Functional Adaptive Programming

Bryan Chadwick and Karl Lieberherr

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

Abstract. We present a functional formulation of Adaptive Programming (AP), that provides a safe, modular solution to the expression problem. Functional Adaptive Programming (AP-F) maintains the separation of traversal and control of AP with computation encapsulated into *function objects* that fold over data structures with support for programmer controlled traversal contexts. Data structures and functions are independently extensible, while traversals become automatically parallelizable and can be statically checked for correctness. We provide a detailed description of our AP-F implementation, called DemeterF, discussing its key features, type checking, performance, and an extended expression compiler example.

Key words: Data structure traversal, functional adaptive programming, object oriented programming

1 Introduction

Developing large programs that are easy to evolve can be very difficult. The expression problem [25] captures this idea, referring to the fundamental problem of simultaneously extending both a given data hierarchy and operations over it. Object Oriented Programming (OOP) provides inheritance based abstractions allowing classes to share methods and data, providing specialized behavior when needed. Because data and methods are combined, it is easy to add new data variants, but adding new operations requires changes to many separate but related classes. In Functional Programming (FP) we structure programs around functional abstractions, making the addition of new operations simple, but the addition new data types requires modification of previous code.

The visitor pattern [8] has been introduced in order to implement operations over a data hierarchy without the need to modify the underlying classes. The idea has been abstracted further [14] to separate not only operations (*what-to-do*) but also control (*where-to-go*) from traversal (*how-to-get-there*). This allows programs to *adapt* to many structural changes, *e.g.*, new data types, without programmer involvement. The cost of the adaptive traversal, as with most visitor implementations, is we are forced to compute via side-effects. This makes parallelization difficult and can lead to subtle bugs and forced traversal orderings that make verification difficult.

In this paper we present a functional formulation of Adaptive Programming (AP) [14, 13], that provides a safe, modular solution to the expression problem. Functional Adaptive Programming (AP-F) maintains the separation of traversal and control of AP, encapsulating computation into *function objects* that fold over the data structures with support for programmer controlled traversal contexts. In addition to separate, mutation free traversals we get the following additional benefits:

Independent extensibility: inheritance can be used to extend both data structures and functions independently.
Automatic parallelization: separate functional operations makes parallel traversals simple.
Verifiable operations: functional traversal makes the types of arguments and return values explicit so we can varify traversals, even with separated operations and control.

Our Java implementation of AP-F, called DemeterF, includes a library with support for sequential and parallel reflective traversals, and a class generator that gnerates data definitions, static traversal code, and data-generic operations including parsing, printing, equals. The rest of this paper describes the details of AP-F and the DemeterF system. In the next section we introduce a motivating example that highlights the features of AP-F and DemeterF. Section 3 gives an overview of DemeterF traversals including control, contexts, types and general traversal performance. We discuss the DemeterF class generator (DemFGen) in section Section 4. An extended DemeterF example of expression compilation is presented in section Section 5, related work is discussed in section Section 6, and we conclude in section Section 7.

2 Motivation

To motivate our traversal implementation and abstractions, we discuss the implementation of evaluation for data structures representing arithmetic expressions. Figure 1 shows Java syntax that describes a version of the classes/structures involved. For the sake of brevity we elide constructors and access modifiers (e.g., `public`). These structures represent a simple language with a chain of variable definitions followed by an expression. With these classes as our abstract syntax, we 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"))))
```

To calculate the final value of a given `Exp` we recursively evaluate each `Def`, appling the accumulated bindings in each `body` expression. This operation, `eval`,

```
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{}
```

Fig. 1. Simple Expression Structures

can be implemented in a number of ways; Figure 2 shows a straightforward OO-style implementation, assuming a functional implementation of environments, `Env`. Comments describe the class where each method belongs. For each abstract class we introduce an abstract method; for each concrete class we implement the specific version of `eval(·)`, recursively calling where needed.

```
/* 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))); }
```

Fig. 2. OO `Exp` Internal Evaluation

Figures 1 and 2 demonstrate the difficulties of adding operations to a hierarchy of classes, as the implementation of a single operation is distributed throughout, interleaving both structural and behavioral concerns. There are also operations such as parsing, printing, and equality, that can be written generically based on the structures themselves, rather than specific instances. To support concise structural specification, the DemeterF class generator (DemFGen) merges separate descriptions of class definitions (structure, behavior, and data-generic operations) into compilable code. An added benefit of our organization is that we can provide different target language modules for DemFGen; in particular, we currently support both Java and C# code generation.

As for the implementation of the `eval` operation itself, a few key issues emerge from this example:

1. *Cooperating methods are scattered*: the OO implementation of `eval(·)` is spread throughout the classes.

2. *The method argument is passed everywhere, but not used often*: here the environment is only needed for evaluation of `Def` and `Var` instances (definitions and uses).
3. *Recursive calls are explicit*: the structure is encoded in both the class definitions and the `eval` implementation. This code is brittle with respect to the structural changes.
4. *Traversal order and direction is hidden*: the order of the recursive `eval` results is not obvious in the body of compound classes.

While inter-type declarations [10] can solve the first issue, visitor-based techniques [13, 14, 20, 21] have been developed in order to address the others. In its original incarnation, the Visitor Pattern [8] 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 `eval`.

```
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();
  }
}
```

Fig. 3. Visitor `Exp` Evaluation

Encoding the traversal in the visitor leaves the target hierarchy untouched, but in order to maintain flexibility in the return types of accept methods the programmer is forced to use mutation, which constrains the traversal order further. In this case, values on the stack must be pushed/popped in the correct order; traversing the `right` expression of a `Bin` first, so the `left` result is at the top of the stack when needed. The use of side-effects does eliminate the passing of the environment, but there is no warning when data structures change. If we attempt add a new variant of `Exp` the visitor continues to compute a value, though it will likely be incorrect. In contrast to the compositional nature of the

hand-written `eval`, the visitor's side-effects are very sensitive to traversal order and difficult to separate into multiple threads.

To eliminate these problems associated with both visitors and hand-coded traversals, DemeterF provides a generic (ataptive) traversal that is parametrized by two objects: a *function object* that computes over the traversal, and a *control object* that directs the traversal. The first is responsible for updating context information (*i.e.*, a traversal argument) and folding together of recursive results. The second 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 base DemeterF class that implements the identity function for contexts and primitive types. To evaluate an expression, we create a `Traversal` object that uses `this` instance to compute, bypassing the `body` field of `Def` instances.

```
class Eval extends ID{
  Traversal trav;
  Eval(){ trav = new Traversal(this, Control.bypass("Def.body")); }

  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)); }
}
```

Fig. 4. DemeterF `Exp` Evaluation

After recursively traversing the selected fields of each instance, our general traversal passes the results along with the context (`Env`) to the most specific `combine` method. The original object is included as the first parameter, allowing methods to match the type of the parent instance. Operationally, we update contexts while traversing down the object tree, computing results at leafs (*e.g.*, `int`) and pushing the them up to containing objects through `combine` methods. As a result, the computation mentions the context and restarts traversal only when needed. The `Eval` class can be extended with new `combine` methods using inheritance when new variants of `Exp` are defined, while the functional nature of our computation allows the implicit traversal to execute in parallel. More information is provided by the methods about expected traversal argument and return types, which can be checked to ensure safety.

3 DemeterF Traversals

To describe the details of DemeterF traversals 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;
}
```

Fig. 5. BST Structures

The main contribution of the DemeterF system is its traversal library. The library includes classes that implement a generic, depth-first traversal over an instance of a data structure. The traversal is parametrized by two objects: a *function object*, which advises the traversal using specially named methods, and a *control object* that guides the traversal through the structures. When creating a `Traversal`, the programmer passes instances of `ID` (the *identity* function object) and `Control`; for programmer convenience the default `Control` permits traversal everywhere when none is given.

3.1 Function Objects

Function classes (subclasses of `ID`) implement `combine` methods that fold together recursive traversal results, and `update` methods that maintain a traversal context. The `ID` implementation includes identity `combine` methods for primitive types (*i.e.*, `int`, `boolean`, *etc.*), and an identity `update` method for contexts. The first parameter passed to a combine method is the original object being traversed and the last is the traversal context, if used. Others are the result of recursively traversing each of the object's fields. Our default traversal implementation uses reflection, so results are passed to methods in the source order of the corresponding fields.

```
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);
  }
}
```

Fig. 6. BST `toString`

Figure 6 shows a simple `toString` function over BSTs that returns a string of nested named parenthesis representing the given tree. The `static` method `ToString.toString(·)` creates a new `Traversal`, passing a `ToString` function object. The default `Control` directs the traversal everywhere, which corresponds to the static creator `Control.everywhere()`. The `traverse` method is called, passing the object we wish to traverse; in this case no context is used.

For traversal contexts, `update` methods are called before traversing each field of an object. The function object can 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);
  }
}
```

Fig. 7. BST Top Down `Height`

The `update` method parameters include a tag that describes which fields it corresponds to; here the traversal will call our method for any field of a `Node`. When calling the `traverse` method we pass a second argument that represents the root context. When calling a `combine` method, the traversal context is passed as the last parameter, 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 choose the maximum depth of the `left` and `right` `BST`s. Note that the context is not mentioned in the `combine` method for `Node`s; for programmer convenience unused parameters can be left off the end of `update` and `combine` method signatures.

3.2 Control

DemeterF provides separate traversal control similar to that found in DemeterJ and DJ[24, 20], but `Control` instances are created using static methods, rather than a domain specific language[1].

Figure 8 shows a function class that traverses a `BST` using the `onestep` traversal. The static creator `Traversal.onestep()` returns a traversal that allows the

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

```
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);
  }
}
```

Fig. 8. BST `IsLeaf`

programmer to step into an object and retrieve the values of its fields as parameters for `combine` matching. The function only needs the first parameter to determine the result, so the others are not included in the method signature for `Node`. Using `onestep`, we can eliminate field accesses and instance checks, letting the traversal matching to do all the hard work, similar to pattern matching in functional programming languages.

```
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); }
}
```

Fig. 9. BST `Insert`

Figure 9 shows a class that implements functional `insert` for `BST`s using the `onestep` traversal. In this case the traversal context is used to pass the integer 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; at a `Node`, we compare the inserted value to the current data, `d`, recursing into the correct sub-tree and reconstructing the resulting `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 contains a traversal implementation of minimum, returning the smallest value stored in a given `Node`. To guide the traversal we pass a control object, constructed with `Control.only(·)` that tells the traversal to *only* recurse into the `left` field of a `Node`, which is where the minimum will be. To eliminate conditionals from our methods we return the left-most `Leaf` and match based on the type calculated from the `left` field of a `Node`. When the `left` tree is a `Leaf` we return the current `data`, otherwise we return the recursive result.

```
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(Node n){
    return new Traversal(new Min(),
      Control.only("Node.left")).<Integer>traverse(n);
  }
}
```

Fig. 10. BST `Min`

The opposite of `Control.only` is the creator `Control.bypass`, describing which fields should *not* be traversed. Figure 11 shows a similar implementation that returns the maximum value in a `Node`. 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 cases in multiple methods.

```
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(Node n){
    return new Traversal(new Max(),
      Control.bypass("Node.left")).<Integer>traverse(n);
  }
}
```

Fig. 11. BST `Max`

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, but leaving it otherwise unchanged. To support functional updates DemeterF provides a function class, `Bc` (the *Building combiner*), that reconstructs a copy of the traversed structure. The provided `combine` methods can then be overridden to transform a particular type and `Control` can be used to change just a portion of a structure.

As an example, Figure 12 shows a function class that extends `Bc`, incrementing each `int` (`data` fields) in a given `BST`. One of the benefits of extending `Bc` is the fact that it can easily adapt to structural changes and adjustments in traversal control, which can be used to limit the extent of a transformation. Figure 13 shows a method that uses our `Incr` function class to increment just the `right` spine of a given `BST`. We use `Control.bypass(·)` since `Node.data` must also be traversed.

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

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

Fig. 12. BST Increment

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

Fig. 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 BST invariant.

```
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);
  }
}
```

Fig. 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 specification (function signatures, data structures, a control description, and a starting class) we can type-check the functions with respect to the 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, though all methods return the same type.

Each `combine` method signature places constraints on traversals of related types: the return types constrain the type of a traversal of an instance of the first

parameter type; the other parameter types place constraints on the return types of traversals of each of the class' field types (in order). In the case of `ToString`, the constraints from the method parameter types become:

· Traversal of a `BST` should return a `String`
· Traversal of an `int` should return an `int`

from the parameters of the `combine` method matching `Node`. We conclude from the method return types that traversal of a `BST` (`Node` or `Leaf`) returns a `String`, and the implementation of `ID` contains a method for `int` that returns `int`, so the traversal is correct.

For `Min` and `Max` (Figures 10 and 11) the situation is more complicated, since the return types of the `Leaf` and `Node` `combine` methods are different, but the algorithm is the same. For `Min` the `Control` specifies that only the `left` field of a `Node` will be traversed, so we can infer constraints from the method parameters:

· Traversal of an `int` should return an `int`
· Traversal of a `BST` should return one of {`int`, `Leaf`}

while the return types of the methods give us information about what a traversal will return:

· Traversal of a `Leaf` returns a `Leaf`
· Traversal of a `Node` returns an `int`

From these we conclude that traversal of a `BST` can return one of `int` or `Leaf` (the union of the subclass return types), which matches the second constraint, and `ID` contains a method that covers the first constraint. Since traversal begins at a `Node`, the whole traversal is correctly assigned the type `int`.

With recursive type uses (*e.g.*, `Node.left`) the constraints are required to capture parameter types in unknown (recursive) field/parameter positions. After observing the `combine` return types, they can be checked against the parameter types to ensure all possibilities are handled. The DemeterF type checker implementation recursively traverses structure definitions from the starting class; for concrete classes without recursive uses (identified using a stack) we check the corresponding `combine` method using the types of field traversals, but for recursive uses, we inspect the possibly applicable `combine` methods, and constrain the unknown return type to the possible parameter types in the correct position. The traversal return of an abstract class is simply the union of the return types of its subclasses.

Generated constraints are of the form:

$$\text{Traverse}(\texttt{C}) <: \texttt{D}$$

where `D` is a *set* of classes (possible return types) and $<:$ is defined over *sets* of classes:

$$X <: Y \quad \text{if } \forall\, x \in X\,.\,\exists y \in Y\,.\; x \leq y$$

with $\leq$ defined between classes as the reflexive, transitive closure of the inheritance relation (`extends`) from the the program. 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 from the function object will never fail.

Similar constraint based type-checking systems have been used to type-check pure object oriented programs [22], though here we solve a slightly simpler problem. Following these ideas, our type checker computes the return type of a given traversal and 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 all the examples here are available with the rest of the code [5].

3.5 Performance

One of the major draws of functional programming is its potential for parallel computation. The DemeterF traversal model was designed to separate traversal and computation to allow programmers to substitute a multi-threaded traversal (`ParTraversal`) for any `Traversal` instance without affecting results. As a preliminary performance test we show a `BST` sum operation and a series of DemeterF traversal runs on a dual-core machine. 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())); }
}
```

Fig. 15. BST Sequential and Parallel `Sum`

To measure the performance of various traversal schemes, we ran a series of Java tests with `BST`s of various heights using multiple traversal schemes. Figure 16 shows our preliminary results using double dispatch visitor (**Visitor**), reflective sequential (**Reflect**), statically generated sequential (**Static**), and reflective parallel (**Par.**) traversals. The slowdowns of reflective sequential and parallel traversals are given in the next columns as a multiple of the visitor times. Each time is is in milliseconds averaged over 10 runs of the given tree size using Sun's JRE 1.6 on a Dell Latitude (laptop) with a 2.26 Ghz Intel Core 2 Duo Processor.

The visitor solution is very fast, since it requires minimal method calls and delegation. The modular design of the DemeterF traversal library accounts for

BST Size	Visitor	Reflect	Static	Par.	Ref. Slwdn	St. Slwdn	Par. Spdup
2^8	2	59	61	65	29 x	32 x	-3.3 %
2^9	2	80	75	66	40 x	33 x	6.3 %
2^{10}	2	102	87	86	51 x	43 x	14.7 %
2^{11}	3	182	119	106	60 x	35 x	34.6 %
2^{12}	3	255	165	141	85 x	47 x	35.3 %

Fig. 16. Performance of BST Sum: Standard-Visitors, Reflective-Sequential, Static-Sequential, and Reflective-Parallel Traversals.

most of the slowdowns due to the depth of method calls in the implementation. Our reflective numbers are much better than similar reflective visitor results [21] though only by a constant factor. More interesting than the visitor numbers is a comparison of reflective, static, and parallel DemeterF traversals. The static traversal generates less garbage and eliminates the need for reflection, while the parallel traversal consistently outperforms both sequential versions.

For this simple benchmark we approach 40% parallel speedup (maximum is 50% with 2 processor cores). Profiling reveals that the majority of slow downs come from garbage collection, as the traversal and dispatch methods in the library tend to create many new objects.

4 Data Description

In order to generate classes specifically for use with DemeterF traversals and avoid having to hand write various common structure based methods (*e.g.*, `parse`, `equals`, *etc.*) we created the DemeterF class generator, DemFGen. Incidentally, the generator is written in DemeterF, and has been a good benchmark test of the library, the type checker, and now, itself. In the spirit of other adaptive programming tools such as DemeterJ [24], our class generator accepts a class dictionary file (CD) that specifies the structure of data types and a behavior file (BEH) that describes static code to be injected into the generated classes. Our major improvements are relate to generic classes and the ability to write data generic functions over CDs.

4.1 CD and BEH Files

Our class dictionary syntax is slightly simplified from DemeterJ; Figure 17 shows a CD file that describes the `BST` structures defined earlier (from Figure 5). 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. Interfaces can be declared with the `interface` keyword, but are otherwise the same as abstract classes.

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

Fig. 17. BST Class Dictionary

In order to support modular CD and BEH files DemFGen allows the inclusion of other CD and BEH files using `include` statements, and definitions can be preceded by `noparse`, `nogen`, or `extern` to suppress parser, class, or all generation respectively. As in DemeterJ, specifying succinct class structures requires a separate file for injecting class behavior, similar to the idea of open classes. Figure 18 shows a BEH file that completes the `BST` class definition by inserting a few method stubs calling our earlier implementations.

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

Fig. 18. BST Added Behavior

4.2 Parametrized Classes

DemFGen's most novel feature is the support for Java generics using parametrized classes. We support type parameter bounds, any depth of nested type parameters, and parser and printing generation for generic 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)) = .
```

Fig. 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 for the checking of uses and type parameters in the other definitions. 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).
```

Fig. 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 describe a generic parenthesis `Wrap` class. The uses of `Wrap` allow us to parse `String`s and `Integer`s within two sets of parenthesis. When nesting parametrized classes we must avoid left recursion and other pitfalls dealing with recursive parameter substitution [15].

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

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

Fig. 21. Nested Parametrization

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 previous code. To support large-scale functional 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 that programmers can modify to create customized parsers/printers, taking advantage of the library classes and methods. The DemeterF type checker was developed with the library, using DemFGen to create type 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 using DemeterF, in this section we discuss the implementation of a simple expression compiler for the `Exp` structures from Section 2. To make the discussion more interesting we add a new `Ifz` (*if zero*)

conditional expression. We first examine our target data structures, then discuss the source structures and the different operations involved in the transformation from one to the other.

5.1 Structures

To build a compiler we need source and target language representations. In this case the abstract and concrete syntax can be described with a few simple CD files. Figure 22 shows a CD file that defines our target language: a simple stack based assembly language with labels, subtraction, stack, and control operations.

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

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

Minus = "minus".
```

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

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

Fig. 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 [5]. It contains methods to evaluate a list of `Ops` used in testing our compiler implementation. For the source expression language, we make use of all the assembly operators by adding a new expression variant, `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 = "-".
```

Fig. 23. Expression Structures CD

Figure 23 shows the full expression CD file, which includes the definitions from Section 2 including their concrete syntax and the new `Ifz` expression. Once the classes have been generated and compiled, we can use the generated static 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 can be parsed with the Java statement:

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

5.2 Compiler Classes

For the sake of code organization and demonstration, we have split the compiler implementation into four classes: one for each category 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(file));
    return new Traversal(new Cond()).traverse(e, List.<ident>create());
  }
}
```

Fig. 24. Main Compile Class

The compiler constructs an `Op` list (`List<Op>`) where `List` is a functional implementation provided in the `demfgen.lib` package. The root traversal context is an empty `List<ident>`, and will contain the local variables for nested definitions. The final code generation function class is named `Cond` and an instance is passed when creating the traversal.

```
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);
  }
}
```

Fig. 25. Compile for Arith Ops

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. 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 changed. 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
```

The `Defs` class in Figure 26 implements the compilation of environment related operations. 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, the code for the `body` expression is wrapped in `Define`/`Undef` and appended 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 Define()).append(b)
            .append(new Undef());
  }
}
```

Fig. 26. Compile for Variables

The final, more complicated extension of the compiler deals with our conditional expression, implemented in the `Cond` class in Figure 27. We use mutation in order to generate unique `Label`s within the generated code, as the `synchronized` (locked) method `fresh(·)` creates an new `ident` in a thread-safe manner. The `IfNZ` operation is used to branch to the *else* portion if the condition is not zero. Otherwise the *then* code was executed and we can safely `Jmp` to the *done* label.

The `synchronized` keyword is the only portion of our compiler 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 expression files. As in Section 3.5, times are in milliseconds and each is an average of 10 different runs on a file of the specified number of lines. Again, the immediate gains are very promising, though the library needs to be optimized more to reach its full potential. In this case garbage collection tends to add a significant amount of time when traversing large expressions.

The expression compiler is separated into four classes to make things easier to develop, but it also demonstrates the flexibility of our approach. Each expression variant corresponds to a `combine` method in the body of some function class;

```
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));
    }
}
```

Fig. 27. Compile for Conditionals

File Size	Sequential	Parallel	Speedup
417	136	118	13.2 %
837	204	164	19.6 %
1256	291	220	24.4 %

Fig. 28. Compiler Performance Results

we begin with `ID` and provide extensions for new expression types. As with pure OOP, adding new variants is easy; we only need to extend a function class.

Similar to functional languages and visitor based approaches, adding new functions is also easy. 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 function class that implements expression simplification using method matching. Figure 29 shows a function class that implements the bottom up simplification of constant expressions.

```
class Simplify 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 Simplify()).traverse(e);
  }
}
```

Fig. 29. Expression Simplification

The special cases in our arithmetic language are nicely captured by each `combine` method, the rest of the reconstruction is handled implicitly by the provided class `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

The library and class generation components of the DemeterF system have ties to Aspect Oriented Programming (AOP) [11], supporting static AOP similar to open classes, and dynamic AOP with function objects and traversals. DemeterF traversals and function objects are similar to visitors [24, 20, 21] and higher-order functions [12, 23, 18, 16] while DemFGen resembles data binding tools [24, 4, 2] and includes a form of generic programming [9] used to implement static traversals and functions (like printing) over all data types.

6.1 DemeterF Library

In [17] the authors discuss the relations of different AOP systems, of which DemeterJ is one. Similarly, 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 parametrized 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, while ignoring later `combine` method parameters allows programmers to select more join points. In contrast to DemeterJ, we execute only the most specific pointcut/advice at a given join point, similar to Fred and Socrates [19].

Our goal is to provide a safer, functional alternative with much of the power of AOP, while maintaining some of its dynamic flexibility. Due to the functional nature of our traversal, execution of `combine`s affects later method selection, but function classes can be checked to be sure that applicable methods can always be found. With the use of reflection we eliminate an extra compile step, at the price of runtime penalties. We are currently exploring the static compilation possibilities in order to reduce reflection overhead.

DemeterJ [24, 14] and DJ [20] make up the static and dynamic Demeter imperative visitor tools. DemeterJ compiles static traversal control descriptions (in a strategy language) and visitor definitions into so-called *adaptive* methods. For certain changes to underlying data structures 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 return `void` and are called during a depth-first traversal of the 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 computation and type checking. Adaptive methods and visitors in DemeterJ are type checked by the Java compiler and strategies can be checked to be sure that valid paths exist, but due to the side-effecting nature of visitors, not much computational information captured by the method types. With DemeterF the programmer gives more information about the traversal computation and method interaction. 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 (re)order sub-computations. There is a DJ based library that adds functional visitors [26] 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.

Recent work in visitors has focused on using OO languages with richer type systems. In [7], the language Scala is used to create a visitor library that captures the types involved in functional visitors. The library 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, eliminating the need for mutation. Though their visitor implementations look quite similar to DemeterF function objects, there are differences in traversal flexibility, control, and contexts. 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 check traversal safety, we require a type checker after compilation, but programmers can usually eliminate traversal code from their function classes.

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 [23, 18] and implementations in the statically typed functional language Haskell have lead to the Scrap Your Boilerplate (SYB) [12] approach, Generic Haskell [16], and a comparison of design patterns and datatype-generic programming [9]. 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 results (*e.g.*, `Min` from Fig-

ure 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 an OOP setting with support for separate traversals, control and extensible functions with automatic context passing. The genericity of our traversal function allows it to adapt to different datatype shapes, and DemFGen can generate specific static traversal code for a given CD. Though we suffer somewhat from the explicitly typed syntax of Java, our approach is sufficiently general to support similar fold-like and data generic idioms.

6.2 DemFGen

DemFGen inherits much of its input syntax from DemeterJ [24] and was originally developed as an upgrade of DemeterJ's existing features. The CD syntax has been simplified to eliminate complex parser annotations and ad hoc inheritance to support concise traversal type checking. Some notable DemeterJ features missing from DemFGen are common fields for abstract classes and null-able (optional) fields, though these features can be written in a safe way using our improved parametrization. One of the major drawbacks of DemeterJ is its limited support for nested parametrized classes: parameters are only substituted a single level so variants of parametrized abstract classes cannot truly be parametrized. Concrete classes for each case are generated with parameters included in their names (*e.g.*, `Integer_List`), and incorrect uses of parameters are not checked. DemFGen uses generics to represent parametrized classes instantiating specific functions when needed (*e.g.*, parsing, printing, and static traversal) with nesting to any depth. 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, *etc.*). 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 subtraversal and method dispatch, but we are currently exploring static compilation alternatives to minimize reflection and optimize traversal paths and computation similar to those used in DemeterJ, DAJ, XAspects.

Tools like XML Beans [4] and JAXB [2] 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) [6], 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 metaprogramming 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 code.

There are several tools for parser and tree generation from application specific grammars. Two notable implementations are ANTLR [1] and JJTree, which is part of the JavaCC distribution [3]. 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. It seems possible, though overly verbose, to construct a concrete data structure from the resulting AST using type casts, but when specific tree building is needed it is easier to actually construct nodes during parsing. When given a CD, DemFGen actually generates parser grammars intended for compilation using JavaCC. 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, a functional formulation of adaptive programming. The implementation includes a library and set of tools for safe, flexible, parallel traversals, providing a complete solution to the expression problem. DemeterF supports the abstraction of a large class of traversal functionality with the addition of a fine-grained type checker that facilitates the development and evolution of traversal based programs in a controlled way. The type checker ensures that the flow of information during computation is consistent with the structures and control of the traversal.

DemeterF also supports programmer controlled traversal contexts (arguments) and because of its functional nature, traversals can automatically take advantage of parallel, multi-core processors. Our class generator, DemFGen, supports the traversal library by creating class definitions from concise structural, behavioral, and data generic descriptions with enhanced support for parametrized classes and generics.

7.1 Future Work

We are currently exploring performance enhancements in the library including static method selection 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.

References

1. ANother Tool for Language Recognition. Website, 2008. `http://www.antlr.org/`.
2. JAXB reference implementation. Website, 2008. `https://jaxb.dev.java.net/`.
3. The Java Compiler Compiler™. Website, 2008. `https://javacc.dev.java.net/`.
4. XML Beans overview. Website, 2008. `http://xmlbeans.apache.org/overview.html`.
5. B. Chadwick. Ecoop-09 submission example code. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/ecoop09/`.
6. W. W. W. Consortium. Xml schema primer. Website, 2008. `http://www.w3.org/TR/xmlschema-0/`.
7. 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.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. J. Gibbons. Design patterns as higher-order datatype-generic programs. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 1–12, 2006.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *European Conference on Object Oriented Programming*, Budapest, 2001. Springer Verlag.
11. 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.
12. 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).
13. 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.
14. 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.
15. 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.
16. 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.
17. H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, pages 2–28, 2003.

18. 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.
19. 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.
20. 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.
21. J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, Washington, DC, USA, 1998.
22. 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.
23. 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.
24. The Demeter Group. The DemeterJ website. `http://www.ccs.neu.edu/research/demeter`, 2007.
25. M. Torgersen. The expression problem revisited. 2004.
26. 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.