ABSTRACT

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 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 adaptive functions. While our reflective traversals provide maximum flexibility, DemeterF can weave the same function classes into traversals that perform as efficiently as hand-coded methods.

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 (functions). 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.

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 [39]. Typically computation is performed using mutation (side-effects), which can be 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.

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

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.

Performance Function-objects can be type-checked against data structure definitions, allowing us to weave inlined traversal and dispatch code (Section 7). The function-object remains 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. Although our traversals support high-level abstractions resulting in modular, extensible functions, we also retain 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 Circles, Squares, and Offset pictures respectively. For reference, all the code examples from this paper are available on the web [9].

```
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;
    Offset(int x, int y, Pict in)
        { dx = x; dy = y; inner = in; }
}
```

Figure 1: Picture Class Skeletons

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. This can be somewhat difficult in Java 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 where each method belongs; the recursive call in Offset is made explicit, but it should otherwise be familiar. If our classes had contained other non-primitive classes we would have to 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

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 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 [14] and Ruby [6]. 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 toString function does not depend on anything intrinsic to the problem, only on the class hierarchy itself. ToString is, of course, a special case, but in general there are many functions that can be written more generically to avoid mentioning unnecessary structural details and allow them to adapt without programmer specialization.

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. 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 [35]). 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 collection of picture classes, Figure 4 shows a function class that implements toString using our DemeterF traversal library.

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.

1The function/data centric dilemma is usually known as the expression problem.
class ToString extends ID{
    String combine(Circle c, int r)
    { return "Circle(\"r\")\"; }
    String combine(Square s, int sx)
    { 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 specialized in the toString method using a function-object, an instance of ToString. The traverse 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.

The base for function classes in DemeterF is ID, which contains identity combine methods for Java’s primitive types. As we can see, 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 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. Though at the top level we need to use our new ToStringOverlay for Picts that may contain overlays, 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

A better 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 the 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]. 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 Picts

Our abstract class Pict is described by a list of variants separated by bars (\(\mid\)), while concrete classes list their field names (in brackets) and types\(^2\). Because a generic form of ToString is included with DemeterF, our CD can be used to generate the necessary functionality with a call to DemeterF:

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

The dgp stands for data-generic programming, and the code 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 our 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). In many cases our inline 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. 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; and the final one, for comparison, is a hand-written visitor implemented using double-dispatch.

<table>
<thead>
<tr>
<th>Type</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>INLINED</td>
<td>48 ms</td>
</tr>
<tr>
<td>HAND</td>
<td>49 ms</td>
</tr>
<tr>
<td>VISITOR</td>
<td>54 ms</td>
</tr>
</tbody>
</table>

Figure 7: Performance of ToString

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.

4. DEMETERF TRAVERSALS

The traversal of data structures can be thought of simply as a walk over the 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; the hand-coded methods are shown in Figure 8.

\(^2\)In fact, a CD can describe any Java class hierarchy, though we won’t discuss all of our CD features in this paper.
The hand-coded, visitor, and DemeterF functions look quite similar, though the differences being that in the DemeterF 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 [7, 15], 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 Squares 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.

\begin{figure}[h]
\centering
\begin{verbatim}
// 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(); }
\end{verbatim}
\caption{circles method implementation}
\end{figure}

\begin{figure}[h]
\centering
\begin{verbatim}
class CircsVis extends Visitor<Integer>{
   Integer exist(Circle c){ return 1; }
   Integer exist(Square s){ return 0; }
   Integer exist(Offset o)
   { return o.inner.accept(this); }
   Integer exist(Overlay o)
   { return o.top.accept(this)+o.bot.accept(this); }
}
\end{verbatim}
\caption{circles Visitor implementation}
\end{figure}

\begin{figure}[h]
\centering
\begin{verbatim}
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; }
}
\end{verbatim}
\caption{circles DemeterF implementation}
\end{figure}

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:

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

\begin{figure}[h]
\centering
\begin{verbatim}
ID func;
<Ret,P> Ret traverse(Overlay o){
P top = traverse(o.top);
P bot = traverse(o.bot);
return func.combine(o, top, bot);
}
\end{verbatim}
\caption{squares implementation using CircsDemF}
\end{figure}

The traverse method is parametrized over the types returned by the traversal of an Overlay (Ret) and a Pict (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 Picts, so their traversals must return a common (unified) type, but multiple or mutually recursive hierarchies can be handled similarly. Traversal is similar for primitive types and user defined classes without fields, where the traversal delegates to the function object, since sub-traversals are needed.

\begin{verbatim}
<Ret> Ret traverse(int i){
   return func.combine(i);
}
\end{verbatim}

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. When dispatching, 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. In DemeterF we provide 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. We simply bypass (or skip) the bot field of all Overlay instances by giving our traversal a Control object. The resulting function is shown in Figure 12, using a dynamic version of Control.
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 botMost(Pict s){
        return Traversal(this).
            Control.bypass("Overlay.bot")).traverse(s);
    }
}

Figure 12: TopMost using bypassing

The combine methods in this case are the same as when
the traversal proceeds everywhere, but the overall function
becomes more efficient by eliminating unneeded traversal.
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, leaving the type
checks and casting to the traversal implementation. Figure
13 shows a function class that returns the bottom most
primitive picture, using a one-step traversal.

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

Rather than letting the traversal control our path through
a picture, we control the recursion ourselves, one step at a
time. Traversal.onestep() returns a traversal that steps
into an object and passes the bypassed fields to the function
object's matching combine method, resulting in functionality
similar to a more traditional visitor solution.

4.4 Contexts

Traditional visitors [16] employ void visit methods to en-
capsulate computations over structures, which forces pro-
grammers 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 parallel-
ize, but this can limit the communication of context sen-
sitive (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 meth-
ods to modify the inherited 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 posi-
tioning 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.

class ToSVG extends ID{
    String combine(Circle c, int r, Ctx ctx){
        String combine(Square s, int sz, Ctx ctx){
            return SVG.square(c.cx.x, c.cx.y, sz);
        }
    }
}

Figure 15: Pict to SVG using Contexts

When traverse is called we pass an initial context represent-
ing the center of the canvas, (w/2, h/2). Before recursively
traversing the fields of an Offset, the traversal will call
our update method to produce a new context. The signa-
ture of the update method can be read as: Before traversing
any field of an Offset, compute a new context from the par-
ent'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 pa-
rameter type, Fields.any, corresponds to a DemeterF class
representing all fields. Alternatively, we can create represen-
tative 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

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 necessarily rely on the specific types of an object’s fields, but on the return types of the traversal of those fields. 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 subtraversals return the same type, and are 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<> 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 `CircsDemF` 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. The final method describes the interesting part of our structure, at a `Circle`, 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, usually involves some form of library structures to collect instances (`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 List<TU><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 p) {
        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

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 `TP` that dynamically calls constructors of the object being traversed. We can extend the generic function with specific combine methods to implement our 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 `Picts` 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 Picts

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 `Squares` 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`. 

---

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

4Java requires that we use `Integer`, a reference type, rather than `int`, value type, though coercion is usually automatic.
class Circ2Spr 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 Fliped 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 [37], 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 from DemeterJ, 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:

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

This defines a 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.

```plaintext
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 instance.

```plaintext
IntList = <Int> List<Integer>.
```

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

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

6.1 Data-Generic Programming

Creating specialized versions of the 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 command line arguments to DemeterF. The chosen classes are used to traverse a portion of the CD (either a single definition, or the entire CD) and generate the necessary code. A typical DemeterF command line looks something like the following:

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

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

Our generic function classes TU and TP can also be specialize for a given CD. The corresponding classes, 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 something very similar to our Copy function from Figure 19, while StaticTU contains inlined calls to TU’s fold. The main reason for generating these classes is to create type-safe versions for precise inlining and improved performance, which we discuss next.

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 selected functions over the traversal fit together correctly. An added benefit of this static analysis is that, with the traversal return types in hand, we can eliminate most the overhead of multiple dispatch. We do this by generating a specialized 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 the traversal of a data structure. For example, consider our Circ2DemF 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 transfor-
mations like Copy (Figure 19) and more ad hoc functions like Circ2Sqr (Figure 21).

The basis of our type system has been formalized [11, 10] with a more algorithmic discussion here [8]. 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:

\[
\text{(Offset, int, int, \_)}
\]

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 Circ2DemF 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 24, that recursively replaces nested Offsets 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 24: Compress redundant Offsets

There are now 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 of a traversal is a common supertype of the return types of 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:

\[
\text{if(inner instanceof Offset)}
\]
\[
\text{return func.combine(o, dx, dx.(Offset)inner);} \\
\text{return func.combine(o, dx, dy, inner)}
\]

Where inner is the recursive traversal result. Our implementation uses a simple algorithm to recursively partition the set of possible methods, generating if statements to narrow the selections until only one method is applicable. Exploring the algorithmic properties of our static dispatch is an item of future work.

### 7.3 Performance

Similar to partial evaluation, the main motivation for weaving traversals is to improve performance. As a comprehensive 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 23 contains the results of running each implementation of the functions on a large generated Pict instance. Again we are using 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 table provides a good comparison of typical implementation styles in Java. The first row 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; the same function-objects are used for both inlined and reflective traversals. 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 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 which runs on the Java Virtual Machine. To keep things interesting our source language includes includes variable definitions and uses, 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 few 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.

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

Minus = "minus".
Push = "push" <\> int.
Pop = "pop".
Define = "define".
Undef = "undef".
Load = "load" <\> int.
Label = "label" <\id> ident.
Jmp = "jump" <\id> ident.
IHNZ = "ifNZ" <\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 [9]. Figure 26 shows a CD file that describes our expression data structures.

```plaintext
// exp.cd
Exp = Ifz | Def | Bin | Var | Num.
Ifz = "ifz" <\cond> Exp "then" <\then> Exp "else" <\els> Exp.
Def = <\id> ident =<\> <\exp> Exp ";" <\body> Exp.
Bin = "(" <\op> Exp <\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 --dgpr:Print:StaticTU:StaticTP
```

A similar command would be used for the assembly structures; DemeterF uses the dgpr 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, though for the implementation we will parse expressions from a file.

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

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 e, int id, int e, int b)
    { return fold(e, 1+b); }
}
```

Figure 27: Maximum local environment calculation.

We extend StaticTU to handle other cases like Num and Bin, since we will eventually generate inlined traversals.

8.3 Simplification

As a second example, Figure 28 shows a function class that implements the 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(); }
    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. 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 then or else, 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<Exp> 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 will use the traversal context to pass a stack of local variable names (List<ident>) for nested definitions. We start with the empty List<ident> as our root context. Here List is a functional (immutable) list implementation provided in the DemeterF library with typical methods like create, append, and lookup. The DemeterF library classes are also described by a CD file, with definitions similar to those in Section 6, and our generative/weaving approach applies equally well to external classes.

class Arith extends ID{
    List<Op> one(Op o){ return List.create(o); }
    List<Op> combine(Sub s){return one(new Minus());}
    List<Op> combine(Norm p, int i){ return one(new Push(i));}
    List<Op> combine(Bin b, List<Op> o, List<Op> r){
        return r.append(1).append(o); }
}

Figure 30: Compile for arithmetic Ops

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 ($4 \times 3$) 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 the identifier from 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 are 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 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 Labels 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+"+\$num++"); }
    List<Op> combine(Ifz f, List<Op> c, List<Op> t, List<Op> e){
        ident le = fresh("else");
        int id = fresh("done");
        return c.append(new IfNZ(le)).append(t)
            .append(new Jmp(id))
            .append(new Label(le)).append(e)
            .append(new Label(id)); }
}

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 ten runs of each on a very large Exp instance with the same configuration used before. DemeterF inlined traversals perform very competitively, beating both the hand-written and visitor implementations in the Compile test. As before the type-unifying case has a bit more overhead, but the type-preserving case is very close. The times for reflective traversal is again provided as a base comparison.

<table>
<thead>
<tr>
<th>Type</th>
<th>MaxEnv</th>
<th>Simplify</th>
<th>Compile</th>
</tr>
</thead>
<tbody>
<tr>
<td>INLINE</td>
<td>26 ms</td>
<td>25 ms</td>
<td>1130 ms</td>
</tr>
<tr>
<td>HAND</td>
<td>9 ms</td>
<td>21 ms</td>
<td>1160 ms</td>
</tr>
<tr>
<td>VISITOR</td>
<td>34 ms</td>
<td>80 ms</td>
<td>1187 ms</td>
</tr>
<tr>
<td>REFLECTIVE</td>
<td>791 ms</td>
<td>893 ms</td>
<td>2723 ms</td>
</tr>
</tbody>
</table>

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 the DemeterF system 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 giving us a less general, but powerful form of advice.

Traversals in DemeterF (similar to DemeterJ [37]) fall under a more traditional AOP model. In [29] the authors discuss the relations of different aspect oriented systems, of which DemeterJ is one. 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 method signatures together are analogous to pointcuts: selecting a set of dynamic join points corresponding to the types of recursive results. Pointcuts are enhanced through optional traversal contexts. In contrast to Demet-
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 a portion of an object instance, over which a visitor is executed. The two major implementations of adaptive programming, DJ [34] and DemeterJ [37], are similar to DemeterF’s reflective and static traversals, respectively. DemeterJ uses a similar class dictionary syntax and generates 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 safe traversals, extensive support for generics, and customizable data-generic function generation.

Other generational tools like JAXB [3] and XMLBeans [5] are used to generate Java classes and XML parsers from schemas. Thought the design of the created classes enforces good programming practices, the tools seem to have little support for other generics or generative features, and do not support a notion of parametrized structures. Parser generators like JavaCC [4] and ANTLR [2] have built in support for generating code for tree based traversals. JavaCC includes a tool JJTree that includes support for writing 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 has also been used [35, 34]. The visitor pattern has a sound type-theoretic background [7, 38], and has been at the center of discussions of extensible functions [22]. There is an opinion that multi-methods [14, 12] eliminate the need for the visitor pattern, but visitors can still be used to abstract traversal code similar to the Walkabout [35] 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 [13], though we must infer more complex recursive cases.

9.3 Generic Programming

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 has also been used [35, 34]. The visitor pattern has a sound type-theoretic background [7, 38], and has been at the center of discussions of extensible functions [22]. There is an opinion that multi-methods [14, 12] eliminate the need for the visitor pattern, but visitors can still be used to abstract traversal code similar to the Walkabout [35] 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 [13], though we must infer more complex recursive cases.

11. REFERENCES


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


