A Language for Specifying Recursive Traversals of Object Structures

Johan Ovlinger and Mitchell Wand*
College of Computer Science
Northeastern University
360 Huntington Avenue
Boston, MA 02115, USA
+1 617 373 2464
{johan,wand}@ccs.neu.edu

Abstract

We present a domain-specific language for specifying recursive traversals of object structures, for use with the visitor pattern. Traversals are traditionally specified as iterations, forcing the programmer to adopt an imperative style, or are hard-coded into the program or visitor. Our proposal allows a number of problems best approached by recursive means to be tackled with the visitor pattern, while retaining the benefits of a separate traversal specification.

Keywords: Visitor Pattern, Recursive Programming, Separation of Concerns

1 Introduction

The visitor pattern [GHJV95] allows a programmer to write behavior that traverses an object structure without embedding assumptions about the structure into the computational code. This is a result of separating *navigational* code from code that performs the computation. In most versions of the visitor pattern, the exact sequence in which parts of an object are visited is hard-coded, either into the object or into the visitor.

We refer to the sequence of object visits that the navigational code produces as the *traversal* of the visitor, and the ideal object structure as the *class graph* of the application. Where the object structure only contains concrete objects, the class graph may contain contain abstract classes and interfaces. Since the code implementing the traversal necessarily has a detailed encoding of the class graph, binding the visitor and the navigational code together harms maintainability and reuse.

An alternative to hard-coding is to separate the navigation from not only the visitor's behavior, but the visitor as a whole. Iterators [GHJV95] and strategies [Lie96] have been suggested for this purpose. Both present the object structure as a sequence of visit events, thereby linearizing the traversal and enforcing an imperative style to program the visitor.

We propose a domain specific language to express recursive traversals of object structures. By not linearizing the object structure being traversed, our language allows recursive computations to be expressed using the visitor pattern.

A range of problems are succinctly solved by synthesizing the result from sub-results. By making value-passing explicit in our specification language, we are able to naturally express recursive algorithms.

Our design allows an object to be used with many different traversals, and a traversal to be used with many different visitors. Factoring behavior into class graph, traversal, and visitor makes a program more robust under evolution of the class graph. A change in the class graph is likely to require only local maintenance to the traversal definitions. Often, the visitors need not be changed.

Some features of our approach are:

1. allowing the programmer to specify the order in which the parts of an object are traversed,
2. allowing the current node to be visited several times in the course of a traversal,
3. allowing the traversal to control its behavior dynamically.
4. providing a convenient mechanism for values to be returned from visits,
5. providing a convenient mechanism for iterating a traversal over collections of objects, and
6. allowing traversals to be named for re-use.

Traversal specifications are translated into Java classes, and are thus first class values that can be stored in variables or passed as arguments.

2 Related Work

Gamma et al. [GHJV95] present the *visitor pattern*, which proposes to separate behavior from navigation. The former is put into a separately compiled class and the latter is put into the class diagram. The intended use of the visitor pattern is to create software robust under the addition of new

*This work has been partially supported by the Defense Advanced Projects Agency (DARPA) and Rome Laboratory, under agreement number F30602-96-2-0239. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency, Rome Laboratory, or the U.S. Government. This research also supported in part by the National Science Foundation under grants CCR-9629801 and CCR-9804115.

To appear in the Proceedings of OOPSLA '99.

behavior, but not under changes to the class structure. In many cases, the navigational aspect is trivial, always covering the entire object graph.

Lieberherr [Lie92, Lie96] argues that a complete traversal of the object graph is often not what is needed, and suggests that the visitor pattern should be modified so that the navigational aspect is packaged up in a concise specification. By gathering all the information in one place, the behavior of a traversal and visitor may be conveniently analyzed without hunting through the program for it. Lieberherr proposes specifying the navigational aspects using a navigation language that leaves unimportant details unspecified. These details are inferred by the system at compile time from information in the class diagram. The benefit of Lieberherr's under-specified navigation language is that for many changes to the class diagram, the meaning inferred by the system will be unchanged. The goal is that *none* of the program should require modification; Lieberherr calls this system *Adaptive Programming.*

A pitfall of this approach is "surprise paths." A surprise path occurs when a change to the class diagram leads the system to infer a path that was not intended. These surprise paths can alter the intended semantics of the program, and by virtue of being surprising, can be difficult to locate. Most importantly, by leaving unimportant details unspecified, the navigational aspect language is a poor match for recursive programming. Recursive programming seems to require that we are explicit at all stages how the result is propagated and computed. This is obviously in conflict with the desire to leave details unspecified.

Our proposal maintains the separation between navigation and behavior, but sacrifices some degree of compactness for *locality* and *robustness*, meaning that changes to the program induced by changes in the class definitions should be localized and easily implemented. We aim for least surprise, whereas Adaptive Programming aims at least modification.

Current implementations of Adaptive Programming require the class diagram to be recompiled for each modification to a navigational aspect [LO97]. Our organization also allows navigational aspects to be added without recompiling the class diagram.

3 Traversal Specification

3.1 An example

To motivate our discussion, we will start with a very simple program written in Java. The example we choose is to sum all the nodes in a tree. The class diagram of our example is shown in fig 3.1.

A Java program in the visitor style looks like:

```
abstract class Tree {
   abstract void visitAllLeaves(TreeVisitor tv);
}
class Node extends Tree {
   Tree left,right;
   void visitAllLeaves(TreeVisitor tv)  {
     left.visitAllLeaves(tv);
     right.visitAllLeaves(tv);
   }
}
class Leaf extends Tree {
   int val;
   void visitAllLeaves(TreeVisitor tv) {
     tv.visitLeaf(this);
   }
```

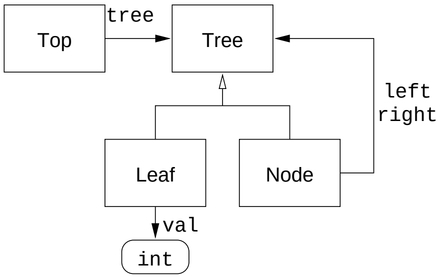


Figure 1: Class diagram for simple tree

Traversal	::=	`traversal` TraversalName = TraversalEntry TraversalEntry*
TraversalEntry	::=	ClassName ⇒ Action*
Action	::=	`traverse` PartName ;
	|	`visit` VisitName ;
	|	`super` ;

Figure 2: The skeleton of the Traversal Specification grammar

```
}
interface TreeVisitor {
   void visitLeaf(Leaf l);
}
class SumVisitor implements TreeVisitor {
   int acc = 0;
   void visitLeaf(Leaf l) { acc += l.val; }
}
class Top {
  Tree tree;
  void visitAllLeaves(TreeVisitor tv) {
    tree.visitAllLeaves(tv);
  }
  int sum() {
    SumVisitor v = new SumVisitor();
    this.visitAllLeaves(v);
    return v.acc;
  }
}
```

In the coming sections we will show different versions of the same program.

3.2 Some Syntax

We wish to succinctly describe traversals over the object graph. We do this by specifying a list of actions to be taken when entering an object of a specified class. As a first approximation, the traversal specification [1] has the grammar shown in figure 2.

Before explaining the semantics of the grammar, we transcribe our summing example to this style.

```
traversal visitAllLeaves =
```

[1] Lieberherr[Lie96] also uses the term "traversal specification", but refers to a slightly different concept.

```
        Top  => traverse tree;
        Node => traverse left;
                traverse right;
        Leaf => visit visitLeaf;

abstract class Tree {}

class Node extends Tree {
  Tree left, right;
}
class Leaf extends Tree {
  int val;
}
class SumVisitor implements visitAllLeaves_Visitor {
  int acc = 0;
  void visitLeaf(Leaf l) {
    acc += l.val;
  }
}
class Top {
  Tree tree;
  int sum() {
    SumVisitor v = new SumVisitor();
    visitAllLeaves.traverse(this,v);
    return v.acc;
  }
}
```

The code has been reorganized. The navigation is now specified in the traversal `visitAllLeaves`. This organization is immediately an improvement over the plain Java code, in that all the code that is part of a behavior has been grouped together.

The traversal is started by invoking the static method `traverse` on the class visitAllLeaves, which is generated by the system from the traversal specification. We pass the object (`this`) to be traversed and the visitor (`v`) to traverse it with to the `traverse` method. The visitor is passed as the first argument to the traversal when the traversal is invoked, after which it is implicit.

The traversal proceeds by inspecting the current object o to determine its class. The list of TraversalEntries is scanned to find the most specific entry for the object. Once an entry is found, the corresponding list of actions is executed in order of occurrence. The action `traverse p` invokes the traversal recursively on the object $o.p$. The action `visit v` invokes method v of the visitor with the current object o as argument.

In general, there may be several applicable entries for an object, as all of its superclasses may have entries. To call the next most specific entry, the `super` directive is provided. Thus, both overriding and extending behavior is supported. If an object is traversed for which no entry can be found, no action is performed, and the void result is returned.

The discrimination of which is the most precise entry for an object could be done using Java's introspection facilities, but as we have stated that we will be generating Java code from the traversal specification, we instead use inheritance information from the class graph to generate efficient Java code. In later sections, we will be using information from the class graph to provide types for variables in the traversal.

3.3 Changing the Class Diagram

The motivation behind our suggestion was both to make recursive programming less painful (by automatically generating lots of code from a minimal functional description) and also to make code robust to changes in the class diagram.

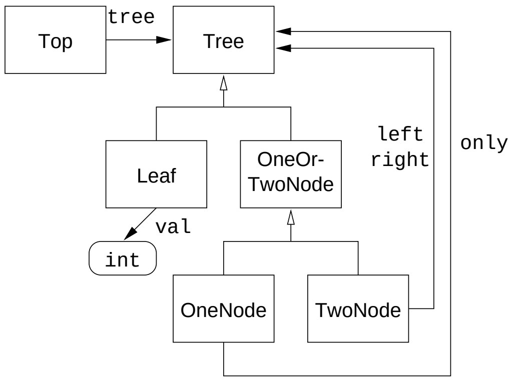


Figure 3: Class diagram for a more complicated set of trees

To illustrate the changes needed, let us modify the class diagram of our running example slightly, to have Nodes with just one child as well as the two we've already seen. See figure 3.

To keep the names meaningful, Node has become TwoNode. The new traversal specification becomes:

```
traversal allLeaves() =
        Top     => traverse tree;
        TwoNode => traverse left;
                   traverse right;
        OneNode => traverse only;
        Leaf    => visit visitLeaf;
```

Apart from the global renaming of Node to TwoNode no further changes are necessary. Obviously, not all modifications to the class diagram are as simple, but the modifications required will often be similar. We make this claim because looking back at the program, we see that the traversal specification makes only local assumptions about the class diagram. Specifically, the traversal only assumes that a given class will be able to access the parts that the TraversalEntry for that class wishes to traverse.

It can be argued that the corresponding modifications needed for the plain Java code are almost as painless as the changes to the traversal specification. The Java code, however, will require changes to separated parts of the program, while the traversal specification organizes the code so that all the changes are localized to the traversal specification.

3.4 Passing Parameters and Return Values

It is important to be able pass arguments to and to return results from traversals and visitors. To do this, we extend our traversal specification grammar to that in figure 4.

In this grammar, both visitors and traversals take arguments and can return results. The result can either be bound to a traversal variable (as opposed to instance variables on the current object) for subsequent use or returned to the caller. There is also a special variable `host`, which is bound to the host object that is being traversed.

The grammar also allows for types to be optionally specified. While we perform some type inference, it is mainly as a convenience to avoid having to repeat type declarations.

Traversal	::=	`traversal` TraversalName [Parameters] = TraversalEntry*
TraversalEntry	::=	ClassName [`returning` Type] ⇒ Action*
Action	::=	[Type] VariableName = ActionRhs ;
	|	ActionRhs ;
ActionRhs	::=	`traverse` PartName Arguments
	|	`visit` VisitName Arguments
	|	`super` Arguments
	|	VariableName
Parameters	::=	(Parameter [, Parameter]*)
Parameter	::=	[Type] VariableName
Arguments	::=	(Argument [, Argument]*)
Argument	::=	VariableName

Figure 4: A more complete Grammar

When dealing with collections, for instance, we require that the programmer specify the type of the collection's elements. Section 3.6 covers collections in greater detail.

Let us rework our example using some of the new features, so that the accumulator is kept in the traversal rather than in the visitor.

```
traversal visitAllLeaves(int acc) =
        Top     => traverse tree(acc);
        TwoNode => int lacc = traverse left(acc);
                   traverse right(lacc);
        OneNode => traverse only(acc);
        Leaf    => visit visitLeaf(acc);

abstract class Tree {}
class Node extends Tree {
  Tree left, right;
}
class Leaf extends Tree {
  int val;
}
class SumVisitor implements visitAllLeaves_Visitor {
  void visitLeaf(Leaf l, int sumsofar) {
    return l.val+sumsofar;
  }
}
class Top {
  Tree tree;
  int sum() { return
    visitAllLeaves.traverse(this,new SumVisitor(),0);
  }
}
```

Now the visitor in the example has no internal state; the accumulator is passed as a return value/parameter[2]. Notice also that we now pass one more argument than before to the `traverse` method; this is because the traversal now takes an argument.

The threading of the accumulator mimics the original example, but doesn't illustrate the recursive style we prefer. It would be more elegant to have the computation proceed in the natural recursive fashion:

[2]Although, in general, visitors can still have state.

```
traversal visitAllLeaves() =
        Top  => traverse tree();
        Node => int lacc = traverse left();
                int racc = traverse right();
                visit combine(lacc,racc);
        Leaf => visit visitLeaf();

abstract class Tree {}
class Node extends Tree {
  Tree left, right;
}
class Leaf extends Tree {
  int val;
}
class SumVisitor implements visitAllLeaves_Visitor {
  void visitLeaf(Leaf l) { return l.val; }
  int combine(Node n,int left, int right) {
    return left+right;
  }
}
class Top {
  Tree tree;
  int sum() { return
    visitAllLeaves.traverse(this,new SumVisitor());
  }
}
```

The program now looks much more like what a functional programmer would expect. The result is inductively computed from the results of the subtrees. The traversal is responsible for passing data, and the visitor is responsible for computing data. The two sub-results are combined by calling a method on the visitor.

3.5 Traversing Non-Parts

It is sometime desirable to traverse objects that are not connected to object structure being traversed. An example of this would be to traverse an object that is the result of some previous computation. One way to achieve this is to make a visitor method that invokes the traversal on the object argument, but this approach has the drawback of hard-coding the name of the traversal into the visitor.

Instead, we extend the `traverse` directive to traverse over traversal variables in addition to instance variables of the host object. Traversal variables shadow the instance variables of the host. To reflect this added capability, the nonterminal PartName in our grammar is changed to PartOrVarName, but the language parsed is unchanged.

We give an example of how this facility may be used in the next section.

3.6 A special case: Collections

Collections are so generally useful that it makes sense to deal with them specially. We propose to do this by adding an operator called `for-each` to the traversal specification language. The grammar is shown in figure 5. The goal of `for-each` is to emulate a fold, or reduction, [FH88] over the collection. Since our target language is Java, we use the iteration construct offered by its class library: Enumerations. All we require of the Collection class is that it has a method called `elements()` that returns an Enumeration. Our strategy is to use the methods provided by Enumeration to access each element in the collection, perform a traversal on each element thus accessed, and combine the results using the method name passed to `for-each`.

```
ActionRhs ::= traverse PartOrVarName Arguments
            | visit VisitName Arguments
            | super Arguments
            | method MethodName Arguments
            | for-each PartOrVarName
                MethodName Argument Arguments
            | VariableName
```

Figure 5: Reducing collections.

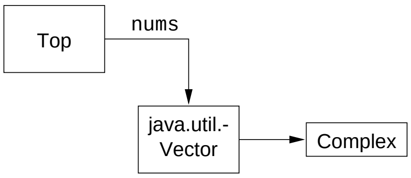


Figure 6: A Vector of Complex numbers

The collection to be traversed is in PartOrVarName of the current host. This indirection is necessary because there might be several collections (`Vectors`) in our class graph, and we need to generate different traversal code for each, depending on the types of the elements it contains.

`for-each` calls the MethodName method of the visitor once for the result of each element in the collection, passing it two arguments: the result of traversing the element and the accumulated result. The first Argument is the accumulator's initial value, and the remaining Arguments (in parentheses) are passed to the traversal of each element. The result of traversing the collection is the the accumulated result.

To illustrate, let us analyze the use of `for-each` to find the maximum magnitude of a list of Complex numbers. Let our class diagram look like figure 6. Top has a part called `nums` that leads to a collection of Complex.

To traverse the collection, we might write:

```
traversal foldComplex() =
    Top   => double maxinit = visit getinit();
             for-each nums calcmax maxinit();
    Complex
      returning double => visit magnitude();

class MaxVisitor implements foldComplex_Visitor {
    double getinit(Vector v) { return 0.0; }
    double calcmax(double magn, double maxsofar) {
        return Math.max(magn,maxsofar)
    }
    double magnitude(Complex comp) { return
      Math.sqrt(comp.re*comp.re+comp.im*comp.im);
    }
}
class Top {
    Vector nums;
    double findmax() { return
      foldComplex.traverse(this,new MaxVisitor());
    }
}
```

When `for-each nums calcmax maxinit()` is invoked by the traversal, the current object is an instance of Top, and the collection to be traversed is in its `nums` instance variable. An Enumeration is created by invoking the method `elements()` on the collection. The traversal iterates over all the elements of the enumeration, using code that is functionally equivalent to the following snippet:

```
double foreach(
  MaxVisitor v, Enumeration enum, double initacc) {
  double acc = initacc;
  while (enum.hasMoreElements()) {
    Complex elem = (Complex) enum.nextElement();
    double traversal_result =
      foldComplex.traverse(elem,v);
    acc = v.calcmax(traversal_result, acc);
  }
  return acc;
}
```

Three issues are worth noting.

1. As mentioned above, we use type information from the class diagram to determine that we are traversing a collection of Complex, which allows us to perform the proper cast to the result of `nextElement()`.
2. However, the class diagram does not contain enough information for us to determine all the types that we need to know in order to produce the above snippet of code. One such type is the type of `traversal_result`, which we assert is double. We must do type inference to determine these types, which is discussed in section 5.2. In this particular case, the fact that `maxinit` is a double is not enough; we also need to know that the result of traversing Complex is `double`. In this above code, we explicitly annotate the entry for Complex, but in other cases this might be inferable by the system.
3. After retrieving an Object from a collection, we cast it to the type we know it should have, and then proceed to traverse it.

In some cases, we don't want to traverse entire collections, but instead only a subset thereof. In section 3.5 introduced the concept of traversing variables from the traversal. We can use this facility to create a suitable collection before traversing it.

For example, if we had wished to only find the maximum magnitude of every other Complex value, we might have written our traversal like this:

```
traversal foldComplex() =
  Top => double maxinit = visit getinit();
         Vector(Complex) mynums = visit makemynums();
         for-each mynums calcmax maxinit();
  Complex
    returning double => visit magnitude();
```

and added the following method to the visitor:

```
  Vector makemynums(Top t) {
    Vector v res = new Vector();
    Enumeration e = t.nums.elements();
    while (e.hasMoreElements()) {
      res.addElement(e.nextElement());
      if (e.hasMoreElements()) nextElement();
    }
    return res;
  }
```

The makemynums method makes a collection containing every other element from the original, which is then traversed using for-each.

3.7 Controlling the Traversal

We would like the traversal to be able to make decisions at runtime as how to proceed. For example, we might be searching for some item in a binary tree; depending on the value stored in the node, we need to search the left subtree or the right. Once we have found the value, the traversal is done.

To this end, we introduce the thunk directive, which produces a thunk from a list of Actions. Its grammar is in figure 8. A thunk can reference the variables that are visible where it is declared, but cannot change their values. Thunks are Java objects that have only one method — invoke. They are typically passed to methods, but can also be invoked by the traversal through the invoke directive.

The return type is determined from the body of the thunk. Since thunks only have return types, this is the basis for their class name. A Thunk_int is a thunk that returns an int when invoked.

To illustrate, figure 7 shows how thunks might be used with a binary search tree. Each element of the tree is either a Node, containing two subtrees, or a Leaf, containing an Item. At each Node, we create two thunks - one to search either subtree - and pass them to the visitor. The visitor determines which subtree to search and invokes the proper thunk.

Using thunks, a wide variety of quite complex behaviors can be programmed. In addition to searching binary trees, they provide a convenient way to traverse cyclic objects. In some situations it is desirable to repeatedly perform the same traversal, for example iterating a stateful operation to a fixpoint. Thunks may be invoked several times in a row, allowing such computations to expressed without a creating a separate traversal for the iterative computation. Instead of invoking the thunks immediately, the visitor can decide to store them in some table, and invoke them only if it turns out that their results are needed at a later time.

A small extension to what we have implemented would be to introduce memoizing thunks that cache their results; subsequent invocations would just return the cached result instead of reperforming the traversal.

3.8 Calling Other Traversals

Being able to call other traversals is very useful, as it enables us to split common abstractions into units that can be reused by several other traversals. The translation is extended in a straightforward way; each traversal generates an interface that visitors that are to be used with that traversal must implement. If a visitor is to be used with several traversals, the visitor will need to implement each of their interfaces.

Another use of multiple traversals is to encode state in the traversals; depending on which traversal is active, the computation is in a certain state. If the traversals are mutually recursive, they must be specified in a block, as they must be type checked as a group.

Figure 8 shows the final version of the grammar. One traversal may call another traversal on a specified part by using the othertrav directive. This is a recursive call; once the called traversal terminates, the calling traversal resumes.

```
Traversals      ::= Traversal [and Traversal]*
Traversal       ::= traversal TraversalName
                      [Parameters] = TraversalEntry*
TraversalEntry  ::= ClassName [returning Type]
                      ⇒ Action*
Action          ::= [ Type ] VariableName
                      = ActionRhs ;
                  | ActionRhs ;
ActionRhs       ::= traverse PartOrVarName Arguments
                  | othertrav TraversalName
                      PartOrVarName Arguments
                  | visit VisitName Arguments
                  | super Arguments
                  | for-each PartOrVarName
                      MethodName Argument Arguments
                  | thunk { Action* }
                  | invoke VariableName
                  | VariableName
Parameters      ::= ( Parameter [, Parameter]* )
Parameter       ::= [ Type ] VariableName
Arguments       ::= ( Argument [, Argument]* )
Argument        ::= VariableName
```

Figure 8: The complete grammar

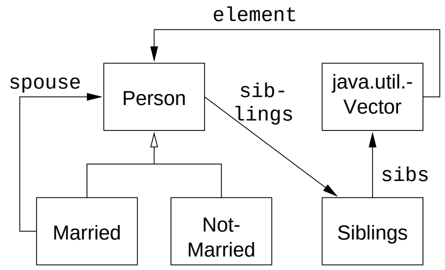


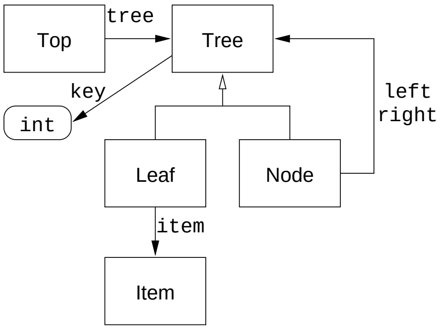
Figure 9: The Inlaws Example

4 Inlaws

The inlaws problem [Wer96] illustrates a weakness in the visitor pattern. When an object plays different roles depending on the context in which it appears, giving role-specific behavior becomes tedious. One workaround is to keep a state variable in the visitor that is updated during the traversal to reflect the behavior the visitor should have. This approach forces the behavior of the visitor to be aware of navigational details, which is contrary to the philosophy of the visitor pattern.

Figure 9 shows a class diagram for the inlaws problem. A Person can be Married or NotMarrried, and has zero or more siblings. An inlaw is the spouse of a sibling or a sibling of a spouse.

We start out with a person, and now wish to apply some operation (an inlaw visit, for example) to all the inlaws of that person. The visitor pattern has difficulty expressing this problem because the class Person plays the role of self, sibling, spouse, and inlaw. The behavior of the visitor depends on the role, not the class.




```
traversal binsearch(int goal) =
  Top  => traverse tree(goal);
  Node => gol = thunk {
                  traverse left(goal);
                };
          gor = thunk {
                  traverse right(goal);
                };
          visit choose(goal, gol, gor);
  Leaf => getitem = thunk {
                      traverse item(goal);
                    }
          visit atleaf(goal, getitem);
  Item => host;
```

```
class SearchVisitor implements binsearch_Visitor {
  Item choose(Node n, int goal, Thunk_Item left, Thunk_Item right) {
    if (goal > n.key) return right.invoke();
    else return left.invoke();
  }
  Item atleaf(Leaf l, int goal, Thunk_Item get) {
    if (goal == n.key) return get.invoke();
    else return null;
  }
}
```

Figure 7: Code to search a binary tree

Borrowing from solutions to similar problems in functional programming, we solve this problem by encoding the state of the visitor in the traversal name. Instead of having one traversal, we have four. The first traversal starts from the person whose inlaws we seek:

```
traversal myself() =
  Person     => othertrav mysibling siblings();
  Married    => sibs   = super();
                spouse = othertrav myspouse spouse();
                visit combine(spouse, sibs);
  NotMarried => super();
```

Anybody's siblings may lead to inlaws, so both Married and NotMarried delegate to the traversal entry for their superclass Person. A Married person also has inlaws via the spouse. We traverse to those using the `myspouse` traversal. Finally, the results from a Married person's spouse and siblings are combined using the `combine` visit on the visitor.

```
and traversal mysibling() =
  Siblings   => val = visit initval();
                for-each sibs reduce val();
  Married    => othertrav myinlaw spouse();
  NotMarried => visit initval();
```

A list of siblings is dealt with by using `for-each`. The visitor is assumed to have a visit `initval` that returns a suitable initial value for the reduction. The reduction is performed using the method `reduce` on the visitor. If a sibling is Married then we have found an inlaw: the spouse. If the sibling is NotMarried, then `initval` is used to return a suitable result.

```
and traversal myspouse() =
  Person     => othertrav myinlaw siblings();
```

In a similar vein, any siblings of a spouse are also inlaws.

```
and traversal myinlaw() =
  Siblings              => val = visit initval();
                           for-each sibs reduce val();
  Person returning int => visit inlaw();
```

When we have found an inlaw, we invoke the `inlaw` visit on the visitor. We specify a return type for the `inlaw` visit, so as to give the type checker a type to propagate. The type checker will determine that all the entries of the traversals `myself`, `mysibling`, `myspouse`, and `myinlaw` return the same type. As Java doesn't allow polymorphic method signatures, we have to specify at least one of them. Above, we chose to annotate the Person entry as `returning int`.

Like the sibling, a list of inlaws is reduced to one value by using `for-each` and `initval`. Each inlaw is `visit`ed, and the results are `reduce`d.

We can now use these traversals to count the number of inlaws of a person.

```
class InlawsCounter
  implements myself_Visitor, mysibling_Visitor,
             myspouse_Visitor, myinlaw_Visitor {

  int initval() { return 0; }
  int combine(Married h, int a, int b) {
    return reduce(a,b);
  }
  int reduce(int a, int b) { return a+b; }
  int inlaw(Person host) { return 1; }
}
class Person {
  ...
  int countinlaws() {
    return myself.traverse(this,new InlawsCounter());
  }
}
```

5 Translating to Java

In the following section we describe the details of the translation of traversals into Java. The translation is fairly straightforward; the largest task in the translation is performing type inference on the traversal specification. The process of type checking incidentally also verifies that the traversal is consistent with the class diagram.

Our code generation strategy assumes that the classes over which we are traversing are be written beforehand, and cannot be recompiled. This allows us to traverse over object structures from third party classes, so long as we are able to construct a class diagram for the subset of the object structure we wish to traverse. However, developing the traversal and the classes traversed in parallel is equally well supported, since the traversal specifications are translated into Java classes, which can be compiled separately or together with the user's classes.

5.1 Class Diagram

In order to infer the types for the traversal, we must know the structure of the program in which it is to be used. The class structure is needed for type-checking and to insert the proper casts for elements extracted from collections.

The class diagram describes the type structure of the program. We have used several graphical representations of class diagrams already (for example figure 9). The class diagram encodes three different types of edges between classes.

- **Part edges** relationships are described by the parts (or variables) of a class. These edges can be traversed explicitly through the `traverse` directive. All part edges are named.
- **Inheritance edges** When an object is traversed, the most precise TraversalEntry for that class is executed. An entry can choose to extend, rather than override, the behavior of its superclass by invoking the `super` directive.
- **Collection edges** are the implicit edges inside collection classes that point to each element. These edges can be traversed by a `for-each` directive. All collection edges are named.

It would be nice if an inheritance edge $c \supseteq c'$ between classes c and c' allowed the return type of the traversal of c' to be a subtype of c. Unfortunately, Java forbids overriding inherited methods with methods having different return types, so the constraint is strengthened to an equality.

5.2 Type Checking and Consistency

In the input grammar, all types are optional. We must know the type of all variables and the return types of all traversals in order to generate code for them. The result of type checking is to determine these types. Additionally, we need to generate an interface that all visitors to be used with the traversal must implement, which also requires detailed information about the types of the system.

While the class diagram and traversal specification do specify many of the types involved, they do not specify all. Typically, for many of the classes in the traversal, a return type will not be specified, and `for-each` needs to infer the type returned by traversing an object from the collection.

Since there are no function types or polymorphism, we can infer types by simple constraint-solving [Wan87]. Since all constraints are equivalence relations, it is possible to sort all type variables into equivalence classes in $O(n\alpha(n))$ time, where α is the inverse Ackerman function and realistically never returns a value greater than 4 [CLR90]. The rules for generating the constraints are given in figure 10. We take the opportunity to both check that the traversal is consistent with the class diagram and to generate the interface for visitors to be used with the traversal.

A traversal is considered consistent with the class diagram if we can verify that all parts that the traversal mentions exist where the traversal expects to find them. Also, the part used for `for-each` must be a collection class. Both of these conditions are verified as a side effect of generating the constraints. The constraints are of the form $t_1 = t_2$, where t_1 and t_2 are type variables or types

Once we have generated the constraints we solve them by sorting type variables into equivalence classes. If we find that this is not possible, we have a typing error. If we find that some type variables are not equivalent to any Java type, then the traversal is polymorphic in those type variables. Since Java does not yet have parametric polymorphism, we currently regard this as a type error. In the inlaws example in the previous section, we had to specify the return type of the `inlaws` visit to avoid the traversal becoming polymorphic.

Once we have determined the values for all type variables, we are ready to generate Java code.

5.3 Generated Java

Figures 11 and 12 show the details of how Java code is generated. The strategy is to generate a class for each traversal. For each class in the application a method with the name **entry**_*classname* is generated. If a class has a traversal entry, that is used to generate the body of the method. Otherwise, a default entry that delegates to the entry of the superclass is used. The method for java.lang.Object is empty by default. Additionally, methods to find the most precise traversal entry for an object (effectively simulating dynamic dispatch) are generated. These dispatch methods have the name **traverse**, and are overloaded on the type of their first argument.

After type inference, we know unambiguously the signature for any visitor method. In order to satisfy the type safety requirements of Java, we output these signatures into a Java interface *traversalname*_**Visitor**, and require all Visitors used with the traversal to implement the interface. The interface allows the visitor and the traversal it is to be used with to be compiled separately.

By generating the traversal code into a separate class, we are able to traverse any class so long as we are able to produce a class diagram for it. This allows us to traverse classes we are not able to recompile, but we need to simulate dynamic dispatch on traversal entries by searching for and invoking the most specific traversal entry for an object.

Furthermore, by putting the traversal code into one class, it becomes possible to use traditional techniques such as subclassing to extend and evolve existing traversals. This is discussed in section 6.

6 Future Work and Conclusion

One of the uses of the visitor pattern is to extend the functionality of existing classes [FFK98]. The typical visitor pattern takes the traversal of the class graph as given; the visitor always traverses the entire object graph. Under this

$$\mathcal{T}\left[\!\!\left[\begin{array}{ll}\textbf{traversal} & \textit{name}\ (\textit{params})\ = \\ & t_1 \\ & \ldots \\ & t_n\end{array}\right]\!\!\right] = \begin{array}{l}\mathcal{E}[\![t_1]\!] \\ \wedge\ \ldots \\ \wedge\ \mathcal{E}[\![t_n]\!]\end{array}$$

$$\mathcal{E}\left[\!\!\left[\begin{array}{ll}\textit{classname} & \textbf{returning}\ \textit{type} \Rightarrow \\ & a_1 \\ & \ldots \\ & a_{n-1} \\ & a_n\end{array}\right]\!\!\right] = \begin{array}{l}\text{Let } \mathsf{v}_1 \ldots \mathsf{v}_n \text{ be fresh type variables in} \\ \mathcal{A}[\![a_1]\!](\mathsf{v}_1) \\ \wedge\ \ldots \\ \wedge\ \mathcal{A}[\![a_{n-1}]\!](\mathsf{v}_{n-1}) \\ \wedge\ \mathcal{A}[\![a_n]\!](\mathsf{v}_n) \\ \wedge \text{ return type of traversal of } \textit{classname} = \mathsf{v}_n\end{array}$$

$$\begin{array}{lll}
\mathcal{A} & [\![\ \textit{varname} = \textit{ar}\ ]\!](\mathsf{v}) & = \mathcal{AR}[\![\textit{ar}]\!](\mathsf{v}) \\
\mathcal{A} & [\![\ \textit{type varname} = \textit{ar}\ ]\!](\mathsf{v}) & = \mathsf{v} = \textit{type}\ \wedge\ \mathcal{AR}[\![\textit{ar}]\!](\mathsf{v}) \\
\mathcal{A} & [\![\ \textit{ar}\ ]\!](\mathsf{v}) & = \mathcal{AR}[\![\textit{ar}]\!](\mathsf{v}) \\
\mathcal{AR} & [\![\ \textbf{traverse}\ \textit{part}\ (\textit{args})\ ]\!](\mathsf{v}) & = \mathsf{v} = \text{type of the traverse of } \textit{part} \\
 & & \quad \wedge \text{ type of } \textit{travparam}_i = \text{type of } \textit{args}_i \\
\mathcal{AR} & [\![\ \textbf{othertrav}\ \textit{trav part}\ (\textit{args})\ ]\!](\mathsf{v}) & = \mathsf{v} = \text{type of } \textit{trav} \text{ traversing } \textit{part} \\
 & & \quad \wedge \text{ type of } \textit{travparam}_i = \text{type of } \textit{args}_i \\
\mathcal{AR} & [\![\ \textbf{visit}\ \textit{meth}\ (\textit{args})\ ]\!](\mathsf{v}) & = \mathsf{v} = \text{return type of } \textit{meth} \\
 & & \quad \wedge \text{ type of } \textit{methparam}_0 = \text{type of } \textbf{host} \\
 & & \quad \wedge \text{ type of } \textit{methparam}_{i+1} = \text{type of } \textit{args}_i \\
\mathcal{AR} & [\![\ \textbf{method}\ \textit{meth}\ (\textit{args})\ ]\!](\mathsf{v}) & = \mathsf{v} = \text{return type of } \textit{meth} \\
 & & \quad \wedge \text{ type of } \textit{methparam}_i = \text{type of } \textit{args}_i \\
\mathcal{AR} & [\![\ \textit{varname}\ ]\!](\mathsf{v}) & = \mathsf{v} = \text{type of } \textit{varname} \\
\mathcal{AR} & [\![\ \textbf{for-each}\ \textit{part meth initvar}(\textit{args})\ ]\!](\mathsf{v}) & = \text{Let } \mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d} \text{ be fresh type variables in} \\
 & & \quad \mathsf{a} = \mathsf{v} \\
 & & \quad \wedge \text{ type of } \textit{part} = \text{collection of } \mathsf{d} \\
 & & \quad \wedge\ \mathsf{a} = \text{return type of } \textit{meth} \\
 & & \quad \wedge\ \mathsf{b} = \text{first argument to } \textit{meth} \\
 & & \quad \wedge\ \mathsf{c} = \text{second argument to } \textit{meth} \\
 & & \quad \wedge\ \mathsf{a} = \mathsf{c} \\
 & & \quad \wedge\ \mathsf{b} = \text{return type of traversal of } \mathsf{d} \\
 & & \quad \wedge\ \mathsf{a} = \textit{initvar} \\
 & & \quad \wedge \text{ type of } \textit{travparam}_i = \text{type of } \textit{args}_i
\end{array}$$

$$\mathcal{AR}\left[\!\!\left[\begin{array}{ll}\textbf{thunk}\{ & \\ & a_1 \\ & \ldots \\ & a_{n-1} \\ & a_n\}\end{array}\right]\!\!\right](\mathsf{v}) = \begin{array}{l}\text{Let } \mathsf{v}_1 \ldots \mathsf{v}_n \text{ be fresh type variables in} \\ \mathcal{A}[\![a_1]\!](\mathsf{v}_1) \\ \wedge\ \ldots \\ \wedge\ \mathcal{A}[\![a_{n-1}]\!](\mathsf{v}_{n-1}) \\ \wedge\ \mathcal{A}[\![a_n]\!](\mathsf{v}_n) \\ \wedge\ \mathsf{v} = \text{thunk returning } \mathsf{v}_n\end{array}$$

$$\mathcal{AR}\ [\![\ \textbf{invoke}\ \textit{thunk}\ ]\!](\mathsf{v}) = \text{type of } \textit{thunk} = \text{thunk returning } \mathsf{v}$$

Figure 10: Type Constraint Generation Rules

For each *type* in the classgraph that is returned from a thunk

```
abstract class Thunk_type{
        abstract type invoke();
}
```

$$\mathcal{T}\left[\!\!\left[\begin{array}{ll}\textbf{traversal} & \mathit{name}(\mathit{params}) \\ & = \; t_1 \\ & \quad \ldots \\ & \quad t_n\end{array}\right]\!\!\right] =$$

```
class name_Traversal {
    name_Visitor thevisitor;
    name(name_Visitor visitor) {
        this.thevisitor = visitor;
    }
    E[[t_1]](name,params)
    ...
    E[[t_n]](name,params)

    For each class in the classgraph
    Let subclass_1 ... subclass_n be the immediate subclasses of class in
    type  traverse(class host, paramlist) {
        if (class instanceof subclass_1)
           return this.traverse((subclass_1) host,paramlist);
        ...
        else if (class instanceof subclass_n)
           return this.traverse((subclass_n) host,paramlist);
        else return this.entry_class(host,paramlist);
    }
}
```

$$\mathcal{E}\left[\!\!\left[\begin{array}{ll}\mathit{classname} & \textbf{returning}\ \mathit{type} \\ & \Rightarrow \; a_1 \\ & \quad \ldots \\ & \quad a_{n-1}; \\ & \quad a_n;\end{array}\right]\!\!\right]_{(\mathit{name},\mathit{params})} =$$

```
static type traverse( classname host,
                      name_Visitor visitor,
                      params              ) {
      return new name_Traversal(visitor)
                 .traverse(host,params);
}
type entry_classname(classname host, params) {
    A[[a_1]](name, false)
    ...
    A[[a_{n-1}]](name, false)
    A[[a_n]](name, true)
}
```

Figure 11: Java translation for Traversals, part 1.

$\mathcal{A}[\![\ \textit{type varname} = \textit{ar}\ ]\!](\textit{name}, \textit{ret})$ = if *ret*, $\mathcal{AR}[\![\textit{ar}]\!](\textit{name}$, "**return** ")
else $\mathcal{AR}[\![\textit{ar}]\!](\textit{name}, \textit{ret}$, "*type varname* =")

$\mathcal{A}[\![\ \textit{ar}\ ]\!](\textit{name}, \textit{ret})$ = if *ret*, $\mathcal{AR}[\![\textit{ar}]\!](\textit{name}$, "**return** ")
else $\mathcal{AR}[\![\textit{ar}]\!](\textit{name}, \textit{ret}$, "")

$\mathcal{AR}[\![$ **traverse** *part* (*args*) $]\!]$ (*name*, *binder*) = Let *getstr* = *part* , if *part* is a traversal variable
host.*part***()** , else (*part* is a host variable) in
binder **this.traverse(***getstr*,*args***);**

$\mathcal{AR}[\![$ **othertrav** *travname part* (*args*) $]\!]$ (*name*,*binder*) = Let *getstr* = *part* , if *part* is a traversal variable
host.*part***()** , else (*part* is a host variable) in
binder *travname***.traverse(**
getstr,
(*travname*_Visitor) thevisitor,
args **);**

$\mathcal{AR}[\![$ **visit** *meth* **(***args***)** $]\!]$(*name*, *binder*) = *binder* **this.thevisitor.***meth***(host,***args***);**

$\mathcal{AR}[\![$ **super (***args***)** $]\!]$(*name*, *binder*) = Let *superclass* be the closest superclass of *class* in
binder **entry_***superclass***(host,***args***);**

$\mathcal{AR}[\![$ *varname* $]\!]$(*name*, *binder*) = *binder* *varname*

$\mathcal{AR}[\![$ **for-each** *part meth initvar***(***args***)** $]\!]$ (*name*, *binder*) = Let *parttype*, *elemtype*, and *travresulttype* be inferred types
getstr = *part* , if *part* is a traversal variable
host.*part***()** , else (*part* is a host variable) in
java.util.Enumeration enum = *getstr***.elements();**
inittype **acc =** *initvar***;**
while (enum.hasMoreElements()) {
elemtype **elem =** (*elemtype*) **enum.nextElement();**
travresulttype **t = this.traverse(elem,***args***);**
acc = this.thevisitor.*meth***(t, acc);**
}
binder **acc;**

$\mathcal{AR}\left[\!\!\left[\begin{array}{l}\textbf{thunk\{}\\ \quad a_1\\ \quad \ldots\\ \quad a_{n-1};\\ \quad a_n;\}\end{array}\right]\!\!\right](\textit{name},\textit{binder})$ = *binder* **new Thunk_** *type* **() {**
type **invoke() {**
$\mathcal{A}[\![a_1]\!](\textit{name}, \texttt{false})$
...
$\mathcal{A}[\![a_{n-1}]\!](\textit{name}, \texttt{false})$
$\mathcal{A}[\![a_n]\!](\textit{name}, \texttt{true})$
};
}

$\mathcal{AR}[\![$ **invoke** *thunkname* $]\!]$(*name*, *binder*) = *binder* *thunkname***.invoke();**

Figure 12: Java translation for Traversals, part 2

system, we are able to extend behavior simply via inheritance on the visitor.

In our system, since we have separate traversal specifications, it would make sense to be able to extend traversals in addition to visitors. A traversal entry for a class would override the traversal entry for its super class. The overridden entry can still be invoked via Java's `super` mechanism. However, we have already used that keyword to invoke the entry for the current object's superclass. By allowing traversal specifications to subclass each other, we introduce a second dimension of overriding, thus raising the question of which overridden behavior we wish to invoke with a `super` call. The combination two dimensions of behavioral specialization is an issue for further study.

We have a prototype system implemented, and have sketched out in detail how to combine traversal specifications with the Dem/Java programming environment. Dem/Java is a visitor-pattern-based programming environment that has been extensively used in a number of medium scale programming projects. It incorporates succinct traversal strategies that make expressing recursive algorithms difficult. The two systems complement each other – Dem/Java simplifies traversals of large class graphs, while traversal strategies allow elegant specification of recursive computations over smaller subgraphs. However, due to time constraints, the combination has reached only a very preliminary stage, with many features missing.

References

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT press, 1990.

[FFK98] Matthias Felleisen, Daniel Friedman, and Shriram Krishnamurthi. Synthesizing object-oriented and functional design to promote reuse. In *European Conference on Object-Oriented Programming*, 1998.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Lie92] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th IFIP World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[Lie96] Karl Lieberherr. *Adaptive Object-Oriented Software, The Demeter Method.* International Thomson Publishing, 1996.

[LO97] Karl Lieberherr and Doug Orleans. Preventative program maintainance in demeter/java (research demonstration). In *Proceedings of the International Conference on Software Engineering*, pages 604–605, 1997. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[Wer96] Mike Werner. Personal Communication to the Demeter Seminar, 1996.