Abstraction of Communication in Traversal-Related Concerns

Bryan Chadwick, Ahmed Abdelmeged, Therapon Skotiniotis, and Karl Lieberherr

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

Abstract. In this paper we propose two new forms of Adaptive Programming. The first approach, AP-P, is an imperative approach that adds a programming construct called, interposition variables, that reduce boilerplate code in computations over recursive data structures. Our current implementation of AP-P is called DemeterP and is provided as a standard Java annotation processor that generates AspectJ code. The second approach, AP-F, is a functional approach that parameterizes a generic traversal with three sets of functions. These functions can be combined to modify traversal behavior to produce various functional transformations and folds. Our current implementation of AP-F is called DemeterF and is provided as a Java library that relies on reflection. We describe the ideas behind the two approaches, discuss their implementations, and use them to solve some typical programming language related problems.

1 Introduction

The corner stone of Object Oriented Programming (OOP) is the notion of object communication via message passing. Collaborations between objects are the building blocks used by programmers to provide desired program functionality. One of the most common collaboration patterns used is *recursive traversal* [1–3]— given an object, recursively traverse its fields, performing necessary computation; recursion terminates when an object without fields is reached. The visitor design pattern [4] gives an abstraction that can be customized to implement operations over a data structure, but programmers are still left with the following questions:

– how should the data structure be traversed,
– which objects are responsible for the traversal implementation, and
– where and how should information be made available during traversal.

Adaptive Programming (AP) emerged as an extension to OOP with the goal of abstracting and separating traversal and computation from object structures. In AP, traversal code is automatically generated from a description of the data structure and a traversal *strategy*— a specification of where and how deep the traversal should go. Computation is defined separately in a specialized *visitor*, with methods executed *before* and *after* reaching an object of a specific type. Values are communicated between visitor methods by mutating visitor instance variables.

In this paper we introduce two refinements of AP: AP-P and AP-F. AP-P uses annotations to define *interposition variables* [5, 6] within visitors. These variables are only available during traversal and can be used to communicate information between different executions of visitor methods. AP-F is a functional formulation of AP that decomposes traversal computation into three sets of functions: *transformers*, *builders*, and *augmentors*. AP-F employs a method dispatch mechanism during traversal, allowing communication to occur through function arguments and return values. Both refinements enhance communication during traversal, maintaining traversal separation by decreasing dependencies between traversal and computation.

As an illustrative example consider the class hierarchy in Figure 1 that describes `Items`; an item is either a single `Element` with a weight, or a `Container` with a maximum capacity and a list of `items`. Given an item, we want to check recursively for container *violations*, *i.e.*, container objects whose total weight is greater than their capacity.

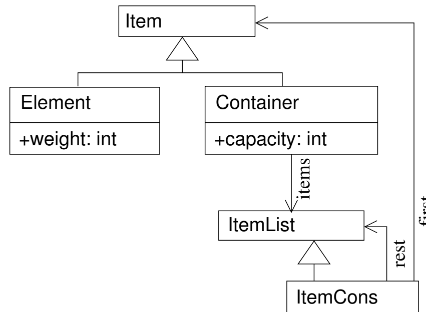

Fig. 1. Containers UML Class Diagram

Figure 2 gives a functional, object oriented solution to the container problem in Java. We define a helper class, `Pair`, to hold the number of violations and the total weight of the current item(s). We introduce the method `check()` into each class, which responsible for calculating the item's weight and nested violations.

In the pure object oriented solution writing traversal code is tedious, but straight forward: for each object type, recurse through its immediate fields until we reach an `Element`, which contains no aggregate fields[1]. The traversal also hard-codes the names and layout within the object structures. As a result any modification of names or structure requires a number of disjoint changes to the implementation; these changes are to "boilerplate code" that is uninteresting to develop and maintain.

Figure 3 shows an AP Visitor solution to the container problem in DJ [7]. The creation of a `ClassGraph` instance extracts a representation of the related classes using reflection. The method `traverse()` traverses its first argument, using the second as a strategy and its third as the visitor to be executed during traversal. In AP, as with the

[1] We do not usually attempt to traverse base types such as `Integer`, though they may actually have fields

```java
// Pair of Ints, <weight, violations>
class Pair{
  int w, v;
  Pair(int ww, int vv){ ... }
  static Pair make(int ww, int vv){
    return new Pair(ww,vv);
  }
  Pair add(int ww, int vv){
    return Pair.make(w+ww, v+vv);
  }
}

// In class Item
  abstract Pair check();

// In class Element
  Pair check(){
    return Pair.make(weight,0);
  }
```

```java
// In class Container
  Pair check(){
    Pair p = items.check();
    return p.add(0, (p.w > capacity)?1:0);
  }

// In class ItemList
  Pair check(){
    return Pair.make(0, 0);
  }

// In class ItemCons
  Pair check(){
    Pair f = first.check(),
         r = rest.check();
    return f.add(r.w, r.v);
  }
```

Fig. 2. Java implementation of the Container problem.

```java
class CheckAP extends Visitor{
    Stack<Integer> weightStk = new Stack<Integer>();
    int weight = 0, violations = 0;

    void after(Element e){ weight += e.weight; }
    void before(Container c){
       weightStk.push(weight);
       weight = 0;
    }
    void after(Container c){
       if(weight > c.capacity) violations++;
       weight += weightStk.pop();
    }
    static int check(Item i){
       CheckAP v = new CheckAP();
       new ClassGraph(true,false).traverse(i, "from Item to *", v);
       return v.violations;
    }
}
```

Fig. 3. AP Visitor Implementation of the Container problem.

visitor pattern, communication is provided by local mutation to visitor instance variables. We use a Stack to mimic the recursive calls of the Java solution, pushing and popping the previous container's running weight when traversing a nested container[2].

AP-P was designed with this type of calculation in mind, encapsulating operations and nested state within visitors. Figure 4 shows the container solution written in our AP-P implementation, DemeterP. The Java annotation associates an Integer variable, named weight, with each Container instance. Each weight is initialized to 0 and available only during traversal, within the the scope of the visitor CheckP. Using the name weight within the visitor methods accesses the current value of the enclosing container's weight. The special syntax ($ipvs.weight.val) references the *last visited* container's weight, *i.e.*, the weight associated with the argument to *after*. Upon traversal entry of a container, the corresponding interposition variable is initialized and becomes active; after exiting each container, the enclosing container's weight is updated and the variable is discarded (similar to the push and pop operations in our AP solution).

```
class CheckP extends InterpositionVisitor{
    @Interposition(classes={Container.class}, initializer="0")
       Integer weight;
    int violations = 0;

    void after(Element e){ weight += e.weight; }
    void after(Container c){
       int currWeight = $ipvs.weight.val;
       violations += ((c.capacity < currWeight)?1:0);
       weight += currWeight;
    }
    static int check(Item i){
       CheckP v = new CheckP();
       new ClassGraph(true,false).traverse(i, "from Item to *", v);
       return v.violations;
    }
}
```

Fig. 4. DemeterP Implementation of the Container problem

AP-F extends AP using a functional traversal in the style of our Java container solution. Communication between functions is provided through arguments and return values, with a method dispatch base on all argument types. Figure 5 shows the container solution written in our AP-F implementation, called DemeterF. We write combine() methods corresponding to each of the check() methods in the Java solution— the first argument to the methods is the object being traversed. The dispatch mechanism selects the most specific applicable method based on the type signature. The method with argument type Element executes when we reach an Element, while the method with argument types ItemCons, Pair, and Pair executes when we reach an object of type ItemCons, after processing its fields: first and rest. The traversal

[2] The stack is needed because we are performing a non-tail-recursive computation. For example, a single variable is sufficient to count the total number of elements in a given container

of these fields yields objects of type `Pair`, which are passed as the second and third arguments. The separation of functions in this way allows for a more functional style traversal, leading to straight forward traversal parallelization and simpler implementations of transformations.

```
class CheckF extends IDb{
    Pair combine(Element e){ return Pair.make(e.weight, 0); }
    Pair combine(Container c, Integer cap, Pair p)
     { return p.add(0, (p.w > cap)?1:0); }
    Pair combine(ItemList il){ return Pair.make(0, 0); }
    Pair combine(ItemCons ic, Pair f, Pair r){ return f.add(r.w, r.v); }

    static Pair check(Item i)
     { return new Traversal(new CheckF()).traverse(i); }
}
```

Fig. 5. DemeterF Implementation of the Container problem

AP-P and AP-F allow for better abstractions for traversal related computations. AP-P encapsulates visitor state and operations on this state, provides context sensitive access to visitor state and boilerplate code for updating visitor state in the presence of recursive object structures. AP-F provides a new decomposition of computation during traversal into transformer, builder, and augmentor functions along with an extend method dispatch mechanism. Communication and state passing occurs through arguments and return values leading to functional style solutions.

Both approaches enhance aspects of communication found in AP and remove the need for boilerplate code; computation along recursive traversals becomes succinct, easier to maintain, and reusable. To support our claims we introduce our running example in the next section and follow our discussion with a more detailed explanation of AP-P in section 3 and AP-F in section 4. We discuss related work in section 5 and conclude in section 6.

2 Running Examples

For the remainder of the paper we introduce a small language and provide solutions for evaluation, type checking and compilation. All three phases are first implemented using AP-P and then AP-F explaining our approach to program design under both AP extensions. We introduce a small language of *integers* and *strings* together with *casts* and polymorphic *addition*. We define classes of our abstract syntax tree for this language using a Demeter style Class Dictionary (CD) that mixes concrete syntax (terminals) with abstract syntax (data type definitions).

Figure 6 shows our CD for the simple language and data structures. In this notation, ':' defines an abstract class (a *sum* or *union* type), and '=' defines a concrete class (or *product* type). Field names are given in <·> followed by their *type*. Concrete syntax is given as string literals in-place and each definition is terminated by a period ('.').

```
Exp: IntExp | StrExp | StrCast | PlusExp.

IntExp = <val> Integer.
StrExp = <val> String.
StrCast = "(string)" <exp> Exp.

PlusExp = "(" <lhs> Exp <op> Plus <rhs> Exp ")".
Plus = "+".
```

Fig. 6. Mixed concrete and abstract syntax for Examples

From this description it is easy to see how we would generate both Java class definitions and a general parser for the data types[3]. In this simple language, the expression `("5"+(string)4)` would be represented by the following Java expression:

```
new PlusExp(new StrExp("5"), new Plus(),
            new StrCast(new IntExp(4)));
```

2.1 Example: Evaluation

For our first example with this language we will produce evaluators based on a simple, intuitive semantics. We use the notation $\lceil n \rceil$ and $\lceil s \rceil$ to denote the representation of the integer n and the string s respectively. Values in our language consist of c_n, constants that represent integers, and c_s, constants that represent strings. For our implementations both representations correspond to the Java classes `Integer` and `String` respectively. The language has an overloaded operator $+$, which, given two string values returns a string value representing their concatenation, and given two integer values returns an integer value representing their sum. If given a string and an integer, $+$ throws an error. We denote concatenation of two string representations with $\cdot$ and integer addition with $\boldsymbol{+}$.

$$\frac{}{\lceil s \rceil \rightarrow c_s} \qquad \frac{}{\lceil n \rceil \rightarrow c_n} \qquad \frac{}{\texttt{(string)}\,c_s \rightarrow c_s}$$

$$\frac{c_s = \lceil \text{“}n\text{”} \rceil}{\texttt{(string)}\,c_n \rightarrow c_s} \qquad \frac{\lceil n \rceil \boldsymbol{+} \lceil m \rceil = \lceil k \rceil}{(c_n + c_m) \rightarrow c_k} \qquad \frac{\lceil s \rceil \cdot \lceil t \rceil = \lceil r \rceil}{(c_s + c_t) \rightarrow c_r}$$

2.2 Example: Type Checking

Our second example involves type checking expressions based on the obvious typing judgments described below. There is no type environment; the types of expressions are based solely on the types of their sub-expressions. Here $\triangleright\, e : t$ means we assign the type t to the expression e. `Str`s and `Int`s are of type `StrType` and `IntType` respectively. Any well typed expression can be cast to a string, assigning it the result type `StrType`, while `PlusExp` is assigned the type of its sub-expressions.

[3] We use DemeterJ[8] to generate Java classes and input for the javacc[9] parser generator

$$\frac{}{\triangleright \lceil s \rceil : \texttt{StrType}} \qquad \frac{}{\triangleright \lceil n \rceil : \texttt{IntType}}$$

$$\frac{\triangleright\, e : t}{\triangleright\, \texttt{(string)}\,e : \texttt{StrType}} \qquad \frac{\triangleright\, e_1 : t \quad \triangleright\, e_2 : t}{\triangleright\, (e_1 + e_2) : t}$$

To represent types in our type checker implementations we introduce the following Java classes with static fields. The fields represent singleton types; they will only be used as tags during type checking.

```
class Type{
    static Type Int = new IntType(),
                Str = new StrType(),
                Plus = new PlusType(),
                Err = new ErrType();
}
class  IntType extends Type{}
class  StrType extends Type{}
class PlusType extends Type{}
class  ErrType extends Type{}
```

2.3 Example: Compilation

As a final example, we will reuse our type checking solutions to create a compiler from our example language to a simple assembly language for an abstract stack machine. The semantics of our assembly language are described below. In this notation each instruction transforms the *stack*; an instruction is a function of type $(stack \rightarrow stack)$. The stack is a list of values separated by commas with the top of the stack to the left. `pushs` and `pushi` instructions are straight forward, pushing values onto the stack. `casts` replaces an integer value on the top of the stack with the corresponding string value; a cast of a string value has no effect.

$$\begin{array}{rrcll}
\texttt{pushs}\ s: & stk & \rightarrow & c_s, stk & \\
\texttt{pushi}\ n: & stk & \rightarrow & c_n, stk & \\
\texttt{casts}: & c_n, stk & \rightarrow & c_s, stk & c_s = \lceil "n" \rceil \\
 & c_s, stk & \rightarrow & c_s, stk & \\
\texttt{add}: & c_m, c_n, stk & \rightarrow & c_{n+m}, stk & \\
\texttt{concat}: & c_t, c_s, stk & \rightarrow & c_{s \cdot t}, stk &
\end{array}$$

The two forms of `Plus` are given by the `add` and `concat` instructions, adding or concatenating the top two values on the stack. Note that the arguments are placed on the stack in order, left to right, so the top of the stack is actually the right most argument. We introduce a functional list class, `OpList`, with an overloaded method `append()` that accepts an `Opcode`— the abstract base class of the above definitions— or another `OpList`, returning the updated `OpList`. The usage is made obvious in the following sections.

3 AP-P

AP-P is concerned with achieving *structure-shy* communication among different points during traversal, limiting the amount of boilerplate code that needs to be written. In a functional style traversal, as in Figure 2, different functions communicate through argument passing and return values. Typically, only one return value is allowed and arguments are passed only to immediate function calls. This form of communication is not structure-shy because values are communicated through possibly unrelated/uninteresting portions of the data structure.

In an AP style traversal, a visit method accepts a single parameter, the current object being traversed, and returns *void*. Communication between visit methods is done through mutation of shared state— the fields of the visitor instance. The approach proposed by AP-P for structuring communication is to have fields in the visitor, called *interposition variables* [5, 6], which store information related to instances of specific types being visited. While traversing, different visit methods can communicate through the fields associated with a given instance.

In the case of non-tail recursive traversals, the state of each piece of traversal advice needs to be kept during the execution of nested visit methods calls. The state can be kept as a mapping from the host object to another object encapsulating the state of traversal advice. In case of recursive data structures (*e.g.*, a *composite* [4] as in Figure 1), the depth of recursive calls is not known in advance A stack can also be used as we did in 3. Maintenance of the mapping structure results in undesirable boilerplate code.

3.1 Interposition Variables

An interposition variable is a *context sensitive* variable. Its *context* is defined by the state of the traversal. As a part of its definition, an interposition variable gets a set of *host types*. During traversal, we signal a context entry before visiting a host type's children nodes – its fields. We signal a context exit after visiting a host type's children.

Statically, an interposition variable of type t_i over a host type t_h introduces a new field in t_h of type t_i. At runtime, during traversal, for each context defined by each instance of t_h we associate a new instance of t_i. We refer to this runtime association between interposition variables instances to host type instances as *incarnation* of an interposition variable. When the visitor is visiting an instance of one of a host types, the interposition variable still refers to the innermost enclosing *incarnation*. On the other hand, the object which is being visited also has a corresponding instance of the interposition variable. This instance can also be referred to using a special Java `enum` whose constants denote all interposition variables defined in the current visitor. Before the visitor hits the first instance of any host class, there are no enclosing *incarnations*. Therefore, the interposition variable refers to an empty incarnation, the *root incarnation*. This *root incarnation* holds the result of the entire computation.

Our implementation of interposition variables, DemeterP, saves the developers the mental burden of identifying those control flow points where the interposition variable context changes. It also saves the developer the effort of writing the boilerplate code required to adapt interposition variables to their context.

3.2 Polymorphic Interposition Variables

The traditional interposition variables described above are *monomorphic*; regardless of the type of the host class they are attached to, all instantiations have the same type. A monomorphic interposition variable is suitable for implementing *type unifying* traversals [10]. A polymorphic interposition variable can have different types based on the type of the host class it is attached to. Therefore, polymorphic interposition variables are suitable for implementing *type preserving* traversals [10], *i.e.*, deep cloning an object structure.

Using Java's subtype mechanism, polymorphic interposition variables can be declared to be of type `Object`. The interposition variable's initialization expression is executed in an environment where the variable `host` is bound to the current object about to be traversed. Initialization expressions can check for the type of the host using `host.getClass()` and initialize the interposition variable to an appropriate type. Pieces of traversal advice need to down cast the polymorphic interposition variable to the appropriate type before using it.

3.3 Syntax of Interposition Variables

An interposition annotation consists of three optional parts: an array of classes, an initializer, and a flag to control the initialization of the root incarnation. Figure 7 shows the abstract syntax of interposition variables. An empty array of classes means *all* classes in the traversal. A `true` initialization flag means that the initializer is also used for the *root incarnation*. This initializes the visitor field.

Each part has a default value. The default value of the class list is an empty array. The default value of the flag is true. Figure 8 shows the Java annotation type. An initializer which uses the type of the predefined variable `host` results in a truly polymorphic interposition variable. On the other hand, an initializer which does not use the type of the predefined variable `host` results in a monomorphic interposition variable.

```
VisitorField: NormalVariable | InterpositionVariable.
NormalVariable = <type> Type <name> Ident [<initializer> JavaExpr].
InterpositionVariable = [IPClasses] [IPInitializer] [IPInitVisitor].

IPClasses : ProperClassList | EmptyClassList.
ProperClassList =  List(ClassName).
EmptyClassList = .
IPInitializer = JavaExpr.
IPInitVisitor = Boolean.

ClassName = Ident.
List(x) ~ {x}.
```

Fig. 7. Abstract Syntax for Interposition Variables

```
@Documented
@Retention(value=RetentionPolicy.SOURCE)
@Target(value=ElementType.FIELD)
@interface Interposition {
        Class<?>[] classes() default {};
        String initializer() default "";
        boolean initVisitorVar() default true;
}
```

Fig. 8. Interposition Annotation

3.4 DemeterP Implementation

DemeterP [11] is implemented as a standard Java annotation processor [12]. Our annotation processor generates aspects in the AspectJ [13] Language. The generated Aspect is responsible for:

- Identifying the points at which the context of an interposition variable changes.
- Introducing interposition variables at their *host types*.
- Initializing the *root incarnation* of the interposition variable.
- Adapting interposition variables to their context.

In AP, visitors have two types of visit methods *before* and *after*. A *before* method is executed at a host object before any of its children are visited. An *after* method is executed at a host object after all of its children are visited.

In our implementation, we trap the execution of these methods using *pointcut*s generated from the templates shown in Figure 9. Interposition variables with incarnations at every visited object have a slightly different template, Figure 9.

Pointcuts alone are not enough; the processed visitor contains only a `before(.. host)` without a corresponding `after(.. host)` or the reverse case of an *after* without a corresponding *before*. In these cases, our generator introduces appropriate methods. That is why we need to have the `<AspectType>` inside the `within()` clause in our pointcuts, capturing execution of our generated visitor methods.

Interposition variables are introduced at their *host types* using AspectJ's *private introduction*. This makes the introduced fields inaccessible outside the aspect [13]. *Root incarnations* of interposition variables are initialized by advising visitor constructor(s).

For every visitor being processed, our processor generates a Java *enum* whose constants denote the interposition variables defined in the visitor. This *enum* is used to access the incarnations of interposition variables at the current object being traversed, in case that object is an instance of one of the *host types*.

Interposition variables are adapted to their context using three pieces of advice. The first is executed before entering to a host class. It sets the appropriate enum constant *enum* to a new incarnation of the interposition variable using the appropriate initializer if one exists. The second is executed after a *before* method terminates. It stores the interposition variable (in the visitor) at the host object and sets the interposition variable to the new incarnation from the *enum*. The third piece of advice is executed before an *after* method begins and restores incarnations to their original state – the state before executing our second advice.

```
ContextChangePointcut :=
   pointcut <method><HostTypeName><VisitorTypeName>
       (<VisitorType> v, <HostType> c): target(v)&& args(c)&&
       execution(public void before(<HostType>))&&
       within(<VisitorType>||<AspectType>);

EverywhereContextChangePointcut :=
   pointcut <method>Object<VisitorTypeName>(<VisitorType> v, Object c):
       target(v)&& args(c)&&
       execution(public void before(..))&&
       within(<VisitorType>||<AspectType>);

Enumeration := enum <VisitorTypeName>$IPVS{
               <InterpositionVariables(VisitorType)>;
               public Object val;
             }
```

where:

- *method* is either ”before” or ”after”.
- *HostType/VisitorType* is the fully qualified class name of the host/visitor.
- *HostTypeName/VisitorTypeName* is the HostType/VisitorType with every ”.” replaced with ”_”.
- *AspectType* is the name of the aspect.
- InterpositionVariables(VisitorType) is a function that takes a VisitorType and returns a comma separated list of all interposition variables defined in the give VisitorType.

Fig. 9. Code Generation Templates

3.5 Example Solutions

Figure 10 shows an interpreter for the example language written in DemeterP. An interposition variable is associated with every class that represents an operation, *i.e.*, `StrCast` and `PlusExp`. The interposition variable holds the results of evaluating the subterm(s) of the expression. After traversing a `StrCast` we add the newly cast value to the *innermost* incarnation of `vals`. We do the same for `PlusExp`, dispatching on the type of the first result, relying on dynamic casts for error checking. After traversal completes the *root incarnation* contains the evaluation result of the entire expression.

Figure 11 shows a type checker for the example language written in DemeterP. The structure of the type checker resembles that of the interpreter, but our interposition variable stores the *types* of any sub-expressions. These types are used to choose the correct result for compound expressions; being careful to propagate any type errors up through the interposition variables.

Figure 12 shows a compiler for the example language written in DemeterP. The compiler is implemented as an extension to the type checker. This way, the translation can use the types to choose the appropriate `Opcode` for `PlusExp`. The type checking code is reused and compilation is performed in a single traversal. The static `compile()` function runs the traversal, checks for a valid return type, and produces the resulting `OpList`. Not that within the `CompilerVisitor` we use a single `OpList`, not an interposition variable, to store the `Opcodes`. This is because the order of operations matches the defined traversal order in DemeterP; each `Opcode` can safely be

```
class EvalVisitor extends Visitor{
    @Interposition(classes={StrCast.class,PlusExp.class},
                   initializer="new Vector<Object>()")
       Vector<Object> vals;

    void after(Integer i){ vals.add(i); }
    void after(String s){ vals.add(s); }
    void after(StrCast e){
       Object val = $ipvs.vals.val.elementAt(0);
       vals.add(""+val);
    }
    void after(PlusExp e){
       Object val1 = $ipvs.vals.val.elementAt(0);
       Object val2 = $ipvs.vals.val.elementAt(1);
       vals.add((val1 instanceof Integer)?
                (Integer)val1+(Integer)val2:
                (String)val1+(String)val2);
    }
}
```

Fig. 10. Example Language Evaluation in DemeterP

```
class TypeCheckerVisitor extends Visitor{
    @Interposition(classes={StrCast.class,PlusExp.class},
                   initializer="new Vector<Type>()")
       Vector<Type> types;

    void after(IntExp e){ types.add(Type.Int); }
    void after(StrExp e){ types.add(StrType.Str); }
    void after(StrCast e){
       Type opT = $ipvs.types.val.elementAt(0);
       types.add((opT == Type.Err)?Type.Err:Type.Str);
    }
    void after(PlusExp e){ types.add(plusType(e)); }
    Type plusType(PlusExp e){
       Type op1T = $ipvs.types.val.elementAt(0);
       Type op2T = $ipvs.types.val.elementAt(1);
       return (op1T == op2T && op1T != Type.Err)?op1T:Type.Err;
    }
    Type getType(){ return types.elementAt(0); }
}
```

Fig. 11. Example Language Type Checker in DemeterP

added to the end of the list, in order. We also use an `append()` method to localize list mutation, since its implementation is immutable.

```
class CompileVisitor extends TypeCheckerVisitor {
    OpList trans = new OpList();
    void append(Opcode op){ trans = trans.append(op); }

    void after(StrExp s){
       super.after(s);
       append(new PushS(s.val));
    }
    void after(IntExp i){
       super.after(i);
       append(new PushI(i.val));
    }
    void after(PlusExp e){
       super.after(e);
       Type plusType = plusType(e);
       append((plusType==Type.Int)? new Add(): new Concat());
    }
    void after(StrCast e){
       super.after(e);
       append(new CastS());
    }

    static OpList compile(Exp e){
       CompileVisitor comp = new CompileVisitor();
       new ClassGraph(true,false).traverse(e, "from Exp to *", comp);
       if(comp.getType() == Type.Err)
          throw new TypeException(comp.getType());
       return comp.trans;
    }
}
```

Fig. 12. Example Language Compiler in DemeterP

4 Functional AP

AP-F was conceived to merge ideas prevalent in functional programming with those already found in AP. Recursive traversals in functional languages are written in an elegant way, but usually repeat common structure. Because many programming solutions involve traversals, and rewriting these becomes tedious, we can factor out the generic traversal code, leaving only the interesting parts of the program.

This has long been done through the use of general AP— separating visitors from traversals— but the complete separation of traversals makes mutation the only form of communication and computation. Mutation works fine when we want to compute a few values from a given data structure, because side-effects are restricted to a visitor's instance variables. For problems which require updates to the data structure being traversed (*e.g.*, rewrites and transformations), we are forced to mutate what could be globally shared state.

This may seem reasonable in a single threaded environment (though changes seen through references may cause problems), but when attempting to parallelize our solu-

tions, as is now often the case, we are required to add locks and synchronization which are difficult (if not impossible) to get right for non-trivial programs. With these ideas in mind, we designed AP-F to allow communication throughout the traversal using arguments and return values, rather than mutation[4].

4.1 AP-F Traversals

An AP-F program is roughly defined by three *sets* of functions, which we call *transformers*, *builders*, and *augmentors*. These sets of functions adapt the behavior of a pre-defined recursive traversal with the help of a multiple dispatch function, which chooses the best function from a given set[5]. The traversal function, $T_{f,\beta,\alpha}$, and related abstractions are described in Figure 13. The traversal is divided into two cases: *BuiltIn* types (*e.g.*, `int`, `boolean`), and user defined types; represented abstractly as a sequence of *fields*. The traversal accepts an extra argument, d_a in the figure, which is *updated* during traversal.

$$
\begin{array}{lll}
T_{f,\beta,\alpha}(D,\ d_a) \ \Rightarrow\ \textbf{if}\ D \equiv (d_0,\ldots,d_n)\ \textbf{then} & & \\
\quad \textbf{let}\ d'_a \leftarrow \delta(\alpha,\ (D,\ d_a)) & & \text{– traversal argument } \textit{update} \\
\quad\quad d'_i \leftarrow T_{f,\beta,\alpha}(d_i,\ d'_a) & & \text{– traverse } \textit{fields} \\
\quad\quad \hat{D} \leftarrow \delta(\beta,\ (D,\ d'_0,\ldots,d'_n,\ d_a)) & & \text{– } \textit{combine} \text{ Results} \\
\quad \textbf{in}\ \delta(f,\ (\hat{D},\ d_a)) & & \text{– } \textit{apply f} \\
\textbf{else}\ \delta(f,\ (D,\ d_a)) & &
\end{array}
$$

f, β, and α are sets of functions.

$\delta(G,\ (a_1,\ldots,a_n))$ applies $g \in G$ to a prefix of the arguments, $(a_1,\ldots,a_m)$, choosing the best function based on the types of the actual arguments and the types of functions in G. (*multiple dispatch*)

Fig. 13. AP-F Traversal Function Definition

The traversal adapter function sets represent three aspects of hand-coded traversals that one might write. To make the traversal adaptation sufficiently general, we define the three sets of functions as:

f : General transformations; run at each node of the data structure

β : Reconstruction or folds using transformed data from sub-traversals

α : Modification or replacement of traversal arguments.

[4] In most plausible implementation languages, Java in particular, mutation is still available so the user can mix the implementation language's features.

[5] The notion of *best function* is based on static attributes of functions: the number of argument accepted and formal parameter types.

When traversing a data type, we first choose an augmentor, in α, to *update* the traversal argument before traversing the fields of the object; this allows information to be passed *down* during traversal. We then traverse each *field* of the object, passing the new argument. Once all fields have been traversed, we dispatch to a builder, in β, which *combine*s these values; then dispatch to a transformer, in f, allowing a final modification to the value before returning to the caller.

The way we have formulated the traversal function minimizes data dependencies between individual values making it simple to parallelize. Each calculation of d'_i can be done in separate threads, implicitly synchronizing on the dispatch to β. Separating the traversal adaptation into three sets of functions also increases opportunities for reuse; allowing an AP-F implementation to provide suitable defaults for common development scenarios. Figure 14 describes the default functions we have found useful in practice. The id_f and id_α functions are straightforward, but the builders id_β and β_c are somewhat special. We chose the behavior of id_β to be *error* to help with debugging programs, before our static type checker was completed. The constructing builder, β_c, attempts to call the constructor of type C, the type of D, passing the traversal results as new fields. We can then use β_c to do functional updates to a data structure by implementing a transformer.

$$id_f(d,\ d_a) \Rightarrow d$$
$$id_\beta(d,\ ...) \Rightarrow error$$
$$\beta_c(D,\ d'_0, \cdots, d'_n,\ d_a) \Rightarrow \mathbf{new}\ C(d'_0, \cdots, d'_n)$$
$$id_\alpha(d,\ d_a) \Rightarrow d_a$$

Fig. 14. AP-F Default Function Definitions

The container checking solution in Figure 5 is defined as a function object, which extends id_β; covering all cases so there is never a call to the default, $error$. To create a traversal using the `CheckF` function object we implicitly use the default function, id_f, and ignore any traversal arguments[6], as shown in the static `check()` method.

4.2 Implementation: DemeterF

Our implementation of AP-F concepts is called DemeterF[14]. It is a generic traversal and function library written in pure Java that uses reflection for both data structure traversal and argument matching dispatch. The library provides the traversal function (a simple Java translation of $T_{f,\beta,\alpha}$ from Figure 13), the dispatch function, δ, and various combinations of the default functions defined in Figure 14 (described in Figure 15).

We use `function objects` to represent sets of functions, which allows users to override and overload methods to completely adapt the generic traversal. To differentiate the three types of functions within objects we use a different method name for each.

[6] Traversal arguments are optional in `DemeterF`.

The various sets of functions f, β, and α are implemented by writing `apply()`, `combine()`, and `update()` methods respectively. This allows us to assemble function objects that implement a number of methods of any kind.

`Traversal`	Generic reflective traversal function over all data types
`IDf`	Java implementation of id_f
`IDb`	Java implementation of id_β
`Bc`	Java implementation of β_c
`IDa`	Java implementation of id_α
`ID`	Java implementation of $(id_f \cup id_\beta \cup id_\alpha)$
`IDfa`	Java implementation of $(id_f \cup id_\alpha)$
`IDba`	Java implementation of $(id_\beta \cup id_\alpha)$

Fig. 15. DemeterF Provided Classes and Function Objects

Figure 15 describes the provided class names and the implementation of the traversal related functions. Most of the default implementations are as simple as:

```
class IDfa{
   Object apply(Object D, Object da){ return D; }
   Object update(Object D, Object da){ return da; }
}
```

Programmers can then use Java inheritance to overload/override methods, implementing desired functionality over the traversal. Figure 16 describes a few of the provided `Traversal` constructors and default function choices for each case. This shows the ability to combine function sets into a single function object, eliminating some details and providing extra flexibility when creating traversal solutions.

`Traversal(IDf f)`	≡	`Traversal(f, Bc, IDa)`
`Traversal(IDfa fa)`	≡	`Traversal(fa, Bc, fa)`
`Traversal(IDb b)`	≡	`Traversal(IDf, b, IDa)`
`Traversal(ID fba)`	≡	`Traversal(fba, fba, fba)`

Fig. 16. DemeterF Default Traversal Constructions

4.3 Dispatch: Function Selection

The last portion of AP-F to explain is our dispatch function, δ, which selects the best (most specific) function from a set, based on the types of actual arguments during traversal.

$$
\begin{array}{l}
\delta(G,\ (a_1 : C_1 \ldots a_n : C_n)) \Rightarrow \\
\quad \textbf{let}\ \ G' \leftarrow \{\ g(S_1 \ldots S_m) \in G \mid m \le n\ \wedge\ \forall i \le m.\ C_i \preceq S_i\ \} \\
\qquad\quad g \leftarrow head(\ sort(G',\ lessthan)) \\
\qquad\quad m \leftarrow arity(g) \\
\quad\ \textbf{in}\ \ g(a_1 \ldots a_m)
\end{array}
$$

$$
\begin{array}{l}
lessthan(\ g(S_1 \ldots S_n),\ h(U_1 \ldots U_m)) \Rightarrow \\
\quad (n > m)\ \ \textbf{or}\ (n = m\ \ \textbf{and}\ \ moreSpecific\,(\,(S_1 \ldots S_n),\ (U_1 \ldots U_n),\ n))
\end{array}
$$

$$
\begin{array}{l}
moreSpecific\,(\,(S_1 \ldots S_n),\ (U_1 \ldots U_n),\ n) \Rightarrow \\
\quad (n = 0)\ \ \textbf{or}\ (S_1 \prec U_1)\ \ \textbf{or} \\
\quad\ (S_1 = U_1\ \ \textbf{and}\ \ moreSpecific(\,(S_2 \ldots S_n),\ (U_2 \ldots U_n),\ n-1))
\end{array}
$$

Fig. 17. AP-F Dispatch Function Definition

Figure 17 describes our algorithm for function dispatch where $\prec$ is the traditional transitive, antisymmetric subtype relation and $\preceq$ is its reflexive extension. To select a function we first filter the set, leaving only those applicable to sub-sequences of the given argument types. We can then sort the functions in G' based on the defined comparison function *lessthan*; applying the *least* function, g, to the first m arguments provided.

The filter and implementations of *lessthan* and *moreSpecific* are chosen to allow the last few arguments to be optional. Placing functions with more arguments at the front of the list is a consequence of allowing optional arguments: we consider more arguments to be more information. This also allows functions with a larger number of more general arguments to be selected ahead of those with fewer but more specific arguments. Avoiding the algorithmic complexity of comparing function types with different numbers of arguments.

The function *moreSpecific* compares equal length sequences of argument types, stopping at the first inequality. This ensures that arguments at the front of the signature are given priority in function selection. It also compliments the inclusion of the original data element being traversed as the first argument— the most important argument for structuring adaptive code is also the most important in function dispatch. These functions (δ, *lessthan*, and *moreSpecific*) are implemented for DemeterF using Java reflection and type lists built from function objects that are used when creating a traversal. We simply compare the reflective types of traversal results with the sets of functions defined at the correct point in the traversal function.

4.4 Type Checking Traversals

Another benefit of this functional traversal organization is the ability to type check traversals. Here a type error is defined as the case when the filter step of the dispatch algorithm returns the empty set. Because of the way our default functions have been defined this can only occur when dispatching to user provided *builders*. Not surprisingly,

this kind of error can be caught with static information about the data structures to be traversed and the functions to be dispatched.

$$\frac{\triangleright\, D : u \quad u \in \text{BuiltIns} \quad \Delta(f,\ (u)) \rightarrow u'}{\triangleright\, \tau_{f,\beta}(u) : u'}$$

$$\frac{\triangleright\, D : u \quad u : u_1|\cdots|u_n \quad \triangleright\, \tau_{f,\beta}(u_i) : s_i \quad \exists u'.\forall i.s_i \preceq u'}{\triangleright\, \tau_{f,\beta}(u) : u''}$$

$$\frac{\triangleright\, D : u \quad u = \langle l_1\rangle\, u_1 \cdots \langle l_n\rangle\, u_n \quad \tau_{f,\beta}(u_i) : s_i \quad \Delta(\beta,\ (u,\, s_1 \ldots s_n)) \rightarrow u' \quad \Delta(f,\ (u')) \rightarrow u''}{\triangleright\, \tau_{f,\beta}(u) : u'}$$

Fig. 18. AP-F Traversal Typing Rules

Ignoring *augmentors* for simplicity, Figure 18 shows our three typing rules for AP-F traversals. We reuse a modified form of the DemeterJ Class Dictionary (CD) syntax to differentiate between type definitions. *Sum* (or *union*) types are represented with ':', using '|' to separate variants. *Product* (or *record*) types are represented with '=' using '$\langle \cdot \rangle$' for field definitions followed by their type. The judgment $\triangleright\ \tau_{f,\beta}(u)\ :\ s$ means traversing an value of type u returns an value of type s, while the type dispatch function, Δ, follows the selection algorithm described earlier, but produces the return type of the chosen function instead.

Though slightly informal[7], this description has been used to produce a static type checker for DemeterF, written in DemeterF. The only difficult portions of the algorithm are the recursion from two of the rules and the unification of subtypes in the third rule. Using this type checker we can rule out the possibility of traversal errors without needing an instance of a data structure.

4.5 Example Solutions

The solutions to the example problems in DemeterF use the argument matching to implement cases from our language descriptions. For most of these problems we chose to extend `IDb` because reconstruction (β_c) of the data structure is not needed. For each of the problems we will discuss any issues and design decisions involved.

Figure 19 shows our evaluation function for the example language. Each *combine* method is a translation of a rule from the example language semantics with the exception of `combine(Plus)`, which is implemented for completeness. Because rightmost arguments are optional in DemeterF, the first two combines (`IntExp` and `StrExp`) could instead take only one argument, looking inside `e` for the result. Because we use the default transformer, id_f, the return types of our `combine` methods are the result

[7] Complete formalization and proof of type safety are items of future work.

```
class Eval extends IDb{
    Integer combine(IntExp e, Integer i){ return i; }
    String  combine(StrExp e, String s){ return s; }
    String  combine(StrCast c, String s){ return s; }
    String  combine(StrCast c, Integer i){ return ""+i; }
    Integer combine(PlusExp p, Integer l, Plus op, Integer r){ return l+r; }
    String  combine(PlusExp p, String l, Plus op, String r){ return l+r; }
    Plus    combine(Plus op){ return op; }
}
```

Fig. 19. Example Language Evaluation in DemeterF

types of the traversal function. The arguments of each method correspond to the original object and the results of traversing any of its fields. For `PlusExp` there are two valid cases; we treat each of them, relying on the `IDb` default (*error*) for any runtime type errors.

```
class TypeCheck extends IDb{
    Type combine(IntExp i){ return Type.Int; }
    Type combine(StrExp s){ return Type.Str; }
    Type combine(Plus p){ return Type.Plus; }

    Type combine(PlusExp p, IntType l, Type op, IntType r){ return Type.Int; }
    Type combine(PlusExp p, StrType l, Type op, StrType r){ return Type.Str; }
    Type combine(PlusExp p){ return Type.Err; }

    Type combine(StrCast c){ return Type.Str; }
    Type combine(StrCast c, ErrType t){ return t; }
}
```

Fig. 20. Example Language Type Checker in DemeterF

Figure 20 shows our DemeterF type checker for the example language. Here in the first two `combine` methods we ignore the second argument (presumably `Integer` and `String` respectively) because it is not needed in the typing judgment. The first two cases for `PlusExp` are similar to evaluation, but we add a less specific case, `combine(PlusExp)`, to catch any invalid cases. Similarly for `StrCast`, we add a more specific method to catch type errors in sub-terms, leaving the more general case to return `Type.Str`.

Our compiler for the example language, in Figures 21 and 22, is slightly more complicated because we need to select different `Opcodes` for addition and concatenation. To demonstrate one of the strengths of AP-F we do this using a simple rewrite pass, which transforms `Plus` operators. We introduce two new variants, `PlusInt` and `PlusStr`, to represent addition and concatenation. The function object, `TypeTrans`, which extends `IDfa`, *updates* the traversal argument to be the `Type` of the outer `PlusExp`[8]. When reaching a `Plus` operator, the traversal will eventually dispatch to one

[8] Calling the type check traversal each time is inefficient; we are currently working on ways of composing traversals to eliminate this type of situation.

of our `apply()` methods— assuming the original expression type checks. Once `apply()` has been called, the parents are then reconstructed by the builder `Bc` (implicit in the traversal construction), producing an expression where `Plus` is replaced by the type correct variant.

```
class TypeTrans extends IDfa{
    // Pass the Type as a Traversal Argument
    Type update(PlusExp p, Object o)
     { return new Traversal(new TypeCheck()).traverse(p); }

    Plus apply(Plus pl, IntType i){ return new PlusInt(); }
    Plus apply(Plus pl, StrType i){ return new PlusStr(); }
}
class PlusInt extends Plus{}
class PlusStr extends Plus{}
```

Fig. 21. Typed Translation of Plus using DemeterF

Once polymorphic `Plus` is removed from the expression, our compilation step becomes simpler. The `Compiler` function object extends `IDb`, following a similar format to our evaluator. `IntExp`s and `StrExp`s produce push instructions— as with our type checker, these combine methods ignore the results of traversing their fields. When traversing our new plus operators we produce `Add` and `Concat` instructions; at a `StrCast` we expect an `OpList` from traversing the sub-expression, `append`ing a `CastS` instruction to the list.

```
class Compiler extends IDb{
    OpList single(Opcode o){ return new OpList().append(o); }

    OpList combine(IntExp i){ return single(new PushI(i.val)); }
    OpList combine(StrExp s){ return single(new PushS(s.val)); }
    Opcode combine(PlusInt pl){ return new Add(); }
    Opcode combine(PlusStr pl){ return new Concat(); }

    OpList combine(PlusExp p, OpList l, Opcode op, OpList r)
     { return l.append(r).append(op); }

    OpList combine(StrCast c, OpList l){ return l.append(new CastS()); }

    static OpList compile(Exp e){
       Type t = new Traversal(new TypeCheck()).traverse(e);
       if(t == Type.Err)
          throw new TypeException(t);
       Exp newe = (new Traversal(new TypeTrans()).traverse(e, Type.Err));
       return (new Traversal(new Compiler()).traverse(newe));
    }
}
```

Fig. 22. Example Language Compiler in DemeterF

The combine method for `PlusExp` is more complicated: we expect an `OpList` from each of the sub-expressions, and an `Opcode` from the operator, `PlusInt` or `PlusStr`. The result `OpList` is simply a concatenation of these lists in the right order. The static `compile()` method calls the type checker traversal. If the expression type checks, then we can safely transform plus operators. The return value is the result of traversing is a list of `Opcode`, translated using an instance of our Compiler function object.

5 Related Work

The three Demeter tools provide different implementations for AP related ideas. DemeterJ [8] is a source manipulation tool and implements Demeter visitors using the visitor design pattern and static traversal code generation. DJ [7] uses Java reflection to traverse objects, removing the need for `accept()` method definitions withing host classes. DAJ [15] uses Aspect Oriented Techniques [16, 17] to introduce the necessary traversal methods. In each of the tools, traversal control is defined using strategies, allowing certain changes [18] to a data structure without affecting the program's meaning. In all three tools communication during traversal is encoded in the tradition manner—via visitor fields. Our AP-P solution is an addition to these tools that removes the need for boilerplate code in some recursive computations. DemeterP uses DJ for implementing its dynamic traversals, as well as AspectJ for its implementation of interposition variables. AP-F is a more drastic extension in that it changes the way computation is decomposed along a traversal. The new decomposition allows communication through arguments and return values leading to purely functional computations. Functional style traversals are easier to parallelize and compose [19], though we haven't fully explored both those areas[9]. Our DemeterF implementation has been seamlessly deployed along side these tools without any extra modifications.

Environmental Acquisition [20, 21] is somewhat related to the contexts of AP-P's interposition variables. With environmental acquisition, information is acquired through the containment structure using explicit declarations. Interposition variables are implicitly available throughout the traversal within a given context and can be used to provide reverse links within composite structures.

Ovlinger and Wand [3] propose a domain specific language as a means to specify recursive traversals for use with the visitor pattern [4]. The language supports the addition of traversal arguments, calling of arbitrary functions during traversal, and functional style combination of intermediate results. The language provides traversal flexibility at a higher level than hand-coded traversals, but is not robust with respect to data structure changes. AP-P and AP-F are immune to some forms of structural changes, simply because traversal and computation are separate; the traversal can be adapted to changes in the data structure separately, leaving computations unchanged. The flexibility of the traversal language is such that it could be used to specify the traversals for the various incarnations of AP, including AP-P and AP-F.

[9] We have developed a working, parallel version of DemeterF, but are still working on traversal compositions

In [22] a functional visitor implementation of DJ [7] is presented that introduces `around()` visitor methods with the ability to control traversal. A second method argument of type `Subtraversal` captures the current traversal context, which can be ignored or continued. A generic `combine()` method accepts an `Object` array and is used to provide default behavior for `around`. The values returned from visitors and sub-traversals are unchecked and typically become `Object`, decreasing static type checking and safety, ultimately forcing runtime checks and many programmer inserted casts. Though the ideas are similar to AP-F, our traversal organization and dispatch functions relieve the programmer of casting and provide static guaranties of traversal result types.

Strategic Programming (SP) has its roots in term rewriting [23, 24], but has been used in other paradigms including Object Orientated Programming [25]. In SP, traversal computations are synthesized by passing a function to an appropriate traversal schema. Programmer definable schemas are compositions of traversal primitives which are applied to a nodes immediate children (or fields). The OOP incarnation of SP is implemented through a generalized visitor pattern, in which object structures implement a predefined Java interface. Visitor methods always return an object of the visitable interface; traversal primitives and compositions are achieved through parametrized constructors. Although SP is more flexible with regards to traversal specification (*e.g.*, top-down or bottom-up [26]) the type limitations force programmers into heavy use of casting. The use of generics could remove the need for casting although it is not clear if it could be removed completely. AP-F does not impose any extra type restrictions, while the extended method dispatch mechanism can be used to implement limited form of the traversal primitives found in SP.

The goal of Scrap Your Boilerplate (SYB) [27–29] is to automatically traverses data structures, using developer provided functions that perform transformations. Generic traversal functions take a combinator and specify what nodes in the data structure a function should be applied to. The traversal combinator's argument is itself a function that transforms data types of interested, acting like *id* for others. SYB provides transformations that can be type-unifying (TU), were each recursive traversal returns the same type, or type-preserving (TP), where each returns the same type as its input type. AP-F abstracts these ideas a bit more, separating traversal, transformation, and construction so intermediate functions can return any type. TU and TP transformations then become special cases. In addition, AP-F does not restrict the use of mutation, while mutation in SYB must to encapsulated inside monads, though this has more to do with the implementation languages, rather than the concepts.

6 Conclusion

We presented two refinements to AP: AP-P and AP-F. AP-P introduces *interposition variables*; visitor variables that are only available during traversal and can be used to communicate information between different executions of visitor methods within the same traversal context. Interposition variable come with implicit updates that mimic the recursive traversal's structure, alleviating programmers from writing boilerplate code. AP-F is a functional formulation of AP that decomposes traversal computation into

three sets of functions. An extended method dispatch mechanism during traversal allows communication to occur through function arguments and return values leading to purely functional computations. The traversal decomposition allows programmers to easily build transformations that, due to their functional nature, lend themselves to easy parallelism.

References

1. Appel, A.W.: Modern compiler implementation in Java. Cambridge University Press, New York, NY, USA (1998)
2. Felleisen, M., Friedman, D.P.: A little Java, a few patterns. Massachusetts Institute of Technology, Cambridge, MA, USA (1998)
3. Ovlinger, J., Wand, M.: 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, New York, NY, USA, ACM Press (1999) 70–81
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
5. Abdelmeged, A., Lieberherr, K.J.: Recursive adaptive computations using perobject visitors. In: OOPSLA Companion. (2007) 825–826
6. Abdelmeged, A., Lieberherr, K.: Recursive Adaptive Computations Using Interposition Visitors. Technical Report NU-CCIS-07-06, Northeastern University, Boston (May 2007) OOPSLA 2007 Poster.
7. Orleans, D., Lieberherr, K.J.: DJ: Dynamic Adaptive Programming in Java. In: Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, Springer Verlag (September 2001) 8 pages.
8. The Demeter Group: The DemeterJ website. http://www.ccs.neu.edu/research/demeter (2007)
9. Sun Microsystems: The Java Compiler Compiler. Website (2007) http://javacc.dev.java.net.
10. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Proc. Practical Aspects of Declarative Programming PADL 2002. Volume 2257 of LNCS., Springer-Verlag (January 2002) 137–154
11. Abdelmeged, A.: Demeterp annotation processor. Website (2007) http://www.ccs.neu.edu/home/mohsen/interposition/.
12. : The aspectj project. Website http://java.sun.com/j2se/1.5.0/docs/guide/apt/.
13. : The aspectj project. Website http://www.eclipse.org/aspectj/.
14. Chadwick, B.: Demeterf library and examples. Website (2007) http://www.ccs.neu.edu/home/chadwick/demeterf/.
15. The Demeter Group: The DAJ website. http://www.ccs.neu.edu/research/demeter/DAJ (2007)
16. Filman, R.E., Elrad, T., Clarke, S., Ak$c_s$it, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
17. : Aspect oriented software design, http://www.aosd.net
18. Skotiniotis, T., Palm, J., Lieberherr, K.: Demeter Interfaces: Adaptive programming without surprises. In: European Conference on Object Oriented Programming. (2006)
19. Chadwick, B., Skotiniotis, T., Lieberherr, K.: Functional Visitors Revisited. Technical Report NU-CCIS-06-03, Northeastern University, Boston (May 2006)

20. Gil, J., Lorenz, D.H.: Environmental acquisition: a new inheritance-like abstraction mechanism. In: OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (1996) 214–231
21. Cobbe, R., Felleisen, M.: Environmental acquisition revisited. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2005) 14–25
22. Wu, P., Krishnamurthi, S., Lieberherr, K.: Traversing recursive object structures: The functional visitor in demeter. In: AOSD 2003, Software engineering Properties for Languages and Aspect Technologies (SPLAT) Workshop. (2003)
23. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), ACM Press (September 1998) 13–26
24. Lämmel, R.: Typed Generic Traversal With Term Rewriting Strategies. Journal of Logic and Algebraic Programming **54** (2003) Also available as arXiv technical report cs.PL/0205018.
25. Visser, J.: Visitor combination and traversal control. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2001) 270–282
26. Lämmel, R., Visser, E., Visser, J.: Strategic programming meets adaptive programming. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2003) 168–177
27. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. Volume 38., ACM Press (March 2003) 26–37 Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
28. Lämmel, R., Peyton Jones, S.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004), ACM Press (2004) 244–255
29. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), ACM Press (September 2005) 204–215