes. More light-weight approaches such as Scrap Your Boilerplate (21) have been developed, making use of modular extension provided by Haskell's typeclasses, and a later paper in the series (22) presents a solution to extensible generic functions. The type checking and extensibility of DemeterF function classes sets it apart from other functional approaches, though our checks are in addition to those of the underlying language.

10. Conclusion and Future Work

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. We use traversals that employ multiple-dispatch to allow function classes to be both flexible and extensible. Together with a generic traversal, they provide OO programmers with a special form of shape polymorphism. Our tool is able to generate classes and functions from structural descriptions of data types. Using the structures and types from the function class we can inline functions to achieve performance that is competitive with hand written instance methods. The traversal based approach of DemeterF supports a programming style that promotes functions that are flexible, extensible, and efficient.

In the future we plan to use our tool to implement parallel traversals without the performance issues that result from reflection. Now that traversal inlining and method residue have been solved, we hope to see even better performance when re-targeting traversals on multi-core architectures.

References

[1] ANother Tool for Language Recognition. Website, 2009. `http://www.antlr.org/`.

[2] JAXB reference implementation. Website, 2009. `https://jaxb.dev.java.net/`.

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

[4] The Java Compiler Compiler™. Website, 2009. `https://javacc.dev.java.net/`.

[5] XML Beans overview. Website, 2009. `http://xmlbeans.apache.org/overview.html`.

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

[7] Bryan Chadwick. Algorithms in DemeterF. `http://www.ccs.neu.edu/home/chadwick/algo.pdf`, May 2009.

[8] Bryan Chadwick. Gpce-09 submission example code. Website, 2009. `http://www.ccs.neu.edu/home/chadwick/gpce09/`.

[9] Bryan Chadwick and Karl Lieberherr. A Type System for Functional Traversal-Based Aspects. In *AOSD 2009, FOAL Workshop*, ACM International Conference Proceeding Series. ACM, 2009.

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

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

[12] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

[13] Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In Gregor Kiczales, editor, *OOPSLA '08*, October 2008.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

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

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

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

[19] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

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

[21] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. volume 38, pages 26–37. ACM Press, March 2003.

[22] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05*, pages 204–215. ACM Press, September 2005.

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

[24] Andres Loeh, Johan Jeuring (editors); Dave Clarke, Ralf Hinze, Alexey Rodriguez, and Jan de Wit. Generic haskell user's guide – version 1.42 (coral). Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

[25] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[26] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[27] Doug Orleans and Karl J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[28] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98*, Washington, DC, USA, 1998.

[29] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

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

[31] Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. *FOOL '04*, January 2004.