Functional Visitors Revisited

Bryan Chadwick
Northeastern University
chadwick@ccs.neu.edu

Therapon Skotiniotis
Northeastern University
skotthe@ccs.neu.edu

Karl Lieberherr
Northeastern University
lieber@ccs.neu.edu

Abstract

In object-oriented programming the visitor design pattern allows for the addition of new operations on a data hierarchy, but lends itself to scattered traversal code and makes visitors difficult to combine. Previous attempts to solve these issues have separated traversal code from the data structure but still face a lack of modularity in visitor computation making it difficult– if not impossible– to duplicate the flexibility of hand written, monolithic methods. In this paper we present a reformulation of the visitor pattern that introduces functional style traversals and visitor methods. Our modifications increase the context information available to visitor methods by way of the implicit stack of the recursive traversal, removing the need for side effects in visitor definitions. This gives visitors more flexibility during traversal, while their functional nature allows for modular, compositional solutions, leading to more reusable designs. To utilize these ideas we introduce a simple new functional *visitor-oriented* language, which allows us to build functional visitors without the syntax burdens of most object-oriented languages. To support visitor composition in our language we introduce five useful visitor combinators that can be used to develop self contained visitors and discuss motivating examples in our new language.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Design Patterns, Functional Composition, Visitors

Keywords visitor pattern, traversals, functional visitors

1. Introduction

The visitor design pattern [7] is a well known solution that supports extensions to the set of operations performed on a data structure. Typically, a visitor encapsulates an operation's behavior, and the data structure encodes traversal of the hierarchy. However, in its original incarnation the visitor pattern admits two limitations: (*i*) data structure extension requires regeneration of traversal code and (*ii*) finer traversal control must be hard-coded inside visitors yielding implementations that are difficult to reuse.

For example, consider the situation where we have a number of *Containers* that we would like to check for capacity violations (Figure 3). A container has a `capacity` and a list of `Items`—essentially a multi-tree where the leaves are `Elements` and the nodes are `Containers`. We would like to check that the combined weight of a container's `items` is not greater than its `capacity` and return the violated containers. Figure 1 (left) shows the relevant portions of the container example with traversal code based on the visitor design pattern.

In order to identify violating containers our visitor needs to maintain a running sum of weights for each container encountered during traversal. After traversing a `Container`, we need to obtain its total weight and check if this is greater than the container's capacity. Once we have the information to detect over-full containers there are three general options for filtering them:

1. Add filtering code to the *after* container method, placing violating containers in the visitor's store.
2. Use two visitors in separate visitor passes; the first visitor calculates container violations and marks them, the second visitor collects and returns marked containers.
3. Modify the return type of the *after* container method to return a violated container, or a distinguished value (*e.g.*, **null**) for no violation.

The first option creates a monolithic method, tangling violation detection and filtering inside the visitor's implementation, decreasing its reusability. The second option incurs extra runtime cost with two traversals and tangles filtering code by relying on the marking of the data structure, even if done externally, *e.g.*, with a hashtable.

The third option separates violation detection and filtering; filtering code can become part of *accept* methods, which results in tangled traversal code, or part of a second visitor whose visit methods wrap the violation detection visitor, collecting flagged containers. Using a visitor to carry out the filtering looks promising, but requires the outer visitor to be parameterized in some way over the inner visitor, including the types of instances it's interested in and its results. If we try to reuse visitors with multiple wrappers these parameterizations quickly become difficult to manage. We can, however, generalize these ideas by making return types of traversals and visitor methods consistent.

Our functional visitors use similar traversal code but return values that are also functional visitor instances. By adjusting the traversal contexts of the classic pattern we can provide access to visitor instances both before and after the traversal of an object's sub-structure. Visitors can take advantage of this fact to provide flexible implementations while remaining completely functional. With functional visitors, deciding if a container is over-full does not require a stack, but rather a comparison of visitor instances before and after traversing a container object. Once we have functional visitors and traversals it is easy to provide combinators, an abstraction over visitor compositions, that require little type information from programmers. To complete our container filtering we simply define a second visitor that collects containers and combine it with our violation detection visitor using a *Conditional Combination* (Section 5), which uses a violation detection visitor to decide whether or not to execute the collector. This allows us to break our

```
class Visitor{
  void before(Object o){}
  void after(Object o){}
}
class Container extends Item {
    /* ... */
  void accept(Visitor v){
     v.before(this);
     items.accept(v);
     v.after(this);
  }
}
class ItemPair extends ItemList{
    /* ... */
  void accept(Visitor v){
     v.before(this);
     first.accept(v)
     rest.accept(v)
     v.after(this);
  }
}
```

```
class FVisitor{
  FVisitor before(Object o, FVisitor v){return v;}
  FVisitor after(Object o, FVisitor v){return v;}
}
class Container extends Item {
    /* ... */
  FVisitor accept(FVisitor v){
     FVisitor bVis = v.before(this,v),
              iVis = items.accept(bVis);
     return bVis.after(this, iVis);
  }
}
class ItemPair extends ItemList{
    /* ... */
  FVisitor accept(FVisitor v){
     FVisitor bVis = v.before(this,v),
              fVis = first.accept(bVis),
              rVis = rest.accept(fVis);
     return bVis.after(this, rVis);
  }
}
```

Figure 1. Container traversal related code; Left: standard visitor pattern. Right: functional visitor pattern.

solution into two modular, reusable visitors: a violation detector, and a general collector; combining them to achieve the desired result.

In order to separate developers from some of the details of visitor implementation we introduce a new functional *visitor-oriented* language for writing and constructing visitor solutions.[1] Using this language we can develop each operation as a functional visitor; encapsulating collaborations as visitor combinations, leading to modular, reusable implementations.

The rest of this paper is structured as follows: the next section describes our traversal modification; Section 3 introduces our new functional visitor language; Section 4 expands on our solution to the Container example; Section 5 defines each of our visitor combinators; Section 6 discusses a more significant example of a SAT solver implementation. We review related work in Section 7 and conclude with Section 8, discussing some possibilities for future work.

2. Functional Traversals

In this section we introduce our modification to the visitor pattern traversal in the setting of the *Container* example. We show hand-written traversal methods in the style of the visitor pattern, but take this chance to point out that our functional traversals are also applicable to other forms of specification [14, 11]. Notably, we have modified a version of DJ [13] to dynamically traverse Java data structures in our functional style without any modifications to underlying classes.

The basic visitors and interesting traversal code for our data structure are defined in Figure 1 with both the canonical and new functional visitors in Java. This is different from the general notion of the pattern, as here we use *before* and *after* methods to expand visitor expressibility.[2] The visitor methods take two arguments: the current object we are traversing and a visitor instance. For *after* methods the second argument is the visitor instance returned from sub-traversals. In *before* methods it is the same instance as the one on which the method is invoked (*i.e.*, **this**). We define visitor methods on the most general type, `Object`, and assume a mechanism for specific type based dispatch.[3] Each *class* of our data structure defines an *accept* method that controls the traversal order of its members.

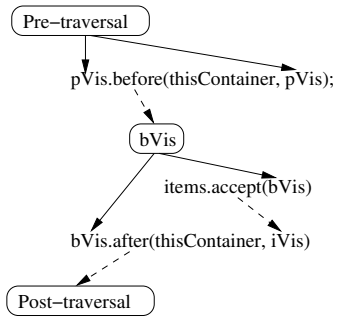

Figure 2. `Container` traversal flow diagram

In the visitor pattern, the *accept* method wraps the sub-traversals of its enclosed members with calls to the visitor's *before* and *after* methods. The difference in the case of the `FVisitor` traversal is not only the existence of a return value, but also the visitor on which we call the *after* method; we give access to the recursive stack by calling *after* on the visitor returned from the call to *before*. As an argument we pass the `FVisitor` that has been returned from sub-traversals. This means the *after* method is called on the same visitor instance that was passed to sub-traversals. In the case of a `Container`, we can compare the visitor states before and after the sub-traversal of `items`. Similarly, the *accept* method of `ItemPair` follows this convention, wrapping traversals of `first` and `rest` with calls to *before* and *after*.

Figure 2 is a visual representation of the *accept* method of the `Container` class showing how the result of each method flows into the next. We use dotted arrows for returned values and solid arrows for forwarded values. *Pre-traversal* represents the visitor that is passed to, and *Post-traversal* represents the visitor returned by, the *accept* method. Figure 4 (right) gives a specific example of how this traversal can be used effectively in our new visitor-oriented language.

Since traversal is orthogonal to visitors and their combination, for the rest of our paper we assume a traversal implementation that follows the calling pattern described here and is independent of visitor executions.

[1] Formalization of the entire functional visitor language is forthcoming

[2] Similar to the *hierarchical visitor* [3] (Section 7)

[3] Similar to Palsberg and Jay's `Walkabout`[15]

3. Visitor Language

We briefly introduce our visitor language and the differences between our function definitions and the method syntax found in most object-oriented languages, *i.e.*, Java or C#.

```
visitor Count{
  int total;
  Count(int t){ total = t; }
  Count after(Object o, Count vis)
   { Count( vis.total + 1 ); }
  int getTotal(){ total; }
}
```

The code above shows our syntax for a visitor, `Count`, which counts the number of `Objects` it visits—remember that the visitor instance passed to the *after* method is the result of possible sub-traversals. This visitor closely resembles a Java style *class* definition with a few changes. To simplify syntax we assume that all functions are public unless specified **private**. We view constructors as functions with global scope instead of treating them with a special **new** syntax as in most object-oriented languages. The other major difference is in the body of functions, where we allow assignments only in constructors; normal function bodies consist of any number of local definitions, *zero* in this case, followed by a result expression. We also eliminate the need for the **return** keyword, since we do not allow **void** functions inside visitors.

For special purpose visitors, *receivers* and *wrappers*, we introduce new syntax. A **receiver** (Figure 8) is a visitor that expects to be combined with another visitor. We declare the name of the receiver followed by the type **of** the visitor we expect to receive and the usual visitor code for constructors and methods. When a receiver expects to be combined with a *pair* of visitors we use angle brackets and commas to nest pairs of types. Fields and methods are defined as usual with the exception of the second argument to *before* and *after* methods where we use the same pair syntax to provide bindings, with names instead of types. The *left* of the pair is an instance of the visitor type we expect to receive and the *right* is a visitor instance of the receiver type. We provide access to the values of both visitors in these methods to allow communication between them.

A **wrapper** (Figure 6) is a visitor that encapsulates another visitor or combination. Wrappers declare their name followed by a binding, an initialization expression, and a visitor body. The binding can be a single name or a *pair*, which can also be parameterized, adding arguments to the wrapper's constructor (Figure 9 `FormulaReduce`). Because visitor methods are forwarded to the wrapped visitor and initialization is given by an expression, there is no need to declare constructors or *before* and *after* methods. The names in the binding are used to unfold the wrapped visitor and are useful for accessing combinations within method bodies as in Figures 6 and 9. The meaning of these visitors will be explained further when we discuss combinations in Section 5.

4. Example: Container Checking

To demonstrate the uses of the new functional visitor pattern we present our solution to the *Container Checking* [20] problem and refactor it to obtain more reusable visitors. As before 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.

Our problem is to check that containers have not been over filled, *i.e.*, that the combined weight of a `Container`'s `items` is not greater than its `capacity`. Specifically, first we would like to count the number of over-full containers in a single traversal while computing the results of sub-containers in the same pass. This problem is of interest because it is simple to write a recursive function that traverses and calculates the desired values, but is difficult to solve without side-effects when traversal and computation are separated.

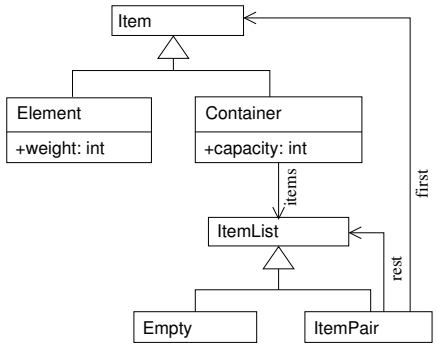

Figure 3. Container example UML class diagram

To aid our discussion, Figure 4 shows two possible solutions: one in the case of the imperative traversal, and one in the case of the functional traversal, written in our visitor language.

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-effects 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 helps in designing modular visitor solutions. Given modular visitors, we have the opportunity to compose functional visitors into larger units when building a specialized solution.

Figure 6 shows two *wrappers* that constitute our modular solutions to the container checking problem. They use a *Conditional Combination* to both count and collect any violated containers. `Checker` counts overfull containers and `Filter` collects overfull containers. The other visitors involved in the combinations, `Count` (Section 3) and `Collect`, blindly complete their tasks without knowledge of the combination. Using combinations and wrappers we can divide our visitors into modular units to implement our intended functionality.

5. Visitor Combinators

In building visitor solutions it is common for computations to require overlapping traversals of a data structure. Sometimes the functionality and/or data of a given visitor can be split into modular units and composed to produce an equivalent visitor that is easier to understand with pieces that are easier to reuse. In this section we discuss some useful visitor combination operations in the context of functional visitors and how they can be used to construct solutions.

For this discussion we introduce five forms of combination: *Conditional*, *Independent*, *Threaded*, *Fork*, and *Join*. In general, the structure of a combination is simply a visitor that is itself a *pair* of visitors. We then add behavior to our pairings to produce the desired composition. Figure 5 contains the semi-formal semantics of our various visitor combinations using a mix of familiar functional and object-oriented style syntax.

We use the functions C, I, T, F, and J to represent the constructors of *Conditional*, *Independent*, *Threaded*, *Fork*, and *Join* combinations respectively. We use P to represent any specific constructor, when behavior must be known and reconstructed, and *pair* when we only require knowledge of a combination's structure. In the syntax of our language we introduce five global construc-

```
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(weight); }
   void after(Container c){
       int last = stack.pop();
       if((weight - last) > c.capacity)
          violations++;
   }
 }
```

```
visitor FuncCheck{
  int weight, violations;

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

Figure 4. Left: Standard visitor solution, Right: Functional visitor solution

For each visitor $method \in \{before,\ after\}$ we define the functional aspects of *pair*s on a given datatype o. We assume that the default implementation of a *method* is the *identity* function. $P \in \{C, I, T, F, J\}$ captures the functional aspect of a given *pair*. We use the generic constructor, $pair(\cdot,\cdot)$, for pair structure with no functional aspects.

$$C(v_1, v_2).method(o,\ C(v_1', v_2')) \rightsquigarrow \begin{array}{l}\text{let } v_1'' = v_1.method(o,\ v_1') \\ \quad v_2'' = \text{if } v_1'' \text{ then } v_2.method(o,\ v_2') \text{ else } v_2' \text{ in} \\ C(v_1'', v_2'')\end{array}$$

$$I(v_1, v_2).method(o,\ I(v_1', v_2')) \rightsquigarrow I(v_1.method(o,\ v_1'), v_2.method(o,\ v_2'))$$

$$T(v_1, v_2).method(o,\ T(v_1', v_2')) \rightsquigarrow \begin{array}{l}\text{let } v_1'' = v_1.method(o,\ v_1') \text{ in} \\ T(v_1'', v_2.method(o,\ pair(v_1'', v_2'))\end{array}$$

$$F(v_1,\ I(v_2, v_3)).method(o,\ F(v_1',\ I(v_2', v_3'))) \rightsquigarrow \begin{array}{l}\text{let } v_1'' = v_1.method(o,\ v_1') \text{ in} \\ F(v_2.method(o,\ pair(v_1'', v_2'), \\ \quad v_3.method(o,\ pair(v_1'', v_3'))\end{array}$$

$$J(P(v_1, v_2), v_3).method(o,\ J(P(v_1', v_2'), v_3')) \rightsquigarrow \begin{array}{l}\text{let } v_1'' = P(v_1, v_2).method(o,\ P(v_1', v_2')) \text{ in} \\ J(v_1'', v_3.method(o,\ pair(v_1'', v_3'))),\end{array}$$

Figure 5. Semi-formal Combination Semantics

tors for creating combinations— `CondPair`, `IndependPair`, `ThreadedPair`, `ForkPair`, `JoinPair`. The rest of this section will explain the intuition behind the combinations and the wrapper visitors used to encapsulate them.

5.1 Conditional Combination

When we develop more complex visitors we are sometimes required to combine multiple computations into a single visitor that is less reusable. One example of this is the `FuncCheck` visitor in Figure 4 where we have a single visitor performing at least three interrelated functions. Most notably, we have hard-coded *what* to do *when* we find an overfull container. The visitor we've created would be more reusable if we could separate *when* from *what*; we then have the option of combining a given *when* with different visitors to suit our needs. To allow one visitor to control whether or not another visitor's methods are executed we introduce *Conditional Combination*.

Here we use the first visitor of the pair to decide *if* the second will be executed; if not, then it is equivalent to the identity function for the second visitor. Either way the results are paired to become the result of the combination's visitor method. Using a conditional combination we can design our container checker as an `Overfull` visitor which can be composed with any number of other visitors.

To reproduce our original solution we use the simple `Count` visitor from earlier (Section 3). The conditional `Overfull` visitor and two resulting wrappers are shown in Figure 6.

The meaning of a Conditional Composition is listed in Figure 5, where we elide the details of how the first visitor is converted to a boolean, since this is implementation dependent. In our Java implementation we use an extended functional visitor interface that includes a `continueVisit` method to facilitate this, as defined in the `Overfull` visitor.

Also in Figure 6 we use a conditional combination to create *wrappers*, named `Checker` and `Filter`. The `Checker` wrapper does exactly the same job as our single `FuncCheck` visitor (Figure 4). The name bindings of the visitors involved in the combination, here `full` and `count`, helps pull apart the results of the combination as shown in the two function definitions. Similarly, `Filter` uses `Overfull` in combination with a simple container collector, `Collect`. These visitors give us a modular and reusable solution to our container checking problem, while the wrappers provide further encapsulation. This makes it easier to use them alone or in combination with other visitors without having to know how they are implemented.

```
visitor Overfull{
  int weight;
  boolean full;

  Overfull(int w, boolean f){ weight = w; full = f; }
  boolean continueVisit(){ full; }

  Overfull after(Element e, Overfull vis)
   { Overfull(vis.weight+e.weight, false); }
  Overfull after(Container c, Overfull vis)
   { Overfull(vis.weight,
              (vis.weight-weight) > c.capacity); }
}

visitor Collect{
  List<Container> list;
  Collect(){ this( Empty<Container>()); }
  Collect( List<Container> l){ list = l; }

  Collect after(Container c, Collect vis)
   { Collect( Cons<Container>(c, vis.list)); }
}

wrapper Checker
  <full, count> = CondPair(Overfull(0, false), Count(0))
{
  int totalWeight(){ full.weight; }
  int getViolations(){ count.total; }
}

wrapper Filter
  <full, result> = CondPair(Overfull(0, false),
                            Collect())
{ List<Container> getList(){ result.list; } }
```

Figure 6. Modular visitors with Checker and Filter wrappers

5.2 Independent Combination

Often problems require multiple separate calculations over the same traversal. For instance, many programming language related problems require calculation of unrelated attributes over a graph-like representation of program text— usually called an Abstract Syntax Tree. In cases where we wish to compute multiple independent results from a given object collection it would normally require one pass for each visitor's results. Assuming we have independent visitors that require no inter-communication, we can simply combine them using an *Independent Combination*.

The results of the composition's visitor methods is simply a re-pairing of the individual results of each visitor. More than two visitors can be combined by nesting *pairs* forming a binary tree. The result of a traversal can then be decomposed into its individual visitor results to be part of further computation using a wrapper or within another combination.

5.3 Threaded Combination

Sometimes when we separate visitor functionality we would like to use the computation of a given visitor as an intermediate result in a larger context. To facilitate this kind of modular communication we define a few combinations which allow a visitor to accept, or *receive*, another visitor in its methods. These last few combinations place additional constraints on the second visitor of a pair, called the *receiver*, which gives us the ability to pass on, separate, and combine visitors in whatever way we find useful for our solution.

Threaded Combination refers to the connection between a single visitor and a single receiver. In Figure 5 we show the meaning of this combination; pairing the result of the first visitor with the earlier result of the second, and passing this pair to the second visitor's method. This is useful when we want to be able to develop and test functionality separately, then chain visitors together to get our final result.

5.4 Fork Combination

As with Threaded Combination, there are times when we would like to pass the results of a single visitor to another. In the case of a *Fork Combination* we we want to pass a single visitor to an independent pair of receivers. The result of the first visitor is passed to both receivers in the nested pair. We can use this combination to *split* a single computation into two paths without performing the first visitor's method twice. The meaning of a Fork Combination is also described in Figure 5; it is essentially an extension of the Threaded Combination to multiple receivers.

5.5 Join Combination

Much like the usage of a Fork Combination to split visitor computation, we introduce the *Join Combination* to combine the results of two visitors into a single receiver. Much like Threaded Combination, the receiver must accept a pair of visitors as its second argument to *before* and *after*, but the left visitor of the pair is expected to also be a pair of visitors. The formal specification is described in Figure 5. We use our visitor language bindings to unfold the various argument pairs and access them in the body of the receiver's methods.

5.6 Combination Wrappers

When using visitors composed of many parts it is often the case that we can abstract details of a combination to hide them from clients. To encapsulate visitor compositions we introduce the idea of a *wrapper*: a visitor which hides the details of a constructed visitor through method delegation. In general we can create a wrapper for any visitor, but they will usually be used to wrap combinations. This allows us to manage larger visitors without having to deal with composition details.

Figure 6 shows two simple wrappers, `Checker` and `Filter`. The pair, `<full, count>`, is constructed by the given expression, in this case a Conditional Combination, which binds `full` to the left visitor of the pair, of type `Overfull`, and `count` to the right visitor, of type `Count`. These names are then available throughout the wrapper's method definitions that follow, as seen in the functions defined within `Checker` and `Filter`.

It is possible to parameterize the wrapper construction expression with an argument list; *e.g.*, both `ClauseReduce` and `FormulaReduce` wrappers (Figure 9) accept a `Lit` argument during construction. The details of how these arguments are used depends on our implementation language, but in general they will become arguments to a single constructor. This is reflected in the construction expression of `FormulaReduce` where we use the given `Lit` to construct a `ClauseReduce`. Once we have constructed a wrapper, its semantics is just the semantics of the wrapped visitor or combination given in the construction expression. This allows us to simplify complex visitors and limit exposed details through the definition of methods. In the next section we show a larger, more complicated example where these wrappers help control the complexity of our visitors.

6. Extended Example: SAT Reduction

As a more complicated example that effectively shows the advantages of functional visitors using our language we present an important portion of a SAT solver. The *satisfiability problem* is one of the first problems proven to be NP-Complete [4] and there has been much research into solving various incarnations of it effectively. For this example we will focus on a typical description where a SAT *formula* is given in *conjunctive normal form* (CNF); a `Formula` is a list of *clauses*, a `Clause` is a list of *literals* (`Lit`), and a `Lit` is either *true*, *false*, or a *signed variable*: a wrapped `Var`, which can be *positive*, `Pos`, or *negated*, `Neg`. We choose unique

integers as our representation of variables and make a few adjustments to increase efficiency. Using these structures, the formula $((1 \vee \neg 2) \wedge (\neg 3 \vee 2))$ could be represented as:

```
Formula( Clause( Pos(Var(1)), Neg(Var(2))),
         Clause( Neg(Var(3)), Pos(Var(2))))
```

Our task: given an assignment to a *variable*– essentially, a *literal* is an assignment of a single variable– reduce the literals within each clause based on boolean logic and a simple function *reduce*. The first argument to *reduce* is the literal representing a single assignment, the second is a `Pos` or `Neg` literal we wish to reduce. The reduction cases are shown in Figure 7. Note that when the two literals are the same *reduce* produces *true*; if they refer to the same variable but differ in sign it produces *false*; otherwise it returns the original literal.

$$
\begin{aligned}
reduce(Pos(v),\ Pos(v)) &= true \\
reduce(Neg(v),\ Neg(v)) &= true \\
reduce(Neg(v),\ Pos(v)) &= false \\
reduce(Pos(v),\ Neg(v)) &= false \\
reduce(\ell,\ \ell') &= \ell' \quad \text{if } \ell \text{ and } \ell' \text{ do not refer to the same variable}
\end{aligned}
$$

Figure 7. Single Literal Reduction

One strategy for solving SAT instances is to successively *reduce* literals in a given CNF formula based on some ordering of variable assignments. After we have reduced all literals to *true* or *false* we check if this ordering produced a satisfying assignment based on the usual boolean rules of conjunction and disjunction. We can then modify our assignment and ordering to limit our search space and reduce based on any new information learned. It would, however, be rather inefficient to continue reducing disjunctions which already contain the literal *true*; for this situation we introduce a new `Clause` variant, `AllTrue`. To help in the identification of *unsatisfying* assignments we introduce a final variant `AllFalse` for clauses which only contain the literal *false*. Thus, we no longer need *true* and *false* literals in our implementation; they simply help in our understanding of the problem.

Now we can think of how to solve the reduction problem using a collection of visitors; for this discussion we will define the process by which we produce a reduced `Formula` given a `Lit` representing a single variable assignment.

Figure 8 contains a version of our basic SAT reduction visitors. These are the results of thinking about each step of the reduction process and choosing a level of *abstraction* that balances complexity and composition. Much like the division of functional abstractions when writing a program, we can organize our compositions and wrappers at different levels to increase clarity, modularity, and reusability. We start with a discussion of the individual visitors, and later see how these are combined into wrappers to form our final `FormulaReduce` visitor.

Each visitor performs a simple action to produce various values. `ClauseTrue` tracks whether or not the current `Clause`– disjunction– has been forced to *true* by some `Lit` reduction, storing this as a boolean. To make the visitor flexible we parameterize it over the current *target* `Lit` (or assignment), passing it through to any `ClauseTrues` we construct. *Before* an `AllTrue` clause we set the boolean value to *true*, for any other `Clause` we set it to *false*; checking *after* each literal whether it is the same as the *target* which corresponds to the first two *reduce* rules in Figure 7. `LitReduce` performs the reduction of a single literal by seeing if they refer to the same variable: if so then we produce **null**, otherwise we keep it— **null** signals to `ListBuilder` that the object should be skipped. In the case that the assignment literal and the current `Lit` refer to the same variable, then it would reduce to *true* or *false* as in the first four cases of *reduce*. Either way we can ignore the `Lit`: if *true* then `ClauseTrue` will keep track of it to eventually produce and `AllTrue` clause; if *false* we use the boolean rule for disjunction: $(false \vee B) \rightarrow B$. If the two *Lit*s do not refer to the same variable then we store the literal for the `ListBuilder` to collect, corresponding to the last case of *reduce*.

`ListBuilder` is parameterized over two *types*, `Targ` and `Reset`; `Targ` is the *type* of objects we wish to collect and and `Reset` triggers us to *restart* the list building. The corresponding *before* and *after* methods collect non-null `Targ` objects and reset the list to `Empty` respectively. `ListBuilder` is assumed to be composed with an `ObjectGetter`; we use this visitor as a simple interface to enable `ListBuilder` to be as reusable and generic as possible. For example, we could use this `ListBuilder` in place of our `Collector`, in combination with a simple `ObjectGetter` and our `Overfull` from earlier to collect all overfull containers.

Note that the *receiver*, `ListBuilder`, declares the type it expects to receive, `ObjectGetter`, to be a single visitor not a pair, implying that it will take part in a Threaded Combination. In general, we can make sure that the *types* of all combinations, receivers, and wrappers are correct by checking the various expectations of each visitor involved.

Figure 9 shows the rest of our reduction visitors and wrappers. The first receiver, `ClauseMaker`, receives a pair of `ClauseTrue` and a combination involving a `ListBuilder`; '`?`' is a wild-card that means any visitor type. The *after* method pulls apart the structure of the combinations and sees which type of clause should be *made*; the `AllTrue` and plain `Clause` cases are as discussed earlier. An `AllFalse` clause is created when the reduced clause is not `AllTrue` *and* the list of `Lit`s is empty, meaning each literal could have only been reduced to *false*.

To help us manage the complexity of `FormulaReduce` we introduce the intermediate `ClauseReduce` wrapper. The only interesting portion of the combination, from `ClauseReduce`'s point of view, is the `ClauseMaker` which is bound to `clsMk`. We make the newly reduced `clause` available to another list builder by overriding `getObject`. When finally composed with a `ListBuilder` we can produce a list of `Clauses` which can then be converted into a `Formula`. A `FormulaMaker` is a receiver that performs this conversion; much like `ClauseMaker`, its *after* method pulls out the list of `Clauses` and produces a `Formula`. Our final task is to give a wrapper which constructs the combination correctly and provides functions to access the results. `FormulaReduce` does just that; we can now construct it by providing a `Lit` which represents our single `Var` assignment.

One thing to note about this example is the conceptual simplicity of the individual visitors; once we understand how combinations can be built and how visitors involved can communicate, we can separate and modularize our functionality. This allows us to develop pieces of a solution which are easier to write, test, understand, and later reuse.

7. Related Work

Adaptive Programming (AP) [11] allows for the separation of traversal related concerns into three parts; (*i*) the data structure to be traversed, (*ii*) a navigation specification (or *strategy*), and (*iii*) an imperative visitor with *before* and *after* methods. The three AP tools DemeterJ [21], DJ [13], and DAJ [17], provide different implementations of AP in Java. DemeterJ is a source manipulation that uses domain specific languages and the visitor design pattern, DJ uses reflection to dynamically traverse a collection of objects, and DAJ uses Aspect Oriented Techniques [5, 1], AspectJ [2] introductions, to provide the necessary traversal methods in the class

```
visitor ObjectGetter<T>{  T getObject(){ null; } }

visitor ClauseTrue{
    boolean clause;
    Lit targ;

    ClauseTrue(Lit t){ this(t,false); }
    ClauseTrue(Lit t, boolean b){ clause = b; targ = t; }

    ClauseTrue before(AllTrue allT, ClauseTrue vis)
      { ClauseTrue(targ, true); }
    ClauseTrue before(Clause cls, ClauseTrue vis)
      { ClauseTrue(targ, false); }
    ClauseTrue after(Lit lit, ClauseTrue vis)
      { ClauseTrue(targ, vis.clause || (targ.equals(lit))); }
}

visitor LitReduce extends ObjectGetter<Lit>{
    Lit targ, current;

    LitReduce(Lit t, Lit curr){ targ = t; current = curr; }

    LitReduce after(Lit lit, LitReduce vis)
      { LitReduce(targ, if (targ.sameVar(lit)) then null else lit); }
    Lit getObject(){ current; }
}

receiver ListBuilder<Targ, Reset> of ObjectGetter<Targ>{
    List<Targ> list;

    ListBuilder(){ this(Empty<Targ>()); }
    ListBuilder(List<Targ> l){ list = l; }
    ListBuilder(Targ first, ListBuilder<Targ, Reset> old)
      { this(Cons<Targ>(first, old.list)); }

    // Check for 'Reset'
    ListBuilder<Targ, Reset> before(Reset obj, <getter, builder>){
        ListBuilder<Targ, Reset>();
    }
    // Check for 'Build'
    ListBuilder<Targ, Reset> after(Targ obj, <getter, builder>){
        Targ toAdd = getter.getObject();
        if (toAdd != null)
        then ListBuilder<Targ, Reset>(toAdd, builder)
        else builder;
    }
}
```

Figure 8. Base SAT Reduction Visitors

hierarchy. The abstraction of traversal paths using strategies allows modifications to the data structure (under certain constraints [16]) without affecting visitor behavior. Composition of visitors in AP tools is 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 that become tangled and more difficult to reuse.

In [20] a functional visitor implementation for DJ is presented where `around` visitor methods are introduced that take two arguments: the first is the object type that the method will be called on, the second argument is a new type–`Subtraversal`. A `Subtraversal` captures the current context of the traversal providing additional navigation control. A `combine` method is used to provide default behavior for the `around` method. The values returned from visitors and sub-traversals are unrestricted, in fact they are typically of type `Object`, forcing runtime checks for cases where the visitor could return more than one type of value. Combinations of visitors are not explicitly discussed, however, one can imagine compositions where the return values of one visitor are used as input to another visitor, but the lack of type information and the use of downcasting would make combinations less reusable and error prone.

Work by Visser [18] 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.). The fact that traversal navigation specification and the visitor composition are combined and visitor compositions cannot be hidden when reused limits the effectiveness of visitor compositions and reduces modularity.

Ovlinger and Wand [14] propose a domain specific language as a means to specify recursive traversals of object structures used with the visitor pattern [7]. The domain specific language further allows for the addition of arguments to traversal methods and combination of intermediate results from sub-traversals supporting functional style traversal and visitor-like definitions. The language provides traversal flexibility at a higher level than hand-coded traversals, but is not robust with respect to data structure changes, unlike AP. Though the concept of functional style visitors and composition are not discussed fully, we have adapted the language for specifying our functional visitor traversals as an alternative to hand-coded or dynamic methods of traversal.

There have been a number of simple extensions to the original definition of the Visitor Pattern [7] but do not address visitor combinations explicitly.

```
receiver ClauseMaker of <ClauseTrue, <?, ListBuilder<Lit, Clause>>>{
    Clause cls;

    ClauseMaker(){ cls = null; }
    ClauseMaker(Clause c){ cls = c; }
    ClauseMaker(List<Lit> l){ this(Clause(l)); }

    ClauseMaker after(Clause cls, <<clsTrue, <?, builder>>, clsMk>){
        if(clsTrue.clause)
        then Clausereduce(AllTrue())       // This disjunction is all True
        else if(builder.list.isEmpty())
             then ClauseMaker(AllFalse())  // All Lits reduced, none were True
             else ClauseMaker(Clause(builder.list));  // Unassigned Lits remain
    }
}

wrapper ClauseReduce extends ObjectGetter<Clause>
    < < clsTrue, < litRed, buildCls>>, clsMk>(Lit lit) =
    (JoinPair(IndependPair(ClauseTrue(lit),
                           ThreadedPair(LitReduce(lit, null),
                                        ListBuilder<Lit, Clause>())),
              ClauseMaker()))
{   Clause getObject(){ clsMk.clause; } }

receiver FormulaMaker of ListBuilder<Clause, Formula>{
    Formula form;

    FormulaMaker(){ form = null; }
    FormulaMaker(List<Clause> lst){ form = Formula(lst); }

    FormulaMaker after(Formula form, <builder, formMk>)
      { FormulaMaker(builder.list); }
}

wrapper FormulaReduce
    < < clsRed, buildForm>, formMk>(Lit lit) =
    (ThreadedPair(ThreadedPair(ClauseReduce(lit),
                               ListBuilder<Clause, Formula>()),
                  FormulaMaker()))
{   Formula getFormula(){ formMk.form; } }
```

Figure 9. SAT Reduction Receivers and Wrapper Combinations

Vlissides [19] presents the *staggered* visitor that allows modification of the visited structure without the changing of existing visitors. The most generic visitor provides a general visit method, referred to as the *catch-all* operation, that delegates according to the visited object's specific type. The catch-all behavior can be overridden to accommodate new elements in the visited data structure and delegate to other visitors, however, visitor combinations are not addressed.

Similarly, the *acyclic* visitor [12] 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 and accept methods use dynamic dispatch to execute the appropriate visitor methods. Combination of visitor behavior is achieved through multiple inheritance, but the hierarchy becomes proliferated as each visitor requires a new concrete and abstract classes.

SableCC [6] uses a variation of the hierarchical visitor [3] with a generic visitor interface that 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 minimizing the effect of visited data structure extensions on existing visitors.

With the same goals as visitors in object oriented programming, the Scrap your Boilerplate (SyB) series of papers [8, 9, 10] present a lightweight approach to generic programming in Haskell based on data structure traversals and combinations. The goal of SyB is to automatically write code that traverses data structure while the developer provides functions that perform operations on them. Generic traversal functions take a combinator that specifies which of the nodes in the data structure a given function should be applied to. The traversal combinator's argument is itself a function, a type extender, that takes the function to be called at each node as an argument. The type extender function behaves as its argument function when applied to nodes of interest and as the identity function when applied to uninteresting nodes. Interesting nodes are the nodes whose types match the argument type of the function given to the type-extender.

Functional visitors take a similar approach by encapsulating a group of functions, over interesting data types, into a single visitor. Because SyB provides a very general collection of strategies, there is less context information available to traversal functions, which reduces their flexibility.

8. Conclusion and Future Work

We present a modification to the classic visitor pattern that provides a functional style visitor definition, which increases the amount of context information available to visitors during traversal. We show how this can be used to better modularize visitor functionality by introducing a visitor-oriented language for constructing visitors and their combinations. We also describe five general visitor combinations and describe how these can be used to encapsulate simple, modular, component-like visitors that are both flexible and extensible, while remaining independent of traversal specifications.

We are currently working on the formalization of our functional visitor language, concentrating on visitor combinator semantics and type inference. In addition we would like to explore various visitor and combination optimization strategies that are generally associated with functional and applicative programming. As part of our implementation we would like to be able to automatically generate many useful functional visitors for data structures, *e.g.*, `Copy` and `Print`, as most AP tools do. After finishing the complete language definition we would also like to compare our visitors to programming styles used in different functional languages, such as Scrap your Boilerplate [8, 9, 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://c2.com/cgi/wiki?HierarchicalVisitorPattern.

[4] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[6] 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.

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

[8] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[9] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[10] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, Sept. 2005.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] T. Skotiniotis, J. Palm, and K. Lieberherr. Demeter Interfaces: Adaptive programming without surprises. In *European Conference on Object Oriented Programming*, 2006.

[17] The Demeter Group. The DAJ website. http://www.ccs.neu.edu/research/demeter/DAJ, 2005.

[18] 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.

[19] J. Vlissides. Pattern hatching - visitor in frameworks. In *The C++ report*, November/December 1999.

[20] 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.

[21] The DemeterJ website. http://www.ccs.neu.edu/research/demeter.