# Functional Adaptive Programming

Bryan Chadwick
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
chadwick@ccs.neu.edu

Karl Lieberherr
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
lieber@ccs.neu.edu

## ABSTRACT

We present a functional extension of Adaptive Programming (AP-F), supported by a library and set of tools (collectively called DemeterF) that provide a safe form of static and dynamic AOP over data structure traversals. A code generator (DemFGen) statically merges structural, behavioral, and data-generic descriptions into class definitions, while a dynamic parametrized traversal library (DemeterF) supports three aspects of functional traversals: contexts, folding, and control. A custom multiple dispatch dynamically chooses functions of our aspects (function objects) based on their declared signature and the types of recursive traversal results. Programmers can define each aspect separately while types capture dependencies between them, assuring that the traversal computation cannot "go wrong". Functional OO programs written using DemeterF become easily parallelizable and can automatically adapt to some changes to the underlying data structures, which can be confirmed by our type checker.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

## General Terms

Languages,Design

## Keywords

Data structure traversal, functional adaptive programming, object oriented programming

## 1. INTRODUCTION

Functional Adaptive Programming (AP-F) is a programming technique improving both object-oriented programming (we have a polished implementations in both Java and C#) as well as functional programming (we have a prototype Scheme implementation). AP-F extends the original notions of adaptive programming [16] providing three key innovations: supurb abstraction of programmer controlled traversals, immediate support for parallelization of functional traversals, and a fine-grained type checker to control the development and evolution of structure-shy programs.

Traversals are everywhere in programs, from simple for-loops to full blown mutually recursive data structures. In Object-Oriented (OO) Programming Languages like Java and C# the *dominant decomposition* of programs is classes, which encapsulate both structure (data) and behavior (functions). This makes certain kinds of operations over a collection of classes difficult to perform without breaking the standard abstraction boundaries present in the language. When using encapsulation, some operations are easy (though tedious) to implement, but are usually spread throughout classes and require a fair amount of extraneous code dealing with subcomponents. For these instances there has been much work on visitors [11] to allow the implementation of operations from *outside* a given hierarchy. Visitors can be made more *structure shy* [16] to allow them to adapt to data structure changes.

In contrast to OO encapsulation, the mantra of Aspect Oriented Programming (AOP) is the *modularization of crosscutting concerns* [14], where we develop with abstractions that support the separation of program aspects that typically overlap. With aspects we can declare methods that implement a new operation, wrap existing methods to support a new interface, and influence a running program by executing code at dynamic join points. While AOP is very useful, it can be tricky to wield its power safely [29]. Typically computation is performed using mutation (side-effects), which can be difficult to reason about, even informally. This becomes increasingly important when we wish to develop concurrent programs with the advent of implicitly parallel processors.

As a compromise, we present a functional extension of Adaptive Programming [16, 17] (AP-F), supported by a library and set of tools (collectively called DemeterF [8]) that provide a safe form of static and dynamic AOP over data structure traversals. A code generator (DemFGen) statically merges structural, behavioral, and data-generic descriptions into class definitions, with particular support for modularity and generics. Generated classes are, by design, well suited for use with a dynamic parametrized traversal library (DemeterF) that supports three aspects of functional traversals: contexts, folding, and control. Traversal contexts can

be functionally updated for sub-components at specific types and recursive traversal results can be folded into a single return value. The control aspect guides the generic traversal function through the data structure, while a custom multiple dispatch dynamically chooses functions (methods) from the aspects based on their declared signatures and the types of recursive sub-traversal results. Programmers can provide each aspect separately, while types capture the dependencies between them, assuring that the traversal computation cannot "go wrong".

A particular advantage of our functional traversal organization is the ease at which we can parallelize a seemingly sequential computation. In addition to sequential traversals, DemeterF contains a parallel traversal class that executes the traversal of fields independently in separate threads. Implicit synchronization gives the traversal the ability to provide plugable heuristics for branching to limit the overhead of extraneous thread creation. Though we have only begun to explore library optimizations, the parallel nature of DemeterF traversals allows us to make up for some of slowdowns associated with Java reflection. Limited experiments with DemeterF have shown average improvements of 20% on a dual-core system, with up to 40% using our C# version.

As proof of our system's usability, the class generator (DemFGen) is written using our traversal library and is now implemented in itself. Our type checker is written using the generator and the library, and has been used to ensure that added features and changes to the DemeterF structures remain safe with respect to the traversal based implementation. We have also used the DemeterF system successfully in both graduate and undergraduate level classes at Northeastern University, implementing various programming language problems and artificial market simulations.

The rest of this paper is organized as follows: Section 2 introduces a motivating example and compares other solutions to our approach. Section 3 details DemeterF parametrized functional traversals, our type checking, and discusses preliminary parallel traversal results. Section 4 describes our class generator DemFGen, and Section 5 presents a more complex DemeterF example of an expression compiler. We review related work in Section 6, and conclude in Section 7 with a suggestion of future work.

## 2. MOTIVATION

To motivate our traversal abstractions and development tools, we present a simple data structure representing arithmetic expressions. Figure 1 shows a bit of Java syntax that describes the structures involved.

```
abstract class exp{}
class num extends exp{
  int val;
}
class bin extends exp{
  oper op;
  exp left, right;
}
abstract class oper{}
```

```
class var extends exp{
  ident id;
}
class def extends exp{
  ident id;
  exp e, body;
}

class sub extends oper{}
```

**Figure 1: Expression Structures**

For the sake of brevity we elide constructors and access modifiers (e.g., `public`). These structures represent a simple language with any number (a chain) of variable definitions followed by an expression. Using the classes as our abstract syntax, we could construct a simple term representing the definition:

```
x = 5;  (- 4 x)
```

As the Java expression:

```
new def(new ident("x"), new num(5),
        new bin(new sub(), new num(4),
                new var(new ident("x"))))
```

The writing of class definitions can become tedious, and tends to interleave both structure (inheritance/containment) and behavioral (method) concerns. There are also structure based functions, operations such as parsing, printing, and equality, that can be written generically based on the structure itself, rather than any specific instance. The DemFGen class generator merges these three separate aspects of class definitions into Java code, allowing the programmer to give a succinct description of the structure along with internal behavior and generic functions. An added benefit of our organization is that we can provide different target languages for DemFGen, in particular, we support both Java and C# versions of DemeterF.

Once we have an `exp` instance, we probably want to be able to calculate its final result by recursively evaluating each `def`, and applying the accumulated bindings in the final expression (`body`). An operation such as `eval(...)` can be implemented in a number of ways. Assuming a functional implementation of environments (*i.e.*, the `env` class), Figure 2 shows a straight-forward OO-style implementation with comments describing in which class to place each snippet. For each abstract class we introduce an abstract method; for each concrete class we implement the specific version of `eval(...)`, recursively calling where needed. If we didn't happen to have access to the original program text we could use something similar to AspectJ inter-type declarations [1].

```
/* exp */ abstract int eval(env ev);
/* oper*/ abstract int eval(int l, int r);

/* num */ int eval(env ev){ return val; }
/* var */ int eval(env ev){ return ev.apply(id); }
/* sub */ int eval(int l, int r){ return l-r; }

/* bin */ int eval(env ev)
{ return op.eval(left.eval(ev), right.eval(ev)); }
/* def */ int eval(env ev)
{ return body.eval(ev.extend(id, e.eval(ev))); }
```

**Figure 2: OO `exp` Internal Evaluation**

A few things to notice from this simple implementation are:

1. Cooperating methods are scattered throughout the classes
2. The method argument, representing context, is passed everywhere, though it is not used often
3. Recursive calls are explicit
4. The true direction of the traversal is somewhat hidden
5. Traversal order is encoded in nested calls

While inter-type declarations (or static aspects) certainly solve the first issue, visitor based techniques [11, 16, 17, 25] have been developed in order to address the others. In its original incarnation, the Visitor Pattern [11] only requires a simple `accept(visitor)` method to aid in the implementation of a double dispatch mechanism. Using `accept(...)`, different operations over the structures can be implemented by encoding traversal within the visitor. Figure 3 shows a visitor implementation of the same `eval(...)` functionality.

```
class EvalVis extends visitor{
  Stack<Integer> stk = new Stack<Integer>();
  env ev = env.empty();

  void visit(num n){ stk.push(n.val); }
  void visit(sub s){ stk.push(stk.pop()-stk.pop()); }
  void visit(bin b){
    b.right.accept(this);
    b.left.accept(this);
    b.op.accept(this);
  }

  void visit(var v){ stk.push(ev.apply(v.id)); }
  void visit(def d){
    d.e.accept(this);
    ev = ev.extend(d.id,stk.pop());
    d.body.accept(this);
    ev = ev.unextend();
  }
  static int eval(exp e){
      EvalVis v = new EvalVis();
      e.accept(v);
      return v.stk.peek();
  }
}
```

**Figure 3: Visitor exp Evaluation**

The tedium of the visitor solution comes from the fact that we must write out the traversal ourselves (the calls to `accept`), even if they can be inferred from the structural definitions. What is also clear from this visitor is that any implementation using mutation must encode traversal order in side-effects. In this instance, values on the stack must be pushed/popped in the right order. Due to the `java.util.Stack` implementation, we traverse the `right` expression of a `bin` first, so that the `left` will be placed on the top of the stack last. Though the use of side-effects allows us to eliminate the passing of the environment, there is no warning when data structures change. If, for instance, we add a new variant of `exp`, the visitor continues to compute a value, though it will probably not be the one we want. The compositional nature of the hand-written traversal makes it reasonable (though tedious) to parallelize, but the parallelization of visitors is difficult since side-effects are very sensitive to traversal order.

To eliminate the problems associated with both visitors and hand-coded traversals, we introduce the DemeterF traversal library. DemeterF provides a generic traversal that can be advised by three different aspects. The first, of type `Augmentor`, is responsible for updating the context information (*i.e.*, a traversal argument) for child nodes of a data structure. The second, `Builder`, folds together the recursive results from children and its context into a single value. Finally, `Control` guides the traversal through the data structure by describing which fields should be traversed.

Figure 4 shows a complete DemeterF implementation of `exp` evaluation. The `Eval` class extends `ID`, a DemeterF class that implements both the `Builder` and `Augmentor` interfaces. To evaluate an expression, we create a static `Traversal` object that uses a new `Eval`, and bypasses (skips) the `body` field of `def` instances.

```
class Eval extends ID{
  static Traversal trav = new Traversal(new Eval(),
            Control.bypass("def.body"));

  static int eval(exp e, env ev)
  { return trav.<Integer>traverse(e, ev); }

  int combine(num n, int i){ return i; }
  sub combine(sub s){ return s; }
  int combine(bin b, sub s, int l, int r){ return l-r;}

  int combine(var v, ident i, env ev)
  { return ev.apply(i); }
  int combine(def d, ident id, int e, exp b, env ev)
  { return eval(b, ev.extend(id, e)); }
}
```

**Figure 4: DemeterF exp Evaluation**

After recursively traversing the selected fields of each instance, the traversal passes the results along with the context (the `env`) to the most specific `combine` method in `Eval`. The original object is included as the first parameter to allow methods to match the type of the parent instance that was traversed. Essentially, while traversing an object, we compute the results at leafs, and push the results up to containing nodes. As a result, the computation contains almost no traversal code and mentions the context only when needed. The functional nature of our computation allows the external traversal to execute in parallel and provides more type information about expected data structures, which can be checked to ensure safety.

## 3. DEMETERF TRAVERSALS

To introduce DemeterF traversals and control, we use typical OO binary search tree (BST) structures, with a functional interpretation. Figure 5 shows simple Java classes implementing a BST class with two variants: `leaf` and `node`. Again, we leave out constructors and access modifiers for space.

```
abstract class bst{}

class leaf extends bst{}
```

```
class node extends bst{
  int data;
  bst left, right;
}
```

**Figure 5: BST Structures**

The main contribution of the DemeterF system is our traversal library. The library includes interfaces and classes that implement a generic, depth-first traversal through an instance of a data structure. The traversal is parametrized by three *function objects* (instances of *function classes*), which provide advice in the form of specially named methods. When creating a `Traversal`, the programmer passes instances of the interfaces, `Builder` and `Augmentor`, and a class, `Control`. For programmer convenience the implementation uses defaults if `Augmentor` or `Control` is not specified.

## 3.1 Basics

`Builder`s implement `combine` methods that fold recursive traversal results into a single value. Because we often want to describe both the `Builder` and `Augmentor` in a single class, the library provides one, `ID`, that implements both and includes methods for primitive types (*i.e.*, `int`, `boolean`, *etc.*), that act as the identity function. The first parameter passed to a combine method is the original object being traversed, while the others are the result of recursively traversing each of its fields. Our implementation uses Java reflection, so fields are traversed in the order they are defined in the program text. Figure 6 shows a simple `toString` function over BSTs that returns a string of nested named parenthesis representing the given tree.

```
class ToString extends ID{
  String combine(leaf l){ return "(leaf)"; }
  String combine(node n, int d, String l, String r)
  { return "(node "+d+" "+l+" "+r+")"; }

  static String toString(bst t){
    return new Traversal(new ToString())
                .<String>traverse(t);
  }
}
```

**Figure 6: BST `toString`**

The `static` method `ToString.toString(...)` creates a new `Traversal`, passing a `ToString` function object (instance). The default `Control` directs the traversal to continue everywhere, which corresponds to the static creator `Control.everywhere()`. We call the `traverse` method, passing the object we wish to traverse; in this case, there no context needed. The default `Augmentor` implemented within `ID` is the identity function for contexts and is unused when `traverse` is only given a single parameter.

`Augmentor`s implement `update` methods that are called before traversing each field of an object. The `update` methods allow programmers to modify the context (or traversal argument) for separate fields of a specific type. Figure 7 shows a top-down calculation of `bst` height. The `update` method increments the traversal context (`dp`, representing the current depth) for *any* field of a `node`.

```
class Height extends ID{
  int combine(leaf l, int dp){ return dp; }

  int update(node n, Fields.any f, int dp)
  { return dp+1; }
  int combine(node n, int d, int l, int r)
  { return Math.max(l,r); }

  static int height(bst t){
    return new Traversal(new Height())
                .<Integer>traverse(t, 0);
  }
}
```

**Figure 7: BST Top Down `Height`**

While traversing, the `Traversal` match's the `update` signature by looking for an inner class with the same name as the field, a *field class*, *e.g.*, `node.left`. If this class is defined then an instance is created, and the signature is matched with the traversal context passed as the last parameter. If the field class is not defined, we match against the DemeterF class `Fields.any`. In this case the field class does not exist, but we can still update the context for any field of a `node`. When calling the `traverse` method we must pass a second argument that represents the root context, in this case, the integer zero (0).

When a traversal context is used, it is passed as the last parameter to `combine` methods, after any recursive results. The `combine` methods within `Height` pass the maximum calculated depth back up the `bst`: at a `leaf` the context is the current depth; at a `node`, we chose the maximum depth of the `left` and `right` `bst`s. Note that the context is not mentioned in the `combine` method for `node`s; any parameters that are not needed can be left off the end of `update` and `combine` method signatures.

## 3.2 Control

Until now we have only shown two of the three traversal parametrizations, `Builder` and `Augmentor`, with implementations that are subclasses of `ID`. In DemeterF we also provide separate traversal control similar to that found in DemeterJ and DJ[28, 25], but through the use of static creators, rather than a domain specific strategy language[1]. Figure 8 shows a function class that traverses a `bst` using the `onestep` traversal. `Traversal.onestep()` returns a traversal that allows the programmer to step into an object, and retrieve the values of its fields as parameters for matching. In this case, the function only needs the first parameter to determine the result, so the other parameters are not included in the method signature for `node`.

```
class IsLeaf extends ID{
  boolean combine(leaf l){ return true; }
  boolean combine(node n){ return false; }

  static boolean isLeaf(bst t){
    return Traversal.onestep(new IsLeaf())
              .<Boolean>traverse(t);
  }
}
```

**Figure 8: BST `IsLeaf`**

Using `onestep`, we can completely eliminate field accesses and instance checks, allowing the traversal matching to do all the hard work, similar to pattern matching in functional programming languages. Figure 9 shows a class that implements a functional `insert` for `bst`s using the `onestep` traversal.

In this case the traversal context is used to pass the `int` value to be inserted. The traversal calls the matching `combine` method after stepping into each object. For a `leaf` we return a new `node` containing the inserted data. If the method for `node` matches, then we compare the inserted value to be sure it goes into the correct sub-tree, reconstructing resulting the `node` after insertion. When traversing with `onestep`, the `Control` object used is actually `Control.nowhere()`, telling the traversal not to explore any edges, but there are several other useful creators for various scenarios. Figure 10 shows a

[1]Support for DemeterJ style strategies is available in a separate (non-default) DemeterF build.

```
class Insert extends ID{
  bst combine(leaf l, int nd)
  { return new node(nd,l,l); }
  bst combine(node n, int d, bst l, bst r, int nd){
    if(nd <= d)
      return new node(d, insert(l,nd), r);
    return new node(d, l, insert(r,nd));
  }

  static Traversal
    trav = Traversal.onestep(new Insert());
  static bst insert(bst t, int d)
  { return trav.<bst>traverse(t, d); }
}
```

**Figure 9: BST `Insert`**

traversal implementation of minimum, returning the smallest value stored in a given `bst`. To guide the traversal we pass a control object, constructed with `Control.only(...)` that tells the traversal to recurse into the `left` field of a `node`, which is exactly where the minimum value will be.

```
class Min extends ID{
  leaf combine(leaf l){ return l; }
  int combine(node n, int d, leaf l){ return d; }
  int combine(node n, int d, int mn){ return mn; }

  static int min(bst t){
    return new Traversal(new Min(),
      Control.only("node.left")).<Integer>traverse(t);
  }
}
```

**Figure 10: BST `Min`**

To eliminate conditionals from our methods, we return the left-most `leaf`, and match based on the type of the calculated value of a to choose the leftmost `data`. Because the `right` field of a `node` is not relevant in the calculation we leave it out of our method signatures. It is worth pointing out that this traversal may actually return a `leaf` if it is called on one. We leave it up to the programmer to call `Min.min` on non-leafs, though we could throw an exception within the static `min` method to be sure the return value is safe[2].

```
if(IsLeaf.isLeaf(t))
  throw new RuntimeException("No Min for Leafs");
```

The opposite of `Control.only(...)` is the creator `Control.bypass(...)`, which describes the fields that should *not* be traversed. Figure 11 shows a similar implementation that returns the maximum value in a `bst`. The main difference here is that we ignore the `left` field of a `node`, but it is kept in the signature as a place holder. Because the bypassed field could be any `bst`, we use the more general type to avoid handling all cases in separate methods.

## 3.3 Transformations

When traversing functional data structures we usually want to make a change to a specific part, reconstructing the rest of the structure, leaving it otherwise unchanged. In order to support functional updates using DemeterF, our library

[2]The Java compiler actually inserts an `Integer` cast before returning, though a class-cast exception is less informative than a custom description.

```
class Max extends ID{
  leaf combine(leaf l){ return l; }
  int combine(node n, int d, bst l, leaf r){return d; }
  int combine(node n, int d, bst l, int mx){return mx;}

  static int max(bst t){
    return new Traversal(new Max(),
      Control.bypass("node.left")).<Integer>traverse(t);
  }
}
```

**Figure 11: BST `Max`**

provides a function class, `Bc`, that reconstructs a copy of the traversed structure. The inherited `combine` methods can be overridden to transform a particular type or section of a structure.

```
class Incr extends Bc{
  int combine(int i){ return i+1; }

  static bst incr(bst t)
  { return new Traversal(new Incr()).<bst>traverse(t);}
}
```

**Figure 12: BST Increment**

As an example, Figure 12 shows a function class that extends `Bc`, incrementing each `data` field in a given `bst`. One of the benefits of extending `Bc` (the *constructing* `Builder`), is the fact that is can easily adapt to changes in the structures and any adjustments to traversal control, which can be used to both optimize and limit the extent of transformations. Figure 13 shows a method that uses our `Incr` function class to increment just the `right` spine of a given `bst`.

```
  static bst incrRight(bst t){
    return new Traversal(new Incr(),
      Control.bypass("node.left")).<bst>traverse(t);
  }
```

**Figure 13: BST Right Increment**

More complex transformations are possible, simply by overriding the `combine` methods for compound types. Figure 14 shows a function class that implements left rotation over `bst`s. When the `right` branch of a `node` is also a `node`, we can rotate it to the left, maintaining the invariant. When used in traversal, `Bc` supports the implementation of generic-map and type-based transformations, similar to *Scrap Your Boilerplate* (SYB) [15] in Haskell.

```
class RotL extends Bc{
  bst combine(node n, int d, bst l, node r){
    return new node(r.data, new node(d, l, r.left),
                    r.right);
  }
  static bst rotLeft(bst t){
    return new Traversal(new RotL()).<bst>traverse(t);
  }
}
```

**Figure 14: BST Left Rotate**

## 3.4 Types

As a model of our functional traversals we view a *function object* as a set of functions. Given the complete traversal information: function signatures, data structures, a control description, and a starting class, we can type-check the functions with respect to the implied traversal. In the case of `Insert` (Figure 9) this checking is straight forward because the `onestep` traversal is used: we check the types of each class' fields against the methods to be sure all cases are handled and they return the correct types. When checking `ToString` (Figure 6) the situation is only slightly different, as the `String` values for the `left` and `right node` fields are calculated recursively by the traversal.

Each method signature of the form:

```
R_0 combine(C c, R_1 f1, ..., R_n fn)
```

with a class definition of:

```
class C{
   F_1 f1;
    ...
   F_n fn;
}
```

places constraints on both the return type (*e.g.*, `R_1`) of a traversal of each corresponding field type of `C` (*e.g.*, `F_1`), and the possible return type(s) (here just `R_0`) of a traversal of an instance of `C`. In the case of `ToString`, the constraints from the method parameter types amount to the following:

- Traversal of a `bst` should return a `String`
- Traversal of an `int` should return an `int`

Because there are only two sub-classes of `bst`, we conclude from the return types of the methods that traversal of a `bst` does indeed return a `String`, and the implementation of `ID` actually contains a method for `int` that returns `int`.

For `Min` and `Max` (Figures 10 and 11), the situation is a little more complicated, since the return types of the `combine`s are different. For `Min`, the `Control` used specifies that the `left` field of a `node` will not be traversed, but we can infer the following constraints for the other fields:

- Traversal of an `int` should return an `int`
- Traversal of a `bst` should return one of {`int`, `leaf`}

The return types of the methods give us information about what the traversals will return:

- Traversal of a `leaf` returns a `leaf`
- Traversal of a `node` returns an `int`

In general, it is possible to type check traversals over non-recursive types with a simple depth-first walk of the data structure description. With recursive types, constraints are required to capture what parameter types the methods handle, so they can be checked after inferring the actual traversal return types. We consider a *sums-of-products* representation of types, similar to ML [22] and Haskell [13], rather than adhoc inheritence, though these techniques are applicable to both given a consistent traversal order over inherited fields. Generated constraints are of the form:

- Traversal of an `X` returns a *subtype-of* `Y`

where `Y` is *set* of classes. The *subtype-of* relation in this case is defined over *sets* of classes:

$$(X, Y) \in \textit{subtype-of} \;\text{ if }\; \forall\, x \in X\,.\, \exists y \in Y\,.\; x <: y$$

with $<:$ defined as the reflexive, transitive closure of the `extends` relation over classes in the data structure. If the constraints are satisfied by the traversal return types then we can be sure that the traversal will not produce an error, meaning that selection of an applicable method will never fail.

Similar constraint based type-checking systems have been used to type-check pure object oriented programs [26], though here we solve a slightly simpler problem. Following these ideas we have implemented a type-checker that computes the return type of a given traversal. It has been used successfully to verify the examples in this paper, and our class generator, which is implemented using DemeterF traversals. The commands used to type check the examples in this paper are available with the rest of the code [7].

## 3.5 Parallelization

One of the major draws of functional programming is its potential for parallel computation. Eliminating side-effects also seems to have a profound effect on multi-processor performance. The DemeterF traversal model was designed with these issues in mind to keep traversal separate from computation. This separation allows programmers to substitute a provided multi-threaded traversal (`ParTraversal`) for any instance of `Traversal` without affecting traversal return values.

As a preliminary test we have implemented a `bst` sum operation and run a series of DemeterF traversals on a dual-core machine[3]. Figure 15 shows our `Sum` function class, with static methods for sequential and parallel data structure traversal. As expected, the methods for `node` and `leaf` add up all the values stored in the `bst`. The novel feature here is that we can use the same function class, `Sum`, for both single and multi-threaded traversals.

```
class Sum extends ID{
  int combine(leaf l){ return 0; }
  int combine(node n, int d, int l, int r){
    return d+l+r;
  }

  static int sum(bst t, Traversal trv)
  { return trv.<Integer>traverse(t); }

  static int seqsum(bst t)
  { return sum(t, new Traversal(new Sum())); }
  static int parsum(bst t)
  { return sum(t, new ParTraversal(new Sum())); }
}
```

**Figure 15: BST Sequential and Parallel `Sum`**

To measure the difference between the two traversals, we ran a series of tests with `bst`s of various heights. Figure 16 shows our preliminary results for DemeterF sequential and parallel traversals. The calculated times (for **Seq.** and **Par.**) are

[3] An Intel® Core™ 2 Dou desktop

milliseconds, averaged over 10 runs of a balanced `bst` with the given number of `nodes`.

| BST Size | Seq. | Par. | Speedup |
|---|---|---|---|
| $2^5$ | 49 | 49 | 0.0 % |
| $2^6$ | 54 | 52 | 3.7 % |
| $2^7$ | 67 | 60 | 10.4 % |
| $2^8$ | 90 | 72 | 20.0 % |
| $2^9$ | 126 | 95 | 24.6 % |
| $2^{10}$ | 165 | 129 | 21.8 % |
| $2^{11}$ | 255 | 190 | 25.4 % |
| $2^{12}$ | 417 | 322 | 22.7 % |
| $2^{13}$ | 575 | 444 | 22.7 % |

**Figure 16: Performance of BST Sum**

We compare only the two DemeterF traversals due to the cost of Java reflection when comparing method signatures. There is room for optimizations within the library, but the consistency of speedup with multiple threads is very promising.

# 4. DATA DESCRIPTION

Separating data structures from class behavior and implementation is important for program readability, modularity, and reuse. To support a static aspect oriented style of programming, we have developed a class generator, named DemFGen, specifically for use with DemeterF. Incidentally, the generator is written in DemeterF, and has been a good test/benchmark of the library, the type checker, and now, itself. In the spirit of other adaptive programming tools such as DemeterJ [28], our class generator accepts a class dictionary (CD) file that specifies the structure of data types and a behavior (BEH) file that describes static code to be injected into the generated classes.

## 4.1 Basics

Our class dictionary syntax is slightly simplified from DemeterJ, but includes all of the main features, with a few of its own. Figure 17 shows a CD file that describes the `bst` structures defined earlier. Abstract classes are defined with a colon (`:`), separating variants with a vertical bar (`|`). Concrete classes are defined using equals (`=`), with field names in brackets (`<·>`), followed by their type. All definitions are terminated with a period (`.`), and concrete syntax strings are allowed before and after field definitions, supporting the creation of customized parsers/printers for the generated structures.

```
// bst.cd
bst: node | leaf.
node = "(node" <data> int <left> bst
                          <right> bst ")".
leaf = .
```

**Figure 17: BST Class Dictionary**

The behavior injection permits a static form of advice, similar to AspectJ's inter-type declarations. Figure 18 shows a BEH file that completes the `bst` class definition by inserting a few method stubs calling our earlier external implementations. For improved modularity, DemFGen supports the inclusion of other CD and BEH files. Previously written/generated classes can be used by writing a CD file with definitions preceded by the `nogen` or `extern` keywords.

```
// bst.beh
bst{{
  boolean isLeaf(){ return IsLeaf.isLeaf(this); }
  int min(){ return Min.min(this); }
  int max(){ return Max.max(this); }
}}
```

**Figure 18: BST Added Behavior**

## 4.2 Parametrized Classes

DemFGen also has extended support for parametrized classes. We allow type parameter bounds, any depth of nested type parameters, and can generate parsing and printing methods for all classes. Figure 19 shows a generic version of the `bst` CD file, storing `Comparable` elements.

```
// genbst.cd
extern interface Comparable(X): .
bst(X:Comparable(X)) : node(X) | leaf(X).
node(X:Comparable(X)) = <data> X <left> bst(X)
                                 <right> bst(X).
leaf(X:Comparable(X)) = .
```

**Figure 19: Generic BST Structure**

Type parameters are introduced in parenthesis with an optional bound placed after the colon; multiple parameters can be separated by commas. The `extern` keyword tells DemFGen not to generate anything for this definition; it is used to complete other definitions and for the checking of names and number of type parameters. Once we have a generic class, we can use it to generate parsers/printers for specific uses of the data structure. Figure 20 shows a CD file that includes the generic `bst` definitions, and wraps it in a concrete class. Once generated, the `intbst` class contains static `parse(...)` methods that support parsing instances of `bst<Integer>`.

```
// usebst.cd
include "genbst.cd";

intbst = <tree> bst(Integer).
```

**Figure 20: Generic BST Use**

The power of correctly parametrized classes can be fully realized when mixing syntax with definitions. Figure 21 shows CD and BEH files that describes a generic parenthesis `wrap` class. The uses of `wrap` allow us to parse `Strings` and `Integers` within two sets of parenthesis. When nesting parametrized classes we must avoid left recursion and other pitfalls dealing with recursive parameter substitution [18].

Other modifier keywords in DemFGen (`nogen` and `noparse`) specify not to generate code or a parser (respectively) for the given definition. For previously written library classes, `nogen` allows us to create parsers and printers based on the structure and syntax in the given CD file. Developers can then change the concrete syntax without needing access to previously created code. To support large-scale functional

```
// nest.cd
wrap(B) = "(" <body> B ")".
S = <s> wrap(wrap(String)).
I = <i> wrap(wrap(Integer)).

// nest.beh
wrap{{ B inner(){ return body; } }}
```

**Figure 21: Nested Parametrization**

OO development we have created a DemFGen library (`demfgen.lib`) that includes useful parametrized classes (like `List(X)`, `Map(K,V)`, and `Set(X)`), implementing various container classes in a functional OO style. The library is described by a CD file included in the DemeterF distribution that programmers can include and modify to create customized parsers/printers. The DemeterF type checker was developed with the library in this manner, using DemFGen creating the structures and the traversal library to generate and check the constraints discussed in Section 3.4.

# 5. EXAMPLE: EXPRESSION COMPILER

As a more complicated example of the usefulness of DemFGen and DemeterF we discuss the implementation of a simple compiler for the `exp` data structures from Section 2. To make the discussion more interesting we add a new `ifz` ("if zero") conditional construct. We first examine our target data structures, then discuss the source structures and the various operations involved in the transformation from one to the other.

## 5.1 Structures

To build a compiler, we need a definition of our target language representation. In this case the abstract and concrete syntax can be neatly represented using the DemFGen CD notation. Figure 22 shows a CD file that defines our target language: a simple stack based assembly language with labels, math, stack, and control operations/branches.

```
// asm.cd
Op: MathOp | StkOp | CtrlOp.

MathOp: Minus.
StkOp:  Push  | Pop | Def | Undef | Load.
CtrlOp: Label | Jmp | IfNZ.

Minus = "minus".

Push = "push" <i> int.
Pop  = "pop".
Def  = "def".
Undef= "undef".
Load = "load" <i> int.

Label= "label" <id> ident.
Jmp  = "jump" <id> ident.
IfNZ = "ifnzero" <id> ident.
```

**Figure 22: Assembly Structures CD**

We do not show the `asm.beh` file, but the full code for all the examples is available on the web [7]. It contains methods to evaluate a list of `Op`s used in testing our compiler implementation. For the main expression (source) language, we make use of all the assembly operators by adding `ifz` to our data structures.

```
// exp.cd
exp : ifz | def | bin | var | num.
ifz = "ifz" <cnd> exp
      "then" <thn> exp
      "else" <els> exp.
def = <id> ident "=" <e> exp ";" <body> exp.
bin = "(" <op> oper <left> exp <right> exp ")".
var = <id> ident.
num = <val> int.

oper : sub.
sub = "-".
```

**Figure 23: Expression Structures CD**

Figure 23 shows the full CD file, which includes the definitions from Section 2 combined with their concrete syntax and the new `ifz` expression. Once DemFGen has been run and the Java files are compiled, we can use the generated methods to parse an `exp` from a `String` or an `InputStream`. A simple term in this expression syntax would look something like:

```
ifz (- 4 3) then 5 else 7
```

And could be parsed with a Java statement such as:

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

## 5.2 Compiler Classes

For the sake of code organization, we have divided the compiler implementation into four classes: one for each type of expression, and a main class named `Compile`. Figure 24 shows the main compiler class containing a single method, `compile`.

```
class Compile{
  // Compile an Expression File
  static List<Op> compile(String file){
    exp e = exp.parse(new FileInputStream(args));
    return new Traversal(new Cond())
      .traverse(e, List.<ident>create());
  }
}
```

**Figure 24: Main Compile Class**

We produce a `List<Op>` where `List` is actually a functional list implementation from the `demfgen.lib` package. The traversal context starts as an empty `List<ident>`, and will contain the defined variables for each nested expression. The final code generation function class is named `Cond` and an instance is passed when creating the traversal. Figure 25 shows the code generation for math related operators. The static field `empty` and the method `one(...)` simplify the creation of single `Op` lists. As is common in stack based assembly languages we push operands onto the stack, then call an arithmetic operator.

For example, the expression:

```
(- 4 3)
```

will compile into the `Op` list:

```
push 3
push 4
minus
```

```
class Arith extends ID{
  static List<Op> empty = List.create();
  static List<Op> one(Op o){ return empty.append(o); }

  List<Op> combine(sub s){ return one(new Minus()); }

  List<Op> combine(num n, int i)
  { return one(new Push(i)); }
  List<Op> combine(bin b, List<Op> o,
                           List<Op> l, List<Op> r){
    return r.append(l).append(o);
  }
}
```

**Figure 25: Compile for Arith Ops**

The `append(...)` methods within the `List` class return a new list with the given element or list placed on the end; the original list is not mutated.

The `Defs` class in Figure 26 implements the compilation of environment based operators. When compiling a variable reference we generate a `Load` operation with the offset of the identifier in the environment. The `update` method adds a defined variable to the environment stack when traversing into the body of a definition. Once all sub-expressions have been compiled, we wrap the `body` code in `Def`/`Undef` and append it to the code for evaluating the binding.

```
class Defs extends Arith{
  List<ident> update(def d, def.body f, List<ident> s)
  { return s.push(d.id); }
  List<Op> combine(var v, ident id, List<ident> s){
    return one(new Load(s.index(id)));
  }
  List<Op> combine(def d, ident id,
                   List<Op> e, List<Op> b){
    return e.append(new Def())
            .append(b)
            .append(new Undef());
  }
}
```

**Figure 26: Compile for Variables**

The final, more complicated portion of the compiler deals with our conditional expression, implemented in the `Cond` class shown in Figure 27. Because we need to generate unique `Label`s within the generated structures, we use side-effects rather than passing lots of state around. The `synchronized` (locked) method `fresh(...)` creates an unused `ident` in a thread-safe manner. The `IfNZ Op` is used to branch to the *else* portion if the condition is not zero. If the *then* portion was executed we can safely `Jmp` to the *done* label.

The use of the `synchronized` keyword is the only portion of our compiler code that has to do with thread safety; all other parts are completely functional, so we can run our compiler traversal in multiple threads for expressions with multiple sub-expressions. Figure 28 shows the results of running sequentially (using `Traversal`) and in parallel (using `ParTraversal`) on large expressions. Each time is an average of 20 different compiles for a file of the specified number of lines (Size); the times are in milliseconds. Again, the immediate gains are quite promising, but more exploration is needed to completely understand the numbers.

```
class Cond extends Defs{
  int lnum = 0;
  synchronized ident fresh(String s)
  { return new ident(s+"_"+lnum++); }
  List<Op> combine(ifz f, List<Op> c,
                   List<Op> t, List<Op> e){
    ident le = fresh("else"),
          ld = fresh("done");
    return c.append(new IfNZ(le)).append(t)
            .append(new Jmp(ld))
            .append(new Label(le)).append(e)
            .append(new Label(ld));
  }
}
```

**Figure 27: Compile for Conditionals**

| File Size | Seq. | Par. | Speedup |
|---|---|---|---|
| 400 | 182 | 154 | 15.3 % |
| 800 | 255 | 211 | 17.2 % |
| 1200 | 384 | 294 | 23.4 % |

**Figure 28: Parallel Compile Results**

As a final DemeterF example, we present expression simplification. Functional languages are famous for, among other things, their ability to match patterns and optimize programs. Here we examine a DemeterF function class that implements simple constant propagation using the method signature matching of the DemeterF traversal. Figure 29 shows a function class that implements (bottom up) constant propagation for our simple expression language.

```
class ConstProp extends Bc{
  class zero extends num{ zero(){ super(0); } }
  num combine(num n, int i)
  { return (i==0) ? new zero() : n; }

  exp combine(bin b, sub p, exp l, zero r){ return l;}
  exp combine(bin b, sub p, num l, num r)
  { return new num(l.val-r.val); }

  exp combine(ifz f, zero z, exp t, exp e){ return t;}
  exp combine(ifz f, num n, exp t, exp e){ return e; }

  static exp simplify(exp e){
    return new Traversal(new ConstProp())
        .traverse(e);
  }
}
```

**Figure 29: Constant Propagation**

The special cases in our arithmetic language are nicely captured by each `combine` method, the rest of the reconstruction is handled implicitly by `Bc`. Instances of `num` that contain zero are transformed into instances of the more specific inner class `zero`. Subtracting a `zero` from any `exp` yields just the left `exp`; for subtraction consisting of only numbers we can propagate the resulting constant as a new `num`. For `ifz` expressions, our two constant cases for the test, `zero` and `num`, are optimized by returning the *then* and *else* fields, respectively.

## 6. RELATED WORK

Both components of the DemeterF system have ties to AOP, supporting static AOP through to open classes, and dynamic

AOP with function objects and traversals. As mentioned, DemFGen supports static injection of behavior, including a bit of generic programming used to implement functions (like printing) over all data types. DemeterF traversals (similar to DemeterJ [28]) fall under a more traditional AOP model. In [20] the authors discuss the relations of different AO systems, of which DemeterJ is one.

Following their description, we can define the join point model of DemeterF as the entry (for `update` methods) and exit (for `combine` methods) of objects during a depth-first traversal of a data structure. DemeterF function objects can be seen as parametrizable advice, while the control and method signatures are analogous to pointcuts, selecting a set of dynamic join points corresponding to the types that result from the previous executions of advice. Pointcuts are enhanced through programmer controlled traversal contexts and allow programmers to select more join points with later method parameters being optional. In contrast to DemeterJ, we execute only the most specific pointcut/advice at a given join point, similar to Socrates [24].

Our goal is to provide a safer, functional alternative with the power of AOP, while maintaining some of its dynamic flexibility. Due to the functional nature of the traversal, execution of advice affects later join point selection, but function classes can be checked to be sure that applicable advice can always be found. With the use of reflection we eliminate an extra compile step, at the price of runtime penalties. In the future we are interested in exploring the static compilation possibilities that AOP provides in order to reduce reflection overhead.

## 6.1 DemeterF Library

As traversals have been around a long time, there are many different related projects, usually centered around visitors and higher-order functions. The original description of the Visitor Pattern [11] has been implemented in both static and dynamic settings as libraries and tools.

DemeterJ [28, 17] and DJ [25] make up the static and dynamic Demeter imperative visitor tools. DemeterJ compiles static traversal control descriptions in a domain specific language (strategies) and visitor definitions into so-called *adaptive* methods. Depending on the nature of changes to an underlying data structure, the traversal computation can automatically adapt (upon recompilation) without programmer interference. DJ is a traversal library that uses reflection to dynamically traverse objects with control specified using the same strategy language. Visitors perform computation using `before(...)` and `after(...)` methods which are called during a depth-first traversal of a data structure.

In DemeterF, `update` methods take the place of `before` and `combine` methods take the place of `after`, but the major differences between DemeterF and DemeterJ/DJ are its functional traversal computation and type checking. Adaptive methods and visitors in DemeterJ are type checked by the Java compiler, while strategies can be checked to be sure that valid paths in that data structure exist, but due to the side-effecting nature of the visitors, not much about the actual computation to be performed is captured by the definitions themselves. With DemeterF the programmer gives more information about the traversal computation in the form of types. Though this constrains the computation more, making it less adaptive, it also allows us to check more of the programmers assumptions against the data structure, and gives the traversal implementation freedom to order sub-computations. There is a DJ based library that adds functional visitors [30] to traversals, but this library is focused more on integrating functions and control using `around` methods, and distinct types are not used, as `combine` methods accept an `Object` array. This mixes computation and control, which eliminates their ability to check the safety of traversals, though it may be possible to infer control from the function object signatures.

Recent work in visitors has focused on using OO languages with richer type systems. In [10], the language Scala is used to create a visitor library that captures the types involved in functional visitors. The library they describes the return type of a visitor traversal, based on where the traversal code is placed: *internal* to the structure, or *external*, in the visitor. Assuming structures and traversals are implemented correctly by programmers, a visitor program is type safe based on Scala's type safety guarantees. Multi-methods are used to implement visitor methods with traversals that return values, which eliminates the need for mutation. Though their visitor implementations look quite similar to DemeterF function objects, there are differences in traversal flexibility, control, and contexts. In their model, visitors are restricted to return a single type, visitor control must be implemented completely by hand, the equivalent of our `onestep` traversal, and mutually recursive data types require separate visitors that maintain programmer assigned mutual references. In addition to fine grained control, DemeterF provides support for parametrized and mutually recursive data types without programmer intervention. In order to guarantee safety, our system requires an external type checker because of our separate traversal, but eliminates the need for programmers to write any traversal code.

There has been much work in the area of specialized XML processing languages, which share roots in tree transducers and automata [23]. XDuce [12] and CDuce [6] are both functional XML processing languages that use regular-expression types and pattern matching to transform labeled trees. CDuce adds first class and overloaded functions in hopes of allowing programmers to express a larger class of transformations directly in the language. The DemeterF system has similar aims at functional transformations of data structures, but we confine our implementation to a library with code written in a target language (*i.e.*, Java or C#). The type systems used in both XDuce and CDuce result in type safe languages and have given us ideas for developing the DemeterF type system. One of the key goals of DemeterF is to remove traversal code from programs, though it would certainly be interesting to apply our ideas to these and other XML based languages.

In the functional programming community, higher order functions are nothing new, and programmers have been using similar techniques to avoid writing boilerplate traversal code for decades. Theoretical results regarding generalized folds [27, 21] and implementations in the statically typed functional language Haskell have lead to the Scrap Your Boilerplate

(SYB) [15] approach, and Generic Haskell [19]. While our generic traversal is an OO mapping of a higher order function, the ideas of generalized folds can be mapped directly to DemeterF function objects with a small type extension, since the traversal (or fold) of an instance of the same type can return different types of results (*e.g.*, `Min` from Figure 10). Compared to SYB, DemeterF has very similar goals, though we aim to be slightly more general, supporting transformations (via `Bc`) and queries (*i.e.*, folds to a single type) as special cases. Our contribution is an implementation of generalized folds in a (functional) OO setting, with support for separate traversals and control with extensible functions and automatic context passing. Though we suffer somewhat from the explicitly typed syntax of Java, our approach is sufficiently general to support similar fold-like idioms.

## 6.2 DemFGen

DemFGen inherits much of its input syntax from DemeterJ [28] and was originally developed as an upgrade of DemeterJ's existing features. The CD syntax has been simplified to eliminate complex parser annotations and adhoc inheritance in order to support concise traversal type checking. Some notable DemeterJ features missing from DemFGen are common fields for abstract classes and null-able (optional) fields, but these features can be written in a safe way using the parametrization found in DemFGen. One of the major drawbacks of DemeterJ is its limited support for nested parametrized classes: parameters are only substituted a single level, meaning abstract classes with variants cannot truely be parametrized. Concrete classes for each case are generated with parameter classes included in their names (*e.g.*, `Integer_List`), and incorrect uses of parameters are not checked. DemFGen uses generics to represent parametrized classes and instantiates specific uses only for parser and printer genration, providing arbitrary nesting to any depth, and parameter definitions and uses are checked at generation time.

Larger differences exist in the two implementations, as both DemFGen and DemeterJ are implemented in themselves. DemeterJ uses a visitor approach with `OutputStream`s to generate class sources. This can be very difficult to modify, and nearly impossible to parallelize given the size and interaction of the various visitors. In DemFGen we use separate, functional traversals for each aspect of the generated code (classes, parser, printer, and visuals). Since each traversal is independent and eventually uses a number of file operations, we can easily speed up generation, even on a single processor, by using a multi-threaded traversal. One drawback of our dynamic traversal approach is the overhead of Java reflection during sub-traversal and method dispatch, but we are currently exploring static compilation alternatives similar to those used in DemeterJ to minimize reflection and to optimize traversal paths and computation.

Tools like XML Beans [5] and JAXB [3] operate on XML schemas, generating Java classes and using specialized XML parsers to read an XML document into memory. These two implementations specifically focus on generating classes and factories with parsing/printing (unmarshalling/marshalling) available in a library for use with standard XML documents. The schema format accepted by the tools is standardized by the World Wide Web Consortium (W3C) [9], and output classes are structured with empty constructors and set/get methods. In contrast, DemFGen uses a custom schema format, but in other respects is quite similar, though our target is functional OO programs so we do not generate empty constructors or set methods. DemFGen users are free to provide any concrete syntax, even for previously generated classes, which makes it very easy to transform a CD into XML syntax by adding start and end tags.

It is not initially clear whether XML Schema definitions support true generics or parametrized classes. Though a limited form of generic/collection classes can be simulated using unions and/or sub-classing, this can introduce unsafe heterogeneous parametrization. As DemFGen was implemented with generics and type safe parametrization in mind, these features are primary. DemFGen's meta-programming facilities make it simple for us to add factory based construction, though we leave it up to users to develop larger frameworks from the generated classes/interfaces.

There are several tools for parser and tree generation from application specific grammars. Two notable implementations are ANTLR [2] and JJTree, which is part of the JavaCC distribution [4]. Both of these tools are able to create a generic abstract syntax tree (AST) that corresponds to the parsed tokens, allowing programmers to walk the created structure support customizations. It seems possible, though overly verbose, to create a specific data structure, but when specific tree building is needed it is probably easier to hand-code node construction during parsing, rather than using the generic AST creation. When given a CD, DemFGen actually generates parser grammar intended for compilation using JavaCC, but we target the more specific problem of creating a specific tree directly from the input language inferred by the CD. Our traversal library does the walking and the function matching, which frees programmers from writing tests based on types. This forces the programmer to be more verbose when annotating functions, but allows us to check the traversal computation, giving programmers static safety guarantees.

# 7. CONCLUSION

We have introduced DemeterF, an easy to learn library and set of tools for functional adaptive programming, which supports a safe limited version of static and dynamic AOP applied to data structure traversal and significantly improves on previous AP tools. First, DemeterF supports the abstraction of a larger class of traversal functionality, *e.g.*, `Bc`, which supports translations and transformations in a structure-shy manner. Second, DemeterF has a fine-grained typechecker that facilitates the development and evolution of adaptive programs in a controlled way. The typechecker verifies that the flow of information during computation is consistent with the data structures and traversal control. Third, DemeterF supports the manipulation of programmer controlled traversal contexts in a flexible, structure shy way. And fourth, because of its functional nature, DemeterF makes it easy to take immediate advantage of parallel and multi-core processors. Our class generator, DemFGen, supports the traversal library by creating class definitions from concise structural, behavioural, and data generic descriptions, supporting a form of static AOP with enhanced support for parametrized classes and generics. In all Deme-

terF supports a safe limited version static and dynamic AOP applied to data structure traversal.

## 7.1 Future Work

We are currently exploring performance enhancements in the library including static traversal generation and the impact of parallel traversals. We believe that our functional approach is amenable to multi-core architectures even with sequential traversals and we are investigating the comparative performance of functional and traditional OO data structures in various applications.

# 8. REFERENCES


[1] The AspectJ Project. Website. `http://www.eclipse.org/aspectj/`.

[2] ANother Tool for Language Recognition. Website, 2008. `http://www.antlr.org/`.

[3] JAXB reference implementation. Website, 2008. `https://jaxb.dev.java.net/`.

[4] The Java Compiler Compiler™. Website, 2008. `https://javacc.dev.java.net/`.

[5] XML Beans overview. Website, 2008. `http://xmlbeans.apache.org/overview.html`.

[6] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An xml-centric general-purpose language, 2003.

[7] B. Chadwick. Aosd-09 submission example code. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/aosd09/`.

[8] B. Chadwick. DemeterF: The functional adaptive programming library. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

[9] W. W. W. Consortium. Xml schema primer. Website, 2008. `http://www.w3.org/TR/xmlschema-0/`.

[10] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. Accepted at OOPSLA 2008, May 2008.

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

[12] H. Hosoya and B. Pierce. Xduce: A statically typed xml processing language, 2002.

[13] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[15] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. volume 38, pages 26–37. ACM Press, March 2003. Proceedings of the ACM SIGPLAN (TLDI 2003).

[16] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[17] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[18] K. J. Lieberherr and A. J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988.

[19] A. Loeh, J. J. (editors); Dave Clarke, R. Hinze, A. Rodriguez, and J. de Wit. Generic haskell user's guide – version 1.42 (coral). Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

[20] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, pages 2–28, 2003.

[21] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[22] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[23] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.

[24] D. Orleans. Incremental programming with extensible decisions. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64, New York, NY, USA, 2002. ACM.

[25] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[26] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161, New York, NY, USA, 1991. ACM.

[27] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

[28] The Demeter Group. The DemeterJ website. `http://www.ccs.neu.edu/research/demeter`, 2007.

[29] M. Wand. Understanding aspects (extended abstract). In *Proc. ACM SIGPLAN International Conference on Functional Programming*, Aug. 2003.

[30] P. Wu, S. Krishnamurthi, and K. Lieberherr. Traversing recursive object structures: The functional visitor in demeter. In *AOSD 2003, Software engineering Properties for Languages and Aspect Technologies (SPLAT) Workshop*, 2003.