

Functional Visitors Revisited

Bryan Chadwick
chadwick@ccs.neu.edu

Therapon Skotiniotis
skotthe@ccs.neu.edu

Karl Lieberherr
lieber@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Avenue 202 WVH
Boston, Massachusetts 02115 USA

ABSTRACT

The visitor pattern has been motivated by the need to perform and add operations to elements of an object hierarchy without changing their structure. The pattern addresses issues of navigation through the object hierarchy and behavior while traversing the object structure. However, traversal code in the standard pattern tends to be scattered. Previous solutions separated traversal code from the structure of data types but still face a lack of modularity in visitor computation. We present a modification to the *Visitor Pattern* which is applicable to different traversal specifications. Our modification introduces functional style traversals and visitor methods which connect the scopes of *before* and *after* methods by way of the implicit stack of the recursive traversal. The functional nature of our visitors removes dependencies due to side effects and allows for more modular, compositional solutions, leading to more reusable designs. In addition we introduce three visitor combinators which can be used to compose modular solutions and discuss the refactorization of a working example.

1. INTRODUCTION

The Visitor Pattern [8] has long been used to localize traversal related concerns in object oriented programs. One problem with the standard pattern is its reliance on tangled traversal code spread throughout many different classes. In an attempt to alleviate this tangling of the traversal, different solutions external to the underlying class hierarchy have been suggested. Lieberherr [14] presented a domain specific language for traversal strategies—based on class dictionaries—which allows traversal concerns to be separated almost completely from program structure. Ovlinger and Wand [18] devised an explicit traversal language which imposes structure and ordering on object traversals, permitting a more functional design of traversal methods than the original pattern. Others [23, 13] have considered visitors to be the main source of traversal control and supplemented this with simple visitors and general traversal combinators. In functional languages, even though there are no explicit visitors per se, polytypic or data-generic functions [4, 10] are used to abstract over a family of types and their structure. This allows

navigation through a type's structure during computation.

All of these implementations share a common thread; separation of the traversal concern from some details of the underlying class hierarchy and visitor computation. Each one solves a specific part of the separation problem by putting a new twist on the classic pattern. In the hopes of unifying different reincarnations of the visitor pattern, we present a new formulation of the *Functional Visitor* [25]. This new formulation solves many of the issues faced in each of the previous solutions by thinking differently about the recursive nature of traversals. The changes made allow visitors to be written in a naturally functional style without worrying about side-effects. The ability to reduce side-effects in visitors makes for clear code, more opportunities for reuse, and simplified visitor and traversal composition.

2. FUNCTIONAL TRAVERSALS

In this section we introduce our modification to the visitor pattern traversal in the setting of simple lists. In many functional languages such as Scheme [11] and ML [16], there are higher order functions—*map*, *fold*, *filter*—that perform a general list traversal and encapsulate behavior in user provided functions or predicates. In this example we show hand-written traversal methods in the style of the visitor pattern, but we would like to point out that this functional style traversal is also applicable to other forms of traversal specifications, including explicit [18] and strategy-based [14] traversals.

The basic Java classes for the example are defined in Figure 1 with both the canonical visitor and the new *Functional Visitor* traversals.¹ This may be slightly different from the general notion of the pattern, as here we use *before* and *after* methods to expand visitor expressibility.² Each of the concrete classes defines an *accept* method which internally controls the traversal order of its members. Since, in our opinion, the functional traversal is orthogonal to visitors and their combination, we assume for our examples that there exists a traversal implementation internal or external [17, 19] to the hierarchy which is independent of the visitor execution and follows the calling pattern of this first example.

In both *accept* methods in the *Cons* class, the call of the visitor's *before* and *after* methods wrap the sub-traversals of *first* and *rest*. The difference in the case of *FVisitor* is not only the existence of a return value, but also the visitor on which we call the *after* method. For our new traversal we give access to the implicit stack of recursion by calling *after* on the visitor that was returned from the call to *before*. As an argument we pass the *FVisitor* which has been returned from the sub-traversals. This means that the *after* method is called on the same instance

¹Sample Implementation and Examples can be found at [5]

²Similar to the *hierarchical visitor* (Section 7)

<pre> class Visitor{ void before(Object o){} void after(Object o){} } class Data{ void accept(Visitor vis){ vis.before(this); vis.after(this); } } abstract class List{ abstract void accept(Visitor vis); } class Cons extends List{ Data first; List rest; void accept(Visitor vis){ vis.before(this); first.accept(vis); rest.accept(vis); vis.after(this); } } class Nil extends List{ void accept(Visitor vis){ vis.before(this); vis.after(this); } } </pre>	<pre> class FVisitor{ FVisitor before(Object o, FVisitor v){return v;} FVisitor after(Object o, FVisitor v){return v;} } class Data{ FVisitor accept(FVisitor vis){ FVisitor bVis = vis.before(this, vis); return bVis.after(this, bVis); } } abstract class List{ abstract FVisitor accept(FVisitor vis); } class Cons extends List{ Data first; List rest; FVisitor accept(FVisitor vis){ FVisitor bVis = vis.before(this, vis); FVisitor aVis1 = first.accept(bVis); FVisitor aVis2 = rest.accept(aVis1); return bVis.after(this, aVis2); } } class Nil extends List{ FVisitor accept(FVisitor vis){ FVisitor bVis = vis.before(this, vis); return bVis.after(this, bVis); } } </pre>
--	--

Figure 1: Left: the standard visitor pattern . Right: the functional visitor pattern

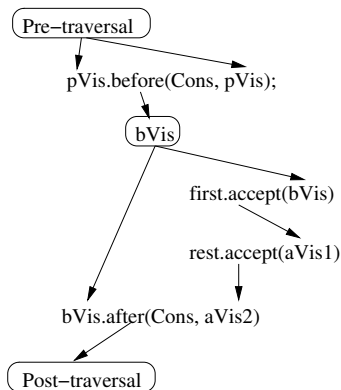


Figure 2: Cons traversal flow diagram

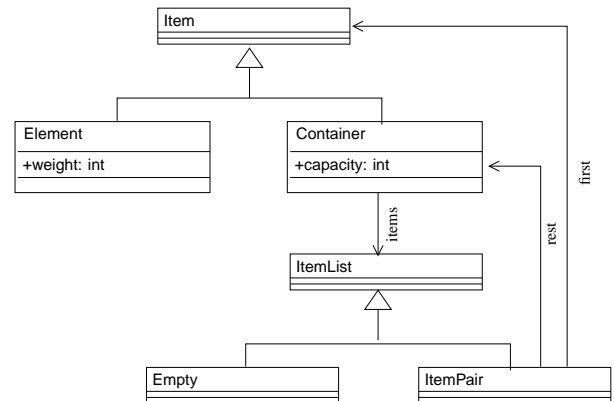


Figure 3: Container example UML class diagram

of the visitor which was passed to the sub-traversal. Assuming no side-effects we can then compare the visitor state *before* and *after* the sub-traversal of *first* and *rest*. The *accept* methods of *Data* and *Nil* are straight forward since these are the leaves of our list structure.

Figure 2 contains a visual representation of the *accept* method of the *Cons* class showing how the results of each method call flow into the next. *Pre-traversal* represents the visitor which is passed into *accept* and *Post-traversal* is the visitor which is returned. Listing 2 gives a specific example of how this new traversal can be used effectively and will be explained in the next section.

3. CONTAINER CHECK EXAMPLE

To demonstrate one of the uses of the new functional visitor pattern we present a small example— *Container Checking* [25]. In this

example we have a number of *Containers* that we would like to check for capacity violations. Figure 3 is a UML class diagram for this scenario. We continue to use this example in the following sections, reusing these same classes and assumed traversal code in the style of the list example.

Here a container has a *capacity* and a list of *Items*—essentially a multi-tree where the leaves are *Elements* and the nodes are *Containers*. Our problem is: check that containers have not been over filled, i.e. that the combined weight of a *Container*'s *ItemList* is not greater than its *capacity*. Specifically, we would like to count the number of overfilled containers in a single traversal while computing the results of sub-containers in the same pass. We could do more for each overfull container, but simply counting them suffices. Assuming we have traversal code in the style of the earlier example, Listings 1 and 2 show two possible so-

lutions: one in the case of the imperative traversal, and one in the case of the functional traversal.

These two visitors compute the same result: the visitor returned from the traversal contains the total weight of all Elements and the number of overfull (or *violated*) containers. The difference being that the *imperative* visitor operates via side-effect and must explicitly control a stack, whereas the *functional* visitor makes use of the implicit stack of the recursive traversal. In addition to not having to maintain a stack we have also gained the ability to remove all side effects from our visitor methods. This localizes state changes and can lead to more understandable control flow which then helps in designing modular visitor solutions. Given modular designs, we have the opportunity to compose functional visitors into larger units when building a specialized solution.

4. VISITOR COMBINATORS

In building visitor solutions it is common for computations to require full or overlapping traversals of an object graph. Sometimes the functionality of a given visitor can be split into modular units and composed to produce an equivalent visitor that is easier to reuse, understand, and modify. In this section we discuss some useful composition operations in the context of functional visitors and how they can be used to build software solutions that are more reusable. These combinations are in contrast to those which are presented by Visser [23] where the traversal control is implemented through the order of visitation. We believe that this tends to tangle traversal code which makes the final composition more difficult to understand. Because of this we chose to assume standard—*independent*—traversals, which are either hand-coded or based on an external domain specific language [14, 18].

Listing 3: Updated Visitor

```
class FVisitor{
  FVisitor before(Object o, FVisitor v){return v;}
  FVisitor after(Object o, FVisitor v){return v;}
  FVisitor visitDefault(Object o, FVisitor v)
    {return v;}
  FVisitor exportVisitor(){return this;}
  boolean continueVisit(){return false;}
}
```

For this discussion we introduce three forms of compositions which we call *Independent*, *Threaded*, and *Conditional*. For the development of these combinations we also introduce a simple addition to the original functional visitor interface; this makes the compositions easier to discuss because we base them on one interface instead of many, which also helps with type-checking in Java 5.

Listing 3 shows the new interface for FVisitor which includes the methods: `visitDefault`, `exportVisitor`, and `continueVisit`. The basic use of these methods is to supplement situations when we have a difference in a visitor's argument type and return type in `before` and `after` methods and when combinators require specific behavior. We use a simple dynamic dispatch for the visitor method calls which, when no matching method is found, calls the visitor's `visitDefault` method. These new methods do not effect the earlier FVisitor examples, so each specific use is explained when behavior needs to be overridden or is used by a combinator.

4.1 Independent Combination

Often problems require multiple calculations over the same traversal or set of objects. For instance, many programming language related problems require calculation of attributes on a graph-like representation of program text—generally called an Abstract Syntax Tree. In most cases we wish to compute multiple independent results from a given object collection. This would normally require multiple traversals; one pass for each visitor. Assuming we have independent visitors that require no inter-communication, we can simply compose them as a collection in some order. The functional visitor that implements this idea is shown in Listing 4.

Listing 4: Independent Combination

```
class IndependComp extends FVisitor{
  FVisitor vis1, vis2;

  IndependComp(FVisitor v1, FVisitor v2)
    { vis1 = v1; vis2 = v2; }
  IndependComp before(Object o, IndependComp v){
    return
      new IndependComp(vis1.before(o, vis1),
                       vis2.before(o, vis2));
  }
  IndependComp after(Object o, IndependComp v){
    return
      new IndependComp(vis1.after(o, v.vis1),
                       vis2.after(o, v.vis2));
  }
}
```

Here the composition is just a packaging of the results of each visitor. More than two visitors can be combined by nesting `IndependComps` forming a binary tree. The result of a traversal can then be decomposed into its individual pieces to be used in other computations. This idea is simple so we use it as an introduction of two specific visitors, `ElementSum` and `Increment`.

Listing 5: Independent Visitors

```
class ElementSum extends FVisitor{
  final int sum;
  ElementSum(int newsum){ sum = newsum; }
  ElementSum after(Element e, ElementSum vis)
    { return new ElementSum(sum+e.weight); }
}
class Increment extends ElementSum{
  Increment(int newsum){ super(newsum); }
  Increment after(Item o, Increment vis)
    { return new Increment(vis.sum+1); }
}
```

To create a simple combination of these two visitors, we can construct an `IndependComp` that sums the weight of all Elements while counting how many Items there are. The `accept` method is also used to initiate the traversal and the result is cast back to `IndependComp`.

```
IndependComp comp =
  new IndependComp(new ElementSum(0),
                  new Increment(0));
comp = (IndependComp)aContainer.accept(comp);
```

4.2 Threaded Combination

Independent Composition is useful if we have information that can be analyzed during a traversal or after control returns, but this can sometimes force developers to manually combine computations into less modular visitors. One example of this is the `FuncCheck` in Listing 2 where we have a single visitor performing at

Listing 1: Standard visitor solution

```

class ImperCheck extends Visitor{
    int weight, violations;
    Stack<Integer> stack = new Stack<Integer>();

    ImperCheck(int w, int viol)
    { weight = w; violations = viol; }
    void before(Element e)
    { weight += e.weight; }
    void before(Container c)
    { stack.push(new Integer(weight)); }
    void after(Container c){
        int last = stack.pop().intValue();
        if((weight - last) > c.capacity)
            violations++;
    }
}

```

Listing 2: Functional visitor solution

```

class FuncCheck extends FVisitor{
    final int weight, violations;

    FuncCheck(int w, int viol)
    { weight = w; violations = viol; }
    FuncCheck after(Element e, FuncCheck vis){
        return new FuncCheck(vis.weight+e.weight,
                               vis.violations);
    }
    FuncCheck after(Container c, FuncCheck vis){
        return new FuncCheck(vis.weight,
                               (vis.violations+
                                ((vis.weight-weight) > c.capacity?1:0)));
    }
}

```

least three separate functions. If we were to divide the functionality up for use with `IndependComp` then we would run into a problem with the lack of communication provided by the combination. Here we introduce a useful composition that allows a single direction of communication between visitors—the `exportVisitor` of the first visitor is *threaded* into the second, hence the name `ThreadedComp`. As a meaningful example we modify the Container checking example from earlier.

Listing 6 contains the code for `ThreadedComp`; a functional visitor that connects two other functional visitors. We introduce the `CompReceiver` visitor to separate the receiver from creating compositions by overriding `visitDefault` to return the `exportVisitor` of the assumed `ThreadedComp` argument when no suitable visitor method is found. The type signature of the `visitDefault` must remain the same, taking an `FVisitor`, in order to be overridden. In order to fully explain this combination we have reorganized the `FuncCheck` into two modules, reusing `ElementSum` from earlier.

In `ThreadedComp` the `exportVisitor` method is used to request the results that are to be threaded to the second visitor of the combination. In this case we are threading the second visitor by convention. `CompReceiver` expects to receive a `ThreadedComp` as a second argument to its `before` and `after` methods and reuses `exportVisitor` in its `visitDefault` method when no other visitor method is found. Listing 7 shows `Overfull`, a concrete example of a `CompReceiver`. `Overfull` is expected to be composed with a version of `ElementSum` as is presented in the type annotations of the argument to its `before` and `after` methods. This composition is used in the next section to produce a complete modular solution to the container check problem.

4.3 Conditional Combination

In order to encapsulate visitor computation we have chosen to discuss combinations independent of the method of traversal; assuming only that we have a means to guide the visitor execution based on some idea of a class hierarchy. This is useful for abstracting visitor behaviors that are as general as possible, but in some instances we need dynamic control over the traversal. We believe that giving the visitor total control over the direction of traversal [23] can increase code complexity and make computation less understandable, in addition to reducing reuse and modularity. To allow for some runtime control over which objects will be visited we introduce the *Conditional Combination*.

If we examine the definition of `Overfull` again, it is clear that it has been designed to answer a question— Is this Container

overfull? It does this by comparing the `ElementSum` *before* and *after* the traversal of a container. The `visitDefault` is overridden so that this question is answered correctly immediately after we have examined a `Container`. This answer, `overfull`, will then be polled by the `ConditionalComp` and the second visitor is executed if it is *true*.

The usage of the new modular container checker is shown below.

```

ConditionalComp comp =
    new ConditionalComp(
        new ThreadedComp(new ElementSum(0),
                          new Overfull()),
        new Increment(0));

```

These details of the composition at the instance level can be cumbersome, especially when the user only wants to know how many overfull containers there are. In the next section we show how this can be done while keeping the solution as flexible as possible.

5. ADAPTABILITY

Users do not generally need to know all the details of a given solution. In the case of the container checking example, the user probably only needs to know how many, or possibly which, containers are overfull. Also, in order for modular solutions to scale well for developers, it is useful to be able to hide the details of composition and abstract them away. In this section we discuss how generic composition can be hidden behind user accessible classes while still maintaining an adaptable solution.

Listing 9 shows the abstract class `Wrapper`, which encapsulates another visitor. The visitor, `forward`, is used as a delegate for all method calls. We also employ the factory method `makeWrapper` so the concrete wrapper class can be dynamically constructed [12]. `OverfullWrap` can now be used without having to know that it is a *Composition*. One such usage which produces the same results as the original `ThreadedComp` solution is shown below.

```

ConditionalComp comp =
    new ConditionalComp(new OverfullWrap(),
                       new Increment(0));

```

In order to show how adaptable this composition is, we modify our problem slightly. Imagine that after our solution has been built, we want to account for the shipping weight of each element. Elements are to be surrounded by padding whose weight is in proportion to the distance the Container is to be shipped, i.e. is not related to an *Element's* weight. In the original solution, `FuncCheck`, this update would require copying existing code, unless the issue was fore-

Listing 6: Threaded Combination

```
class CompReceiver extends FVisitor{
    FVisitor visitDefault(Object o, FVisitor v)
    { return v.exportVisitor(); }
}
class ThreadedComp <TVis1 extends FVisitor,
    TVis2 extends CompReceiver> extends FVisitor{
    TVis1 vis1;
    TVis2 vis2;

    ThreadedComp(TVis1 v1, TVis2 v2){ vis1 = v1; vis2 = v2; }
    ThreadedComp before(Object o, ThreadedComp<TVis1,TVis2> comp){
        TVis1 tvis = (TVis1)vis1.before(o, comp.vis1);
        ThreadedComp arg = new ThreadedComp(tvis.exportVisitor(), comp.vis2);
        return new ThreadedComp(tvis, (TVis2)vis2.before(o, arg));
    }
    ThreadedComp after(Object o, ThreadedComp<TVis1,TVis2> comp){
        TVis1 tvis = (TVis1)vis1.after(o, comp.vis1);
        ThreadedComp arg = new ThreadedComp(tvis.exportVisitor(), comp.vis2);
        return new ThreadedComp(tvis, (TVis2)vis2.after(o, arg));
    }
    FVisitor exportVisitor(){ return vis2; }
    boolean continueVisit(){ return exportVisitor().continueVisit(); }
}
```

Listing 7: More Modular Container Check

```
class Overfull extends CompReceiver{
    ElementSum last;
    final boolean overfull;

    Overfull(){ this(null, false); }
    Overfull(ElementSum es, boolean of){ last = es; overfull = of; }
    Overfull before(Container c, ThreadedComp<ElementSum, Overfull> comp){
        return new Overfull(comp.vis1, false);
    }
    Overfull after(Container c, ThreadedComp<ElementSum, Overfull> comp){
        return new Overfull(null, (comp.vis1.sum-last.sum) > c.capacity);
    }
    Overfull visitDefault(Object o, FVisitor vis){
        Overfull passed = ((ThreadedComp<?, Overfull>)vis).vis2;
        return new Overfull(passed.last, false);
    }
    boolean continueVisit(){ return overfull; }
}
```

Listing 8: Conditional Combination

```
class ConditionalComp <TVis1 extends FVisitor,
    TVis2 extends FVisitor> extends FVisitor{
    TVis1 vis1;
    TVis2 vis2;

    ConditionalComp(TVis1 v1, TVis2 v2){ vis1 = v1; vis2 = v2; }
    ConditionalComp before(Object o, ConditionalComp v){
        FVisitor tvis = vis1.before(o, v.vis1);
        FVisitor newVis2 = v.vis2;
        if(tvis.continueVisit())
            newVis2 = vis2.before(o, v.vis2);
        return new ConditionalComp((TVis1)tvis, (TVis2)newVis2);
    }
    ConditionalComp after(Object o, ConditionalComp v){
        FVisitor tvis = vis1.after(o, v.vis1);
        FVisitor newVis2 = v.vis2;
        if(tvis.continueVisit())
            newVis2 = vis2.after(o, v.vis2);
        return new ConditionalComp((TVis1)tvis, (TVis2)newVis2);
    }
    boolean continueVisit(){ return exportVisitor().continueVisit(); }
}
```

Listing 9: Wrapped Combinations

```
abstract class Wrapper<TVis extends FVisitor> extends FVisitor{
    TVis forward;
    Wrapper(TVis vis){ forward = vis; }
    Wrapper before(Object o, Wrapper vis){return makeWrapper(forward.before(o, vis.forward));}
    Wrapper after(Object o, Wrapper vis){return makeWrapper(forward.after(o, vis.forward));}
    FVisitor visitDefault(Object o, FVisitor v){return forward.visitDefault(o,v);}
    FVisitor exportVisitor(){ return forward.exportVisitor(); }
    boolean continueVisit() { return forward.continueVisit(); }
    abstract Wrapper makeWrapper(FVisitor v);
}

class OverfullWrap extends Wrapper<ThreadedComp<ElementSum, Overfull>>{
    OverfullWrap(){ this(new ThreadedComp(new ElementSum(0), new Overfull()));}
    OverfullWrap(ThreadedComp<ElementSum, Overfull> vis){ super(vis);}
    Wrapper makeWrapper(FVisitor v)
        { return new OverfullWrap((ThreadedComp<ElementSum, Overfull>)v); }
}

class PaddedSum extends ElementSum{
    PaddedSum(final int newsum){ super(newsum); }
    PaddedSum after(Element e, PaddedSum vis){
        ElementSum tvis = (ElementSum)super.after(e, vis);
        return new PaddedSum(tvis.sum+getPadding());
    }
    int getPadding(){ /* ... */}
}

class PaddedWrap extends OverfullWrap{
    PaddedWrap()
        { super(new ThreadedComp(new PaddedSum(0), new Overfull())); }
}
```

seen. The best solution, aside from seeing the future—since copy/paste is not reasonable— would be to add a method that returns the *shipping* weight of an `Element`. The new method can then be overridden to return $(weight+n)$ in the *padded* case. There are two major problems with this solution. First, although the functionality of `FuncCheck` should not have changed, all classes which depend on it must now be recompiled. Second, `FuncCheck` must now be re-tested, both on its own and in collaboration with other classes.

In contrast, the composed solution to this modification is also shown in Listing 9. We have made the exact same change to our `ElementSum` as was needed with `FuncCheck` but it is done in a new class, `PaddedSum`, which currently has no dependents. Because the functionality is localized, `PaddedSum` can be tested separately, then used to create the `PaddedWrap` as shown. These modules can then be further wrapped to produce full solutions or used in place of `OverfullWrap` where needed.

6. FUNCTIONS ON GENERIC LISTS

As a further exposition of the functional visitor and its usage, in this section we present generic lists and the implementation of functional abstractions such as *map*, *foldl* and *foldr*.

Extending our earlier traversal example from Figure 1, our list representation consists of `Nil`, empty, or `Cons` that contains two objects, a datum and a list. In the Listings 10 to 13 we elide some of the methods, e.g., `accept`, and the method bodies of some simple, straightforward, methods e.g. `toString()`. The class `List` acts as a wrapper class for creating and manipulating a list of `ListI` elements.

One of the most useful higher-order functions used on lists in functional programming is *map*. `Map` takes two arguments, a one

argument function and a list and applies the function to each list element in order, and returns these results in a list. Implementing generic `map` using functional visitors requires two visitors; one visitor applies the function to each element, *ApplyVisitor*, and a second visitor constructs a list from these results, *ListBuildVisitor*.

We define a parameterized interface `Function` (Listing 14) that needs to be implemented by any class used as a function. Our `Map` class (Listing 15) is parameterized over its one argument function, `F`, and the function's input, `X`, and output types, `Y`. `Map` is also a function itself from lists of `X` to lists of `Y`. The heart of the `map`'s implementation is the composition of the `ApplyVisitor` (parameterized over the list element type and the function's return type) and the `ListBuildVisitor` (parameterized over the function's return type). Computation proceeds as one expects; first we apply the function to the list elements (the `after` method in Listing 16) and the results are “cons-ed” together (the `after` method in Listing 17).³

Building incremental solutions in this way is made possible by the general `ThreadedComp`. Using the presented compositions allows us to create simpler piecewise solutions that can be tested and used to compose intermediate results without worrying about explicit store management. This is aided by the traversal style—connecting *before* and *after* methods— enabling comparison and combination of corresponding results.

7. RELATED WORK

Since the first definition of the Visitor pattern [8] there have been a number of attempts to extend the pattern to

³The implementations of *foldl*, *foldr* and a list pretty printer are available at [5].

Listing 10: List delegates to its ListI member.

```

class List<T> implements ListI<T> {
    ListI<T> elems;

    List(ListI<T> l){elems = l;}
    List(T ... args){ /*...*/ }
    T first(){return elems.first();}
    ListI<T> rest(){return elems.rest();}
    boolean isNil() {return elems.isNil();}
    String toString(){return elems.toString();}
}

```

Listing 11: ListI interface.

```

interface ListI<T>{
    T first();
    ListI<T> rest();
    boolean isNil();
    FVisitor accept(FVisitor fv);
}

```

Listing 12: Generic Cons cell.

```

class Cons<T> implements ListI<T>{
    T datum;
    ListI<T> rest;

    Cons(T d, ListI<T> r){datum = d;rest = r;}
    T first() { return this.datum; }
    ListI<T> rest() { return this.rest; }
    String toString(){ /*...*/ }
    boolean isNil() { return false; }
}

```

Listing 13: Generic Nil.

```

class Nil<T> implements ListI<T>{
    T first(){throw new FirstNilException();}
    Nil<T> rest(){throw new RestNilException();}
    String toString(){ /*...*/ }
    boolean isNil() { return true; }
}

```

- separate traversal control from the visitor's behavior [18, 7, 26, 17, 21, 23]
- provide better mechanisms to define different traversal strategies (bottom up, top down, conditional navigation etc) [23, 25, 22]
- allow for the addition of new variants on the visited data structure without affecting existing visitor definitions [19, 26, 17, 21, 12, 15, 24]
- allow the combination of visitors and the exchange of information between them [23, 26, 17, 21]

Adaptive Programming (AP) [14] allows for the separation of traversal related concerns. An AP program is defined in terms of loosely coupled contexts; structure, behavior and navigation. Structure refers to a program's class hierarchy, behavior refers to visitor computation and navigation refers to traversal strategies. Strategies define paths on a graph representation of the underlying data structure to which visitors can be attached. Demeter visitors contain one argument methods that are executed during the traversal of an object. These methods are executed before or after traversing an object. The methods are triggered if the type of the object being traversed matches the type of the visitor method's argument.

The three AP tools provide different implementation for the Demeter ideas. Demeterj [26] is a source manipulation tool and implements Demeter visitors using the visitor design pattern. DJ [17] uses reflection to traverse a collection of objects removing the need for accept method definitions. DAJ [21] uses Aspect Oriented Techniques [6, 1] (AspectJ [2] introductions) to introduce the necessary accept methods to the class hierarchy. Traversal control is separated into strategy definitions while the AP tools allow modifications to the data structure (under certain constraints [20]) without affecting the program's meaning. Composition of visitors in AP tools can be achieved by attaching an array of visitors to a strategy. By default visitors are executed in position order, however, a different execution order can be encoded as a collaboration between visitors. This results in combinations and individual visitor implementations becoming tangled and more difficult to reuse.

In functional programming, functions that work for a whole family of types are known as polytypic or data-generic [4]. Equality and pretty printing are examples of polytypic functions. As already

noted in [10] polytypic functions and adaptive methods are related. Adaptive methods also work for a whole family of types, namely the class graphs that satisfy a Demeter interface [20]. The Demeter interface plays a similar role as the kinds used in polytypic programming [9]. The visitors presented in this paper serve to better organize the non-traversal related computations of polytypic functions, but the focus is different than in the polytypic programming community. While the polytypic programming community does not focus on traversals and how to separate them, this paper decomposes polytypic functions into traversals and their computations.

In [25] a functional visitor implementation for DJ is presented where visitor methods accept a second argument and return a value. *Around* methods are introduced that take two arguments; the first is the object's type that the method will be called on; the second argument is a new type-Subtraversal- that captures the context at that point of the traversal. Subtraversal provides additional navigation control based on the current point in the traversal and a combine method is used to provide default *around* behavior. The values returned from visitors and subtraversal are not restricted, in fact they (typically) are of type Object, this forces runtime checks for cases where the visitor can return more than one type of value. Combinations of visitors are not explicitly discussed in [25], however, one can imagine compositions where the return values of one visitor are used as input to another visitor. The lack of type information and the use of downcasting makes combinations less reusable and more prone to error. String matching on field names used to direct subtraversals makes for a very flexible yet statically unchecked programs.

Vanderperren et.al. [22] introduce combination strategies in the JAsCo component system with support for AP and dynamic aspect oriented programming. JAsCo's AP system is similar to DJ in that reflection is used to traverse over Java Bean components. Visitors in JAsCo are implemented as aspect beans which extend Java Beans. An aspect bean contains *hooks* that define the points during the program execution where behavior can be augmented. The system provides reflective access to hooks at runtime allowing for their direct manipulation. *Combination Strategies* consume a list of applicable hooks and manipulate this list to remove and/or add hooks based on the traversal that needs to be followed. This technique is powerful but difficult to reason about since modifications to the list might invalidate previous combination strategies making

Listing 14: Function interface, ($f : X \rightarrow Y$)

```
interface Function<X,Y> { Y apply(X a); }
```

Listing 15: Map, ($map : (X \rightarrow Y) \rightarrow List(X) \rightarrow List(Y)$)

```
class Map <X,Y,F extends Function<X,Y>> implements Function<List<X>,List<Y>>{
    F f;
    ApplyVisitor<X,Y> appV;

    Map(F fun) { /* ... */ }
    List<Y> apply(List<X> a) {
        appV = new ApplyVisitor<X,Y>(f); ListBuilderVisitor<Y> listBuilder = new ListBuilderVisitor<Y>();
        ThreadedComp<ApplyVisitor<X,Y>,ListBuilderVisitor<Y>> vis =
            new ThreadedComp<ApplyVisitor<X,Y>,ListBuilderVisitor<Y>>(appV,listBuilder);
        ThreadedComp<ApplyVisitor<X,Y>, ListBuilderVisitor<Y>> mapV = (ThreadedComp)a.accept(vis);
        return mapV.vis2.getResult();
    }
}
```

Listing 16: ApplyVisitor applies f on each element during traversal.

```
class ApplyVisitor<X,Y> extends CompReceiver{
    Y result;
    Function<X,Y> f;

    ApplyVisitor(Function<X,Y> f){ /* ... */ }
    ApplyVisitor(Function<X,Y> f, Y r){ /* ... */ }

    ApplyVisitor<X,Y> after (Cons<X> c, ApplyVisitor<X,Y> v) {
        return new ApplyVisitor<X,Y>(v.f, f.apply(c.datum));
    }

    FVisitor visitDefault(Object o, FVisitor v){
        return new ApplyVisitor<X,Y>(this.f,this.result);
    }
}
```

Listing 17: ListBuildVisitor conses together the results from ApplyVisitor.

```
class ListBuilderVisitor<X> extends CompReceiver{
    ListI<X> result;

    <T extends X> ListBuilderVisitor(){ /* ... */ }
    ListBuilderVisitor(ListI<X> l) { /* ... */ }
    List<X> getResult(){ return new List<X>(this.result); }
    ListBuilderVisitor after (Cons<X> c, ThreadedComp<ApplyVisitor<?,X>,ListBuilderVisitor<X>> v){
        return new ListBuilderVisitor<X>( new Cons<X>(v.vis1.result,v.vis2.result));
    }
    FVisitor visitDefault(Object o, FVisitor v){
        return new ListBuilderVisitor( ((ThreadedComp<ApplyVisitor,ListBuilderVisitor>)v).vis2.result);
    }
}
```

combinations difficult to reuse correctly.

Work by Visser [23] addresses composition and traversal control in visitors. Given a small set of combinators (*e.g.* identity, Sequence, Choice etc.) these can be composed to obtain more complicated visitors with different traversal strategies (*e.g.* top down, bottom up, conditional etc.). In this system the navigation specification and the visitor composition are combined together. As such the fact that a visitor is a composition cannot be hidden when reused which limits visitor composability and modularity.

Vlissides [24] presents the *staggered* visitor that allows modifications to the visited structure without requiring modifications to existing visitors. The most generic visitor provides a general visit method, referred to as the *catch-all* operation, that delegates according to the visited objects specific type. The catch-all behavior can be overridden to accommodate new elements in the visited data structure and delegate to visitors. Similarly, the *acyclic* visitor [15] deploys multiple inheritance to break the cyclic dependency between elements and visitors. At the top of the visitor hierarchy we find an empty virtual visitor class. Each visitor is required to provide a new abstract class that extends the empty visitor and introduces visit methods for each data-type that it manipulates. Accept methods use dynamic dispatch to execute the appropriate visitor method and combination of visitor behavior is achieved through multiple inheritance. However, the hierarchy becomes proliferated as each visitor requires two new classes; one concrete and one abstract.

In SableCC [7] the most generic visitor interface contains no methods. Each new structure element extends this interface and provides a case-like method for the new variant. Each variant then provides the appropriate cast operation to the argument in its visit method (in SableCC this is called `apply`). A default visit method is also generated which can be further specialized through inheritance. In this way extensions to the visited data structure do not affect existing visitors.

In [19] reflection is deployed to model generic visiting behavior. The `Walkabout` class traverses an object structure and at each object instance obtains through reflection a list of the object's methods. If a visit method exists then it is called, else, again through reflection, the instance's members are extracted and traversal proceeds on these data members.

The *hierarchical* visitor pattern [3] defines a visitor interface that adds two methods; `visitEnter` which is called upon entering a node and `visitExit` which is called upon exit. This pattern allows for depth information as well as conditional and depth based traversals, *i.e.* cutting off a traversal after a certain depth, or depending on a nodes internal state.

8. CONCLUSION AND FUTURE WORK

We presented a modification to the visitor pattern that provides a more functional style visitor definition, and shown how they can be used to better modularize visitor functionality. We provided three general visitor compositions and have shown how these compositions can be encapsulated to produce simpler components that are both flexible and extensible.

As with most dynamic dispatch systems, our implementation relies on both reflection and runtime type-checking. In the future we would like to apply the functional visitor pattern in a more statically inclined context, *i.e.* in C++. Also, much of the code which needs to be written for a statically verifiable type-correct program has to do with certain peculiarities of Java's generics. We would like to explore alternative ways of expressing and verifying composition dependencies. Knowing how visitors expect to be composed and enforcing their assumptions would make development more reli-

able and composition less error prone.

9. ACKNOWLEDGEMENTS

We would like to thank Matthias Felleisen, Shriram Krishnamurthi, Johan Ovlinger and Jens Palsberg for their valuable feedback on earlier versions of this work.

10. REFERENCES

- [1] Aspect oriented software design, <http://www.aosd.net>.
- [2] The AspectJ project, <http://www.eclipse.org/aspectj>.
- [3] The pattern index, <http://www.c2.com/cgi/wiki?PatternIndex>.
- [4] R. S. Bird, O. de Moor, and P. F. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.
- [5] B. Chadwick, T. Skotiniotis, and K. Lieberherr. Functional visitor implementation and examples. <http://www.ccs.neu.edu/home/chadwick/FVisitor>.
- [6] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
- [7] E. M. Gagnon and L. J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] R. Hinze and A. Lø'h. Generic programming, now! In R. H. Roland Backhouse, Jeremy Gibbons and J. Jeuring, editors, *Spring School on Generic Programming, Lecture Notes in Computer Science*, 2006.
- [10] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *ACM Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
- [11] R. Kelsey, W. Clinger, and J. R. (Editors). Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [12] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 91–113, London, UK, 1998. Springer-Verlag.
- [13] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 168–177, New York, NY, USA, 2003. ACM Press.
- [14] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.
- [15] R. C. Martin. *Pattern languages of program design 3*, chapter Acyclic Visitor, pages 93–103. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [16] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
- [17] D. Orleans and K. 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.

- [18] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–81, New York, NY, USA, 1999. ACM Press.
- [19] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, Washington, DC, USA, 1998.
- [20] T. Skotiniotis, J. Palm, and K. Lieberherr. Demeter Interfaces: Adaptive programming without surprises. In *European Conference on Object Oriented Programming*, 2006.
- [21] The Demeter Group. The DAJ website. <http://www.ccs.neu.edu/research/demeter/DAJ>, 2005.
- [22] W. Vanderperren, D. Suvee, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.
- [23] J. Visser. Visitor combination and traversal control. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, pages 270–282. ACM, October 2001.
- [24] J. Vlissides. Pattern hatching - visitor in frameworks. In *The C++ report*, November/December 1999.
- [25] P. Wu, S. Krishnamurthi, and K. Lieberherr. Traversing recursive object structures: The functional visitor in demeter. In *AOSD 2003, Software engineering Properties for Languages and Aspect Technologies (SPLAT) Workshop*, 2003.
- [26] The DemeterJ website. <http://www.ccs.neu.edu/research/demeter>.